Space and energy efficient molecular programming and space efficient text indexing methods for sequence alignment

by

Christopher Joseph Thachuk

M.Sc., Simon Fraser University, 2007
B.C.S., The University of Windsor, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2013

Abstract

Nucleic acids play vital roles in the cell by virtue of the information encoded into their nucleotide sequence and the folded structures they form. Given their propensity to alter their shape over time under changing environmental conditions, an RNA molecule will fold through a series of structures called a folding pathway. As this is a thermodynamically-driven probabilistic process, folding pathways tend to avoid high energy structures and those which do are said to have a low energy barrier.

In the first part of this thesis, we study the problem of predicting low energy barrier folding pathways of a nucleic acid strand. We show various restrictions of the problem are computationally intractable, unless $\mathsf{P} = \mathsf{NP}$. We propose an exact algorithm that has exponential worst-case runtime, but uses only polynomial space and performs well in practice. Motivated by recent applications in molecular programming we also consider a number of related problems that leverage folding pathways to perform computation. We show that verifying the correctness of these systems is PSPACE-hard and in doing so show that predicting low energy barrier folding pathways of multiple interacting strands is PSPACE-complete. We explore the computational limits of this class of molecular programs which are capable, in principle, of logically reversible and thus energy efficient computation. We demonstrate that a space and energy efficient molecular program of this class can be constructed to solve any problem in SPACE—the class of all space-bounded problems. We prove a number of limits to deterministic and also to space efficient computation of molecular programs that leverage folding pathways, and show limits for more general classes.

In the second part of this thesis, we continue the study of algorithms and data structures for predicting properties of nucleic acids, but with quite different motivations pertaining to sequence rather than structure. We design a number of compressed text indexes that improve pattern matching queries in light of common biological events such as single nucleotide polymorphisms in genomes and alternative splicing in transcriptomes. Our text indexes and associated algorithms have the potential for use in alignment of sequencing data to reference sequences.

Preface

The candidate contributed to all major ideas and writing of the published manuscripts and wrote all chapters of this thesis. We now detail the contributions of the candidate in published articles resulting from this work. Non-thesis related work published by the candidate during the course of their degree is not listed.

Research from Part I of the thesis was conducted in collaboration with a number of co-authors; primarily the candidate's supervisor Dr. Anne Condon and also Dr. Jan Maňuch. In no instance was a co-author a graduate student. Other work presented in this thesis part that is not yet published was conducted under the supervision of Dr. Anne Condon and written by the candidate.

- The introductory chapter of Part I was written by the candidate, but uses selected content from publications that he co-authored [25, 26, 80, 81, 129, 131]. Details of the contribution for each of these publications is given below.
- A version of Chapter 2 has been published in the proceedings of the 15th Annual International Conference on DNA Computing and Molecular Programming (2009) [80] and also the Journal of Natural Computing (2011) [81]. The candidate collaborated with co-authors in developing the reduction proof. The candidate contributed to writing the manuscripts, particularly the journal version [81].
- A version of Chapter 3 has been published in the proceedings of the Pacific Symposium of Biocomputing (2010) [129]. The candidate was the primary researcher to design and implement the main algorithm of the paper and performed all experiments. He also contributed to writing the manuscript. The presentation of the algorithm and correctness proofs have been rewritten by the candidate in the thesis to facilitate additional results.
- A version of Chapter 4 appeared in the proceedings of the 17th Annual International Conference on DNA Computing and Molecular Programming (2011) [25] and also the Journal of the Royal Society: Interface Focus (2012) [26]. The candidate contributed to all aspect of research and was one of the main writing authors contributing various sections of the manuscript. Some alternate proofs and additional results have been given by the candidate in the thesis version to provide deeper insight of the topics considered.

- A version of Chapter 5 appears in the proceedings of the 18th Annual International Conference on DNA Computing and Molecular Programming (2012) [131] and was awarded best student paper. The candidate contributed to all aspects of research and in particular developed the SAT verification procedure, integrated it with the tree traversal procedure proposed by the co-author, proved the correctness of the result, extended the result into a DSD implementation, and formally proved the computational hardness of a number of related problems. The manuscript was written by the candidate. The results of this chapter have been significantly enhanced and supplemented compared with the published version.

All research from Part II of the thesis was conducted independently by the candidate and all chapters and published manuscripts were written by the candidate.

- Versions of Chapter 8 and Chapter 9 have been published in the proceedings of the 22nd Annual International Symposium on Combinatorial Pattern Matching (2011) [127] and the journal Theoretical Computer Science [128]. The candidate was the sole author and the paper won best student paper.
- A version of Chapter 10 was published in the proceedings of the 18th Annual International Symposium on String Processing and Information Retrieval (2011) [130]. The candidate was the sole author.
- Chapter 7, the introductory chapter of the part, used content from the previously mentioned papers published by the candidate[127, 130].

Table of contents

List of tables

List of figures

Glossary

bandwidth Given a CRN $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}} \rangle$, B_s, the *bandwidth* of signal species $s \in S$ is the maximum number of copies of s that appears in a multiset I of any reaction $(I, P) \in \mathcal{R}$. The bandwidth of $\mathbf{C}$ is the sum of bandwidths for all signal species in S. xviii, 79

blunt-end displacement Any attempted toehold mediated strand displacement where the invading strand does not first bind its toehold domain to the template strand. xviii, 9

bound strand A strand where one or more of its bases are paired to other bases on a template strand. xviii, xx, xxi, xxiii, 8, 18, 20, *see also* unbound strand

BWT Burrows-Wheeler transform. xviii

catalyst A signal molecule that is required to be present for the application of a corresponding chemical reaction equation. It is not consumed nor produced when acting as a catalyst. In an equivalent interpretation, it is both consumed and produced by the application of the corresponding chemical reaction equation. xviii, xix, xxi, 11, 13, 67, 70

chemical reaction equation Either a reversible chemical reaction equation or an irreversible chemical reaction equation. xviii–xxv, 11, 13, 14, 17, 18, 20, 65, 70

Chemical Reaction Network (CRN) Consists of an initial signal multiset and a set of chemical reaction equations. Formally, we define a CRN to be a tuple $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}} \rangle$, where

- S is the set of all signal types (or species) of signal molecules used in any reaction.
- $\mathcal{R}$ is a set of chemical reaction equations, where each $R \in \mathcal{R}$ is an ordered pair of multisets of signal molecules. Intuitively, a reaction equation $R = (I, P)$ consumes the signal molecules in I as the input reactants and produces the signal molecules in P as products. Note that it is only the signal molecules in $I - P$ that are actually consumed. The others act as catalysts for the reaction. Our formalism is directional to allow modeling non-reversible reactions; a reversible chemical reaction is modeled as two separate elements of $\mathcal{R}$, *i.e.,* (I, P) and (P, I).

- S_0 is the initial signal multiset where $s \in S_0 \rightarrow s \in S$.
- $s_{\text{end}} \in S$ is a signal molecule denoting the end of computation.

. xviii–xx, xxii, xxiv, 11, 13, 14, 16, 18, 20, 64, 65, 69, 70, 73, 79, 85, 90

closed system A system, such as a CRN or DSD, is *closed* if no outside interference can occur such as the removal or addition of signals. xviii, xxv, 11, 13, 17, 22, 64, 66, 68, 69

configuration graph A *configuration graph* for a computation has a node for every possible state on every possible input for the underlying Turing machine being modeled. There is a directed edge from node i to node j if and only if state j is reachable from state i in a single state-transition of the Turing machine. xviii, xxi, 20

consumed A signal molecule is said to be *consumed* when it is removed from the current signal multiset due to the application of a chemical reaction equation. xviii, xix, xxi–xxiv, 11, 14, 18, 20, 69, 71, 72

crosstalk In molecular programs, *crosstalk* occurs when signals from different copies of the system present in the same reaction volume interfere with the intended sequence of reactions that would occur if only a single copy were present.. xviii, 79

current signal multiset The multiset of all signal molecules currently present in the reaction volume. xviii, xx, xxii, xxiii, 11, 13

DNA Strand Displacement system (DSD) Consists of one or more signal strands and double stranded complexes. Signal strands can be consumed and produced by means of toehold mediated strand displacement. xviii, xx, xxii–xxiv, 8, 11, 14, 20, 64–66, 70–73, 85

double stranded complex One template strand base paired with one or more bound strands. xviii, xx, xxiv, 8, 14, *see also* template strand & bound strand

energy efficient computation In a logically reversible computation there is no inherent lower bound on the required amount of energy lost to complete the computation. We call such a computation *energy efficient*. xviii, 20, *see also* logically reversible computation

evading strand The bound strand of a toehold mediated strand displacement. xviii, xxi, xxiv, 9

fuel See transformer. xviii, xx, 66, 68, 69

fuel-depletion Denotes a scenario where an insufficient amount of fuel or copies of transformers are available to complete a computation. xviii, 66, 68

illegal displacement Any toehold mediated strand displacement that is not a legal displacement. xviii, 9, *see also* mismatch displacement, blunt-end displacement & spontaneous displacement

initial signal multiset The multiset of all signal molecules present in the reaction volume prior to the application of any chemical reaction equations. xviii–xx, xxiii, 11, 13, 14, 16–18, 65, 68–70, 73, 90, 92

initial tag multiset The multiset of all tags present in the reaction volume prior to the application of any chemical reaction equations. xviii, xxiii, xxiv, 16–18, 66, 68, 82, 90

invading strand The signal strand that binds to a template strand in order to displace a currently bound strand during a toehold mediated strand displacement. xviii, xix, xxi, xxiv, 9

irreversible chemical reaction equation Specifies an event that can consume a multiset of reactants and a produce a multiset of products. Some of the specified reactants may be catalysts and are therefore also produced as products.. xviii, xix, xxii, xxiii, 11

legal displacement Any toehold mediated strand displacement where the invading strand first binds its toehold domain to the template strand and the adjacent long domain of the invading strand involved in three way branch migration is identical to respective long domain of the evading strand that is currently bound to the template. xviii, xxi, 9, 20, 72

logically reversible computation A *logically-reversible-computation* is a form of deterministic computation where the configuration graph induced on any particular input forms a chain and each node i along the chain has a directed edge to node j, if and only if there is a directed edge from node j to node i. Therefore any state along the chain is reachable (and recoverable) from any other and previous state information is never lost. xviii, xx, 20, 65, 67–69, 72, 90, *see also* energy efficient computation & configuration graph

long domain A longer strand domain that binds irreversibly to complementary regions on template strands and can only be unbound from a template strand by toehold mediated strand displacement. xviii, xxi, xxiii, xxiv, 8, 9, 18, 20

mismatch displacement Any attempted toehold mediated strand displacement where the long domains of the one or more invading strands are not identical to the long domain of the evading strand. xviii, 9

mutex strand A special single copy signal strand that is required to perform any toehold mediated strand displacements. xviii, xxiv, 72, *see also* transaction

produced A signal molecule is said to be *produced* when it is added to the current signal multiset due to the application of a chemical reaction equation. xviii, xix, xxi–xxiv, 11, 14, 18, 20, 72

product A signal molecule that is produced due to the application of a chemical reaction equation. xviii, xix, xxi, xxii, 11, 13, 18, 20, 71

proper chemical reaction equation Either a reversible chemical reaction equation or an irreversible chemical reaction equation where the number of proper-products equals the number of proper-reactants. xviii, 18, *see also* proper-reactant & proper-product

proper Chemical Reaction Network (proper CRN) A CRN or tagged CRN is *proper* if each of its chemical reaction equations consumes the same number of reactants and that it produces as products. xviii, 18, 68, 90, *see also* CRN & tagged CRN

proper-product A signal molecule that is produced due to the application of a chemical reaction equation and is not consumed by the same reaction (*i.e.,* it is not a catalyst). xviii, xxii, 18, *see also* proper chemical reaction equation & proper-reactant

proper-reactant A signal molecule that is consumed due to the application of a chemical reaction equation and is not produced by the same reaction (*i.e.,* it is not a catalyst). xviii, xxii, 18, *see also* proper chemical reaction equation & proper-product

QSW construction A construction proposed by Qian et al. [98] to realize any tagged CRN by a DSD. xviii, 18, 20, 73

reactant A signal molecule that is consumed due to the application of a chemical reaction equation. xviii, xix, xxi, xxii, 11, 13, 18, 20, 71

reaction rate The relative speed that a chemical reaction equation can be applied within a given reaction volume. xviii, 12

reaction volume The container and medium where chemical reaction equations can be applied using the signal molecules currently present in the container. xviii, xx–xxiii, xxv, 11, 13, 17, 22, 64, 66, 68, 69, 79

required space of a tagged CRN The minimum size of the reaction volume for a tagged CRN to complete its intended sequence of reactions (computation). xviii, 17, 66, 90, *see also* space complexity of a tagged CRN computation

reversible chemical reaction equation A convenience of notation that denotes a reaction that can be applied in either direction (*i.e.,* the products and reactants can switch roles). Formally this equivalent to having two irreversible chemical reaction equations; one for each direction.. xviii, xix, xxii, xxiii, 11

saturated template strand A template strand where all long domains and all but one toehold domain is bound to other strands. xviii, 18

signal molecule An elementary (chemical) molecule that can be consumed and produced by the application of chemical reaction equations. xviii–xxiii, 11, 13, 18, 69–71

signal species A generic term to refer to a *type* of signal molecule or signal strand rather than a specific instance of that type. xviii, *see also* signal molecule & signal strand

signal strand A strand that is not bound (has paired bases with) any other strand. xviii, xx, xxi, xxiii, xxiv, 8, 9, 14, 18, 20

space complexity of a tagged CRN See space complexity of a tagged CRN computation. xviii, 18, 68

space complexity of a tagged CRN computation Given a trace ρ for a tagged CRN $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}}, T \rangle$, let S^* be the largest signal multiset of the sequence of multisets induced by ρ. The *space complexity* is defined to be $|S^*| + |T|$. xviii, xxiii, 17

spontaneous displacement Any event where a bound strand spontaneously breaks its bases pairs with the corresponding template strand to become a signal strand. xviii, 9

state of a CRN Defined by the current signal multiset of the CRN. xviii, 13, 70

strand domain A subsequence of a strand used in displacement reactions. xviii, xxi, xxiv, 8

tag A special signal assigned to chemical reaction equations that are implemented in a DSD to denote the required state of a transformer. In a reversible chemical reaction equation, the required tag for the reaction in the reverse direction is produced by the forward reaction, and *vice versa*. xviii, xxi, xxiii, xxiv, 16, 17, 82, 85

tagged chemical reaction equation Either a reversible chemical reaction equation or an irreversible chemical reaction equation which additionally requires that a tag specific to that reaction is present in the reaction volume before it can be applied. In a reversible chemical reaction equation, the required tag for the reaction in the reverse direction is produced by the forward reaction, and *vice versa*. xviii, xxiii, 16, 17, 65, 67

tagged Chemical Reaction Network (tagged CRN) Consists of an initial signal multiset, an initial tag multiset, and a set of tagged chemical reaction equations. Formally, we define a tagged CRN to be a tuple $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}}, T, T_0 \rangle$, where all members are defined the same as a

CRN and additionally T is the set of all tag species, and T_0 is the *initial tag multiset*, containing one or more tags for each reaction $R \in \mathcal{R}$. xviii, xxii, xxiii, 16, 17, 20, 24, 65–67, 69–73, 85, 90, *see also* CRN, tag & initial tag multiset

template strand A long strand that can bind one or more signal strands. xviii–xxi, xxiii, xxiv, 8, 9, 18

three way branch migration See toehold mediated strand displacement. xviii, xxi, xxiv, 9

toehold domain A short strand domain that binds reversibly to complementary regions on template strands. xviii, xix, xxi, xxiii, xxiv, 8, 18

toehold mediated strand displacement The toehold domain of an invading strand A binds (forms base-pairs) to the complementary toehold of the template strand B. Then in a random walk process (often referred to as three way branch migration), the bases of the long domain of A compete with those belonging to the identical long domain of the evading strand C to form base pairs with the complementary long domain of the template strand B that must be adjacent to the toehold domain. Once the long domain of A has bound to its complement of B, C remains bound to B by just its short toehold domain. The toehold bonds can break, thereby releasing signal C. (Of course A may detach from the template before C is released, in which case the displacement does not happen.). xviii–xxi, xxiv, 9, 18, 20, 71, 72

trace Given a CRN $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}} \rangle$, a *trace* for $\mathbf{C}$ is a sequence of reactions $\rho = R_1, R_2, \ldots, R_m$ from $\mathcal{R}$, where each $R_i = (I_i, P_i)$ such that ρ induces a corresponding sequence of multisets $S_0, S_1, \ldots, S_m$, with S_0 being the multiset of initial signal molecules in $\mathbf{C}$, and for all $1 \leq i \leq m$, we have both $I_i \subseteq S_{i-1}$ and $S_i = S_{i-1} - I_i + P_i$. xviii, xxiii, 13, 17, 82

transaction In the context of DSD, a *transaction* is a sequence of toehold mediated strand displacements, the first of which consumes a mutex strand and the last of which produces a mutex strand. xviii, 71, 72

transformer A collection of one or more strands, some forming a double stranded complex, which are used to implement a chemical reaction equation by consuming a set of signal strands and producing an alternate set of signal strands. xviii, xx, xxiii, xxv, 14, 16–18, 20, 65, 69, 70

unbound strand A strand where none its bases are paired to other bases on a template strand.. xviii, 8, 18, *see also* bound strand

universal toehold When all toehold domains in a DSD share a common sequence they are said to be universal. xviii, 8, 18, 20

waste One or more bound and unbound strands of a transformer used only once to effect a chemical reaction equation. The strands remain in the reaction volume in a closed system. xviii, 23, 66, 68

Notes on reading the text

By definition every PhD thesis should be unique and I hope this case is no exception. In what follows, I provide two distinct, self-contained and seemingly unrelated thesis parts. Both are motivated by problems related to nucleic acids, and my interest lies in the underlying combinatorial challenges they present. Briefly, in Part I, *Space and energy efficient molecular programming: energy barriers, chemical reaction networks, and DNA strand displacement systems*, I explore the combinatorial challenges associated with predicting and designing low-energy barrier folding pathways of one or more nucleic acid strands. In Part II, *Space efficient text indexes motivated by biological sequence alignment problems*, I design a number of compressed data structures to support efficient pattern matching queries for applications involving nucleic acid sequence alignment. While distinct, and self-contained, both thesis parts share common themes that motivated me to pursue both directions simultaneously. In particular, both parts address combinatorial problems related to nucleic acids and also explore the theme of space efficiency. Otherwise each thesis part stands on its own, with its own introduction and conclusion, and can be read independently. Each part has been written with a different research community in mind; however, a significant effort has been made to provide a sufficient background so that either part can be read by an interested computer science researcher.

On the use of *I*, *the candidate*, and *we*

In all technical chapters, I will avoid the use of the personal pronoun *I*. When specifically identifying myself, I will often use the term *the candidate*. Even when presenting material where I was the sole author, I will say *we* instead of *I*, as is tradition in scientific writing. However, when an opinion is being expressed, it can be assumed that the opinion is mine, but is not necessarily the opinion of my co-authors. This is often the case in chapter summaries and conclusions where I have had the benefit of considering the totality of the research considered in this thesis. When rewriting significant portions of published work, I have taken the opportunity to think of the implications beyond what was originally presented in isolation. I believe deeper insight is offered in this current form for both the motivations, and the contributions, of the research undertaken in this thesis.

Acknowledgements

I owe the greatest debt of gratitude to my research supervisor Professor Anne Condon. Anne not only supervised my PhD research, but also co-supervised my masters research. Most everything I know about being a scientist I have learned from her. I could not have found a better mentor. Anne's example continues to serve as an inspiration to become a better scientist, a better community citizen and a better person. Thank you Anne for your time, your mentorship and friendship, the academic freedom to pursue a number of research directions, and for teaching me how to be a scientist. It has been a wonderful experience to know you and work along side you.

Thank you to my committee members, Professor Will Evans and Professor Arvind Gupta. My research and the writing in this thesis have benefited from your feedback, suggestions and support. Thank you Will for many helpful discussions and in particular I would like to thank you for your suggestions and feedback that significantly improved the writing in the second part of my thesis. Thank you Arvind for all of your great advice over the years, both in terms of research and also more generally.

A special thanks to Dr. Jan Maňuch who has been not only my most active collaborator, but also a good friend. I have enjoyed the countless hours we have spent working together on problems and have learned a lot from you for which I am grateful. I have had many other wonderful collaborators over the years, and for that I am lucky. In particular, thank you to Professor Holger Hoos and Professor Alan Hu who have been a great source of advice at different stages of my research career. I also had the good fortune to work with a number of talented undergraduate students including Jay Zhang, Daniel Lai, John Cheu and Leigh-Anne Mathieson. Thanks to current and former members of the βeta lab at UBC, including Frank Hutter, Dave Tompkins, Hosna Jabbari, Mirela Andronescu, Bonnie Kirkpatrick, Baharak Rastegari, Murray Patterson, and Monir Hajiaghayi for their friendship, help and feedback. I would also like to thank the members of the DNA computing community who have offered insightful and constructive feedback for the research found in this thesis. In particular I would like thank Professor Erik Winfree, Dr. Lulu Qian, Dr. David Soloveichik and Dr. Dave Doty for helpful discussions that clarified my understanding of a number of complex topics.

I would also like to acknowledge the generous funding of my research from scholarships given by the National Science and Engineering Research Council of Canada (NSERC), the Michael Smith Foundation for Health Research (MSFHR) and also funding from UBC and my supervisor.

A special thanks to my family who have always encouraged me in everything I have done. Finally, thank you Meagan for your unwavering support and the happiness you give me. Completing this thesis meant many evenings and weekends spent in a lab away from you. The time put into this thesis was not entirely mine to spend and so this work is as much yours as it is mine.

Dedication

For Meagan.

Part I

Space and energy efficient molecular programming: energy barriers, chemical reaction networks, and DNA strand displacement systems

Chapter 1

Introduction

In this chapter we motivate work in Part I of our thesis, on nucleic acid folding pathways. Our research in this area has two primary motivations. First, the work was begun with the aim to better understand and computationally predict folding pathways exhibited in biological systems. During the course of that initial research, we realized the computational hardness of solving the prediction problem. This suggested to us that folding pathways may be a mechanism for performing non-trivial computation. Indeed, designed folding pathways of multiple interacting nucleic acid strands were already being used to perform simple computation. This potential was the impetus for our second motivation: understanding the computational limits of molecular programs that leverage folding pathways. Such pathways also have the potential to perform logically reversible, and thus energy efficient, computation. A better understanding of these designed pathways can shed light on the complexity of predicting folding pathways involving multiple strands. In what follows, we discuss each motivation and its related work, summarize the contributions of this thesis part, and give an overview of the ensuing chapters that detail our technical contributions. This chapter introduces concepts as needed to understand our motivations and gives a comprehensive overview of the models we study in this thesis. Additional concepts and definitions are introduced as needed in later chapters.

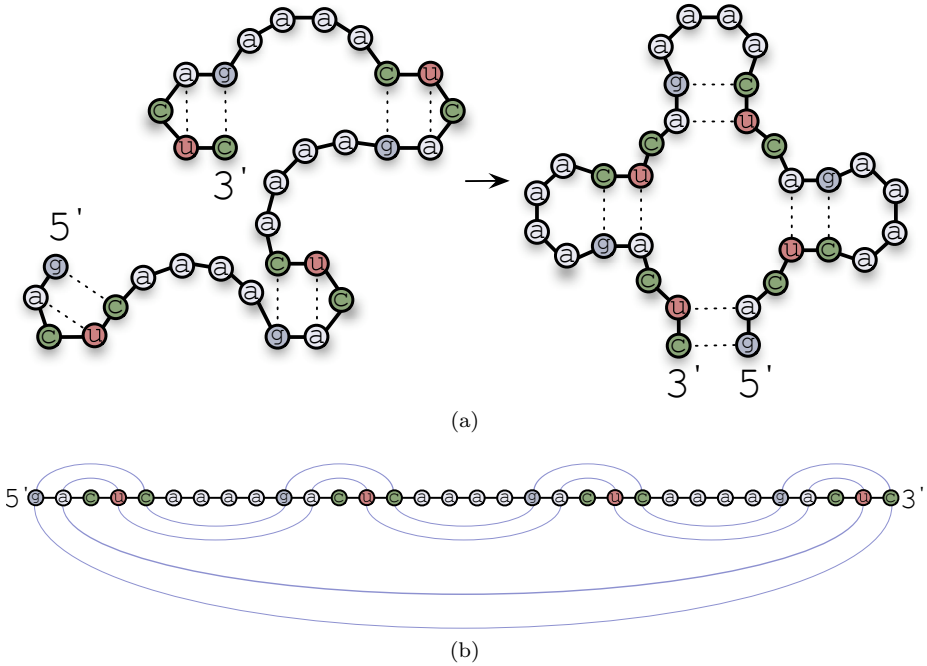
1.1 Nucleic acid folding pathways

RNA molecules play vital roles in the cell by virtue of the information encoded into their nucleotide sequence and the structures the molecules form. The primary structure, or nucleotide sequence, can be thought of as a string over the alphabet $\Sigma_{RNA} = \{A, C, G, U\}$. The tertiary, or 3-dimensional, structure of an RNA is determined in large part by the bonds formed between pairs of complementary nucleotides within its sequence, such as the Watson-Crick base pairs: A pairs to U and C pairs to G[1]. These bonds constitute the secondary structure of the molecule. An example of two alternate secondary structures for the same nucleotide sequence is given in Figure 1.1(a). Shown in Figure 1.1(b) is a common representation for secondary structures called the arc diagram representation, where each arc denotes a base pair. Prediction of RNA secondary structure is

[1] Formal definitions are given in the following chapter.

crucial to understand their myriad[2] biological functions. Throughout, we will use the term structure to mean secondary structure.

Given their propensity to alter their shape over time under changing environmental conditions, an RNA molecule will fold through a series of structures called a folding pathway [4, 42, 55, 104, 115, 145]. Thus knowledge of folding pathways between pairs of alternative RNA structures is very valuable for inferring RNA function in such environments, and is also valuable for predicting RNA structure, *e.g.,* in light of co-transcriptional folding [20, 42, 108, 117, 133]. As illustrated in Figure 1.2, each structure differs from its predecessor by a single base pair (or equivalently by a single arc in the arc diagram representation).



(a)

(b)

Figure 1.1: (a) An initial secondary structure (left) and a final secondary structure (right) for a given RNA strand. (b) A corresponding arc diagram. The arcs on top of the nucleotide sequence denote the base pairs in the initial secondary structure while the arcs below the nucleotide sequence denote the base pairs of the final secondary structure.

Much focus to date has been on pathways of pseudoknot-free secondary structures—structures in which no base pairs cross in the arc diagram representation[3]. Since folding is a thermodynamically-driven probabilistic process,

[2]While the number of biological functions of RNA in a cell is certainly finite, new functions of this versatile molecule continue to be elucidated.

[3]Complex pseudoknots are rare in nature and therefore algorithmic approaches usually assume their absence in any reasonable solution space.

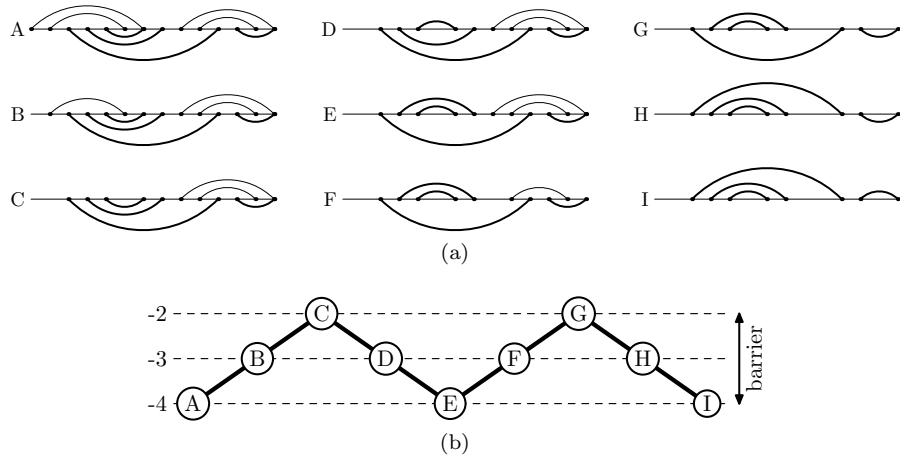


Figure 1.2: (a) A possible folding pathway is shown for an initial structure A transitioning through intermediate structures $(B, C, \ldots)$ until the final structure I is reached. For a particular position in the pathway, the top of the arc diagram denotes the current base pairs of the structure and the bottom of the arc diagram denotes the base pairs still to be added to reach the final structure. Each structure along the pathway differs from its neighbours by one arc. (b) The corresponding energy plot. The barrier in this example is two.

folding pathways tend to avoid high-energy structures[4]. As a result, many methods for predicting folding pathways or entire energy landscapes[5]—particularly course-grained methods designed to work for large structures which do not attempt to model the complete energy landscape—are guided by calculations of the *energy barrier* [42, 125]. Intuitively, this is the highest energy difference between the initial structure and the other structures along a pathway. For example, the folding pathway in Figure 1.2 has an energy barrier of 2 under the *simple energy model* where each base-pair contributes -1 to the overall energy of a structure. Contrast this with a naive folding pathway that first removes all base-pairs in the initial structure before adding all base-pairs in the final structure, resulting in an energy barrier of 4.

There is a rich literature on the problem of predicting folding pathways and energy landscapes, both in theory and in practice; see the recent work of Geis at al. [42], Tang et al. [125] and the references therein. We focus here on algorithms for energy barrier calculation which are an important component of many approaches to estimation of entire energy landscapes. Such methods have been proposed, for example, by Morgan and Higgs [83], Wolfinger [143], Flamm et al. [36–38], Geis et al. [42] and Dotu et al. [29].

[4]These structures are energetically unfavourable. In simple energy models, high-energy structures have fewer base-pairs than alternative structures of the same molecule.

[5]A discrete interpretation of an energy landscape for an RNA molecule is the set of all pathways between all structures the molecule can form.

Several versions of the energy barrier problem have been studied, which are distinguished by properties of the intermediate structures. Morgan and Higgs focus on *direct* folding pathways from structure $\mathcal{A}$ to structure $\mathcal{B}$ in which intermediate structures contain only arcs in $\mathcal{A} \cup \mathcal{B}$ and such that the total pathway length is $|\mathcal{A} \triangle \mathcal{B}|$ (the size of the symmetric difference). In such pathways, each arc from the initial structure not also in the final structure is removed exactly once and each arc from the final structure not also in the initial structure is added exactly once along the pathway. The example in Figure 1.2 is a direct pathway. A larger class of pathways is obtained by allowing the length of the pathway to exceed $|\mathcal{A} \triangle \mathcal{B}|$. We call such pathways *direct-with-repeats* pathways since an arc from $\mathcal{A}$ or $\mathcal{B}$ may be added or removed multiple times along the pathway. An even more general class of pathways allows intermediate structures to contain "temporary" arcs which are neither in $\mathcal{A}$ nor in $\mathcal{B}$. These temporary arcs would denote base pairs that are not found in either the original nor the final structure. Morgan and Higgs call such pathways *indirect*. Thus, direct pathways are a subclass of direct-with-repeats pathways, which in turn are a subclass of indirect pathways.

Morgan and Higgs assume the simple energy model in which each base pair contributes -1 to the total free energy. Using a randomized greedy approach, they construct several low-barrier direct pathways and take the minimum energy barrier of these as their estimate. (They also construct indirect pathways using a "single link clustering" method.) Wolfinger et al. use a barrier tree to represent the folding landscape; identifying nodes in the tree (which are called saddle points) is analogous to calculating energy barriers. Flamm et al.'s method [37] for approximating energy barriers explores direct pathways by performing a breadth-first search, maintaining the best m candidate solutions at each step. As m becomes large, the search does become exhaustive, yielding an exact solution, however exponential runtime and memory are required. The program `barriers` [38] is capable of computing exact direct and indirect pathways, provided a complete sample of low energy states separating the two structures is provided. However, this approach is also exponential in runtime and space and thus impractical for medium (100-500 nucleotides) or large (> 500 nucleotides) problem instances.

For their Kinwalker folding pathway predictor, Geis et al. [42] describe a heuristic which explores the space of possible direct pathways in a more sophisticated manner than does the Morgan-Higgs heuristic, incorporating a parameter look-ahead technique to avoid excessive runtimes. While their method uses the Turner energy model [74, 79, 134][6] to evaluate energy barriers, it relies on simple addition and removal of base pairs (and thus the simple energy model) while generating putative low-barrier pathways.

It is important to note that, in general, determining energy barriers is not restricted to alternate structures of a single RNA (or DNA) strand (molecule). The idea of computing an energy barrier can be generalized to consider alternate

[6] The Turner energy model is a more realistic energy model parameterized by experimental data and is the standard model used for algorithmic prediction of nucleic acid secondary structure.

structures of sets of multiple interacting strands. Inter-molecular base-pairs can form between strands in addition to intra-molecular base-pairs. As we discuss in the next section, the ability for strands to predictably interact can be exploited to design and perform molecular computation. Given the importance of understanding interactions of multiple strands in problem domains such as DNA computing, there are already efforts to provide practical and effective probabilistic simulation tools—see the work of Schaeffer [112] and references therein.

In summary, all current methods are either heuristic in nature and thus are not guaranteed to find the exact energy barrier between two structures, even for a single molecule, or are exponential in both runtime and space, precluding their use on even medium sized problem instances. Thus there is strong motivation for finding a *fast* method which can *exactly* compute the energy barrier between two structures. Indeed, the Geis et al. method can estimate energy barriers for structures of long sequences (1,500nt or more) but the authors note that "as the performance of Kinwalker crucially depends on approximating saddle heights, further improvements to the Morgan-Higgs heuristic as well as alternative approaches will be investigated".

1.1.1 The *simple* energy model

We now formally define the simple energy model. To do so, we must define the concept of a strand *complex*. A single strand, that has no base pairs to another strand, is a complex. A group of more than one strands forms a complex if (i) no strand in the group has a base pair with a strand outside of the group and (ii) given any partition of the group into two subgroups, there is at least one base pair from a strand in one subgroup to the other. Formally, they form a connected component in the circle arc diagram representation. For instance, Figure 1.5(a) shows a circle arc diagram for three strands where edges denote base pairs; strands C and B form a complex and strand A forms its own complex. In Figure 1.5(b) there is only one complex formed by strands A, B and C.

Each Watson-Crick base pair (*i.e.,* `A-U` and `C-G` for RNA and `A-T` and `C-G` for DNA) contributes -1 to the overall energy. Therefore, the energy of a structure for a single strand can be formally defined as:

$$-\#\texttt{basepairs} \tag{1.1}$$

For instance, both structures depicted in Figure 1.1(a) are composed of a single complex, contain eight base pairs, and therefore have energy -8. Base pairs are also counted in the multiple strand case. Additionally, there is an entropic penalty constant K_{assoc}, $K_{\text{assoc}} > 1$, for each strand association that results in fewer strand complexes. A strand association event occurs when the first base pair is formed between two complexes that were not previously associated; thus, the energy will instantaneously change by $K_{\text{assoc}} - 1$. The

simple energy model for multiple interacting strands can be formally defined as:

$$(\#\texttt{strands} - \#\texttt{complexes})K_{\text{assoc}} - \#\texttt{basepairs} \tag{1.2}$$

The example of Figure 1.5(a) would have energy $(3-2)K_{\text{assoc}}-3 = K_{\text{assoc}}-3$ whereas the example of Figure 1.5(b) would have energy $(3-1)K_{\text{assoc}} - 4 = 2K_{\text{assoc}} - 4$ as it contains one fewer complex and one additional base pair.

As with early studies of RNA structure prediction algorithms, in this thesis we will study folding pathways by adopting the simple energy model. The reasoning for this choice is two-fold. First, this model is significantly simpler and remains sufficient to understand the complexity of the underlying combinatorial problem. If the problem is hard in the simple energy model, it provides evidence that it is hard for more complex models. Second, if effective algorithms are developed in the simple energy model, then it is possible that they could be adapted for more complex energy models. This was the case for the RNA structure prediction problem that was first studied with the simple energy model [89] and later improved to use the Turner energy model [79].

1.2 Molecular programming

The area of molecular programming enjoys active research from both theoreticians and experimentalists due in part to its promise of embedded logical computation that can naturally interface with biological systems. For instance, *if* a condition is detected in a cell, *then* a certain therapeutic agent can be released. A widely studied and experimentally practical model of computation in molecular programming entails so-called DNA[7] strand displacement systems (DSD). DSDs leverage the fact that substrings of DNA strands will hybridize to their perfect complements and can also displace other bound strands sharing the same substring. By a careful, non-trivial design of strands one can realize a complex, yet deterministic computation. DSDs have been experimentally implemented and verified to simulate logic circuits [21, 116], neural networks [99], and DNA walkers [119], among numerous other applications [51, 75, 92, 97, 122, 136, 151]. They have also been shown capable, in principle, of energy-efficient Turing-universal computation [64, 98]. An underlying property of DSDs is that intended sequences of strand displacements form low-energy barrier pathways, while unintended sequences must over come a high-energy barrier. This connection to folding pathways was our initial motivation for studying DSDs in this thesis. However, as discussed below, DSDs consider strands at an abstract domain level, and not at the sequence level. Thus, any formal conclusions that we draw about underlying folding pathways of DSDs will necessarily consider a corresponding sequence design, and will be in terms of the multiple interacting strand folding pathway model.

[7]DNA is the molecule of choice for molecular computing because of its stability in comparison to RNA. Concepts like secondary structure still apply as DNA forms base pairs just as RNA does, although using a different set of nucleotides ($\{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$).

1.2.1 DNA strand displacement systems (DSD)

A *DNA Strand Displacement system (DSD)* consists of *signal strands* and *double stranded complexes* consisting of one or more *bound strands* to a long *template strand*. Strands that are not bound are said to be *unbound strands*. DNA strands are oriented and have a 5' end and a 3' end. A strand can only bind to another strand in opposite orientation. Consider the example DSD in Figure 1.3. There are two signal strands A and B, and one double stranded complex. The 3' ends of the strands are depicted in the example with arrows ($\rightharpoonup$). The bound strands C and D have opposite orientation to the template strand E.

Figure 1.3: A DNA strand displacement system consisting of two signal strands, A and B, and one double stranded complex consisting of two bound strands C and D and a template strand E. Long domains of the template strand are shown in **red** while long domains of signal strands and bound strands are shown in **gray**. Universal toehold domains are shown in **black**.

Strands in the system are composed of two types of *strand domains*: short *toehold domains*, and *long domains*. All DSDs that we study make use of *universal toeholds* meaning that all toehold domains share a common sequence. Distinct long domains are assumed to have a distinct sequence design. In the example DSD of Figure 1.3, universal toehold domains are shown in black and long domains of bound and signal strands are shown in gray. The long domains of the template strand E are shown in red. The template strands have complementary toehold domains to those on signal and bound strands.

Toehold mediated strand displacement

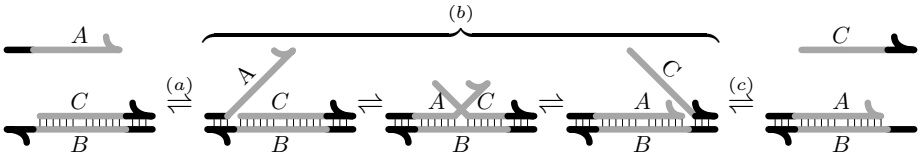


Figure 1.4: Strand displacement. (a) Toehold (black subsequence) of signal strand A binds with its unpaired complement on the template strand B. (b) The single long domain (gray subsequence) of A competes via a random walk process with the single long domain of strand C to bind with the complementary long domain of B until all bases of A are bound to B. (c) Toehold of C detaches from B, at which point it has been displaced and becomes a signal. The process is reversible and signal strand C could next displace the bound strand A.

Toehold domains bind reversibly, and long domains irreversibly, to complementary regions on template strands. The fundamental operation in a DSD is *toehold mediated strand displacement*, whereby a toehold domain of a signal strand, called the *invading strand*, binds to an unbound complementary toehold domain of a template strand and, if the adjacent long domain is complementary, it can displace a currently bound signal strand, called the *evading strand*, of the same length.

We illustrate a simple, reversible version of toehold mediated strand displacement in Figure 1.4. First, the toehold of invading signal strand A binds (forms base-pairs) to the complementary toehold of the template strand B. Then in a random walk process (often referred to as *branch migration*), the bases of the long domain of A compete with those belonging to the identical long domain of the evading strand C to form base pairs with the complementary long domain of the template strand B. Once the long domain of A has bound to its complement of B, C remains bound to B by just its short toehold domain. The toehold bonds can break, thereby releasing signal C. (Of course A may detach from the template before C is released, in which case the displacement does not happen.) The displacement is reversible because signal C can bind to the template strand B to displace strand A via the same principles.

Folding pathway of a strand displacement

Figure 1.5 illustrates the same displacement in terms of the corresponding folding pathway for a hypothetical sequence design where toehold domains have a length of one base and long domains have a length of two bases. After the first (and only) toehold base pair is formed between strand A and B, relative to that energy, branch migration of the long domains of A and C occur within energy barrier 1. This is because a base pair between A and B can be added immediately after a base pair between C and B is removed.

Illegal strand displacement

The example in Figure 1.4 and Figure 1.5 illustrates a *legal displacement*. Now let us consider how an evading strand can be displaced (*i.e.,* produced as a signal) from a template other than by a legal displacement. First, it is possible that one or more invading strands with a different long domain are used for displacement. We call this a *mismatch displacement*. Second, it is possible the invading strand does have an identical long domain, but toehold base pairs between the invading strand and template strand are not formed prior to branch migration. We call this *blunt-end displacement*. Finally, it is possible that the evading strand simply breaks all base pairs with the template strand and disassociates. We call this a *spontaneous displacement*. The occurrence of any one of the three types of *illegal displacements* would result in a folding pathway with a higher energy barrier. Consistent with Soloveichik et al. [122], we assume throughout that only legal displacements can occur and that sequences of domains can be designed to be sufficiently different that a strand with domain δ is very unlikely

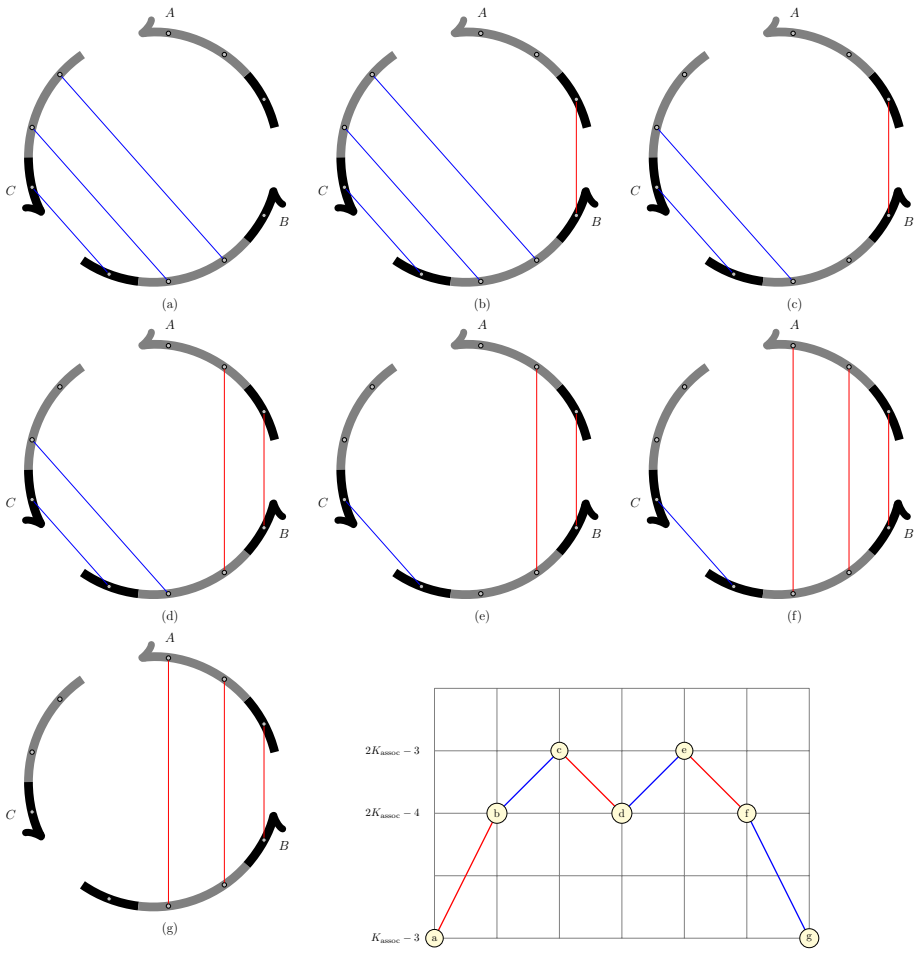


Figure 1.5: A corresponding folding pathway is shown for the displacement example of Figure 1.4 using a simple sequence design where toehold domains have one base and long domains have two bases. The displacement of strand C by strand A is shown in seven steps, from (a) to (g). Base pairs are shown as edges between strands. The energy changes between each structure, assuming $K_{\text{assoc}} = 2$, are shown in the bottom right. Since toehold domains have one base, then the energy barrier of the underlying folding pathway, relative to (a), is K_{assoc}. If the toehold domains had length $L_T > 1$ then the energy barrier, relative to (a), would be $K_{\text{assoc}} - 1$.

to displace a strand with domain δ'.[8]

1.2.2 Chemical reaction networks (CRN)

Just as a DSD abstracts sequence level details of a folding pathway using the concept of domains, *Chemical Reaction Networks (CRNs)* abstract details about displacements. CRNs provide a concise language for writing molecular programs and affords us the opportunity to express complex ideas more succinctly in this thesis.

Chemical reactions and signal molecules

A *chemical reaction equation* details a process whereby certain molecule types can be *consumed*—the *reactants*—and others *produced*—the *products*—within some *reaction volume*. A reaction may also require the presence of *catalyst* molecules of certain types. We refer to all three categories, generically, as *signal molecules*. For example, the reaction $A + B \xrightarrow{C} D$ consumes a signal of type A and a signal of type B and produces a signal of type D in the presence of the catalyst[9] signal C. This is an example of an *irreversible chemical reaction equation*; however, $A + B \overset{C}{\rightleftharpoons} D$ is an example of a *reversible chemical reaction equation* meaning that both a signal of type A and of type B can also be produced by consuming a signal of type D in the presence of the catalyst signal C. A CRN is a set of chemical reactions, in addition to a multiset of signals present within the reaction volume, prior to any reaction occurring, called the *initial signal multiset*. The *current signal multiset* is the current composition of signals of a given CRN within a reaction volume. In this work, we consider a reaction volume to be a *closed system*, meaning that signals cannot be added to the current signal multiset unless they are produced by a reaction, and signals cannot be removed from the current signal multiset unless they are consumed by a reaction.

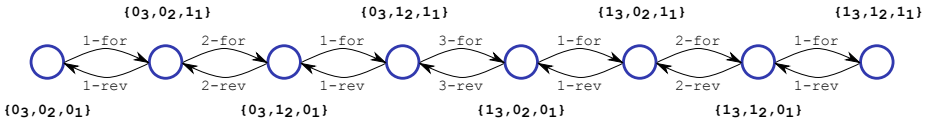
Example 1.2.1. Let us consider a concrete example of a 3-bit standard binary counter that should begin at count 000, advance to 001, and so on, until reaching the count 111. In our molecular program, we let signal 0_i and signal 1_i denote that bit i has value 0 and 1, respectively, for $1 \leq i \leq 3$. Thus, our 3-bit counter will have the following initial signal multiset: $\{0_3, 0_2, 0_1\}$. Figure 1.6(a) gives three chemical reaction equations for exchanging signals and thus changing the state, or current signal multiset, of the counter. Figure 1.6(b) represents all

[8] This is a reasonable assumption one can make and can be shown formally using results from coding theory. Schulman and Zukerman [114] show how to construct a set of $2^{\Theta(n)}$ domains (*i.e.,* binary strings in their code) of equal length $\Theta(n)$ such that the energy barrier (Levenshtein distance) between any pair of domains is at least cn, for any given constant c.

[9] Some reactions require the presence of one or more signals, called *catalysts*, which they do not consume. Note how we represent catalysts in our reaction equations. These are not to be confused with rate constants which do not significantly factor into our research. Catalysts do play a significant role our thesis research and this representation was chosen for its succinctness.

$$
\begin{array}{lrcl}
(1) & 0_1 & \rightleftharpoons & 1_1 \\
(2) & 0_2 + 1_1 & \rightleftharpoons & 1_2 + 0_1 \\
(3) & 0_3 + 1_2 + 1_1 & \rightleftharpoons & 1_3 + 0_2 + 0_1
\end{array}
$$

(a)



(b)

Figure 1.6: (a) Chemical reaction equations for a 3-bit standard binary counter. (b) The configuration graph of the computation performed by the 3-bit standard binary counter forms a chain and is logically reversible. The nodes represent the state of the computation and the edges are directed between states reachable by a single reaction.

reachable states of the counter as nodes and has edges between states that are reachable within one reaction step.

Chemical reaction rates

When reasoning about time complexity of various chemical reaction networks, we will use the well known stochastic chemical kinetics model [43]. This model, based on a continuous time Markov process, permits us to reason about the probability and the expected time to completion of sequences, and of individual, chemical reactions within a well mixed reaction volume. In general, each reaction has an associated *reaction rate* denoting the relative speed of the reaction within some defined reaction volume. The rate of a reaction is dependent on (i) the order of the reaction, (ii) the size of reaction volume, and (iii) the reaction rate constant. In all examples of this thesis, we will assume a reaction volume of size v and assume a uniform reaction rate constant, $k = 1$. As all reactions share the same reaction rate constant, we omit them when reasoning about the expected reactions times. The order of a reaction is the number of required reactants. For instance, $R_1 \rightarrow \ldots$ is a *unimolecular* reaction, $R_1 + R_2 \rightarrow \ldots$ is a *bimolecular* reaction, $R_1 + R_2 + R_3 \rightarrow \ldots$ is a *trimolecular* reaction, and so on. In a volume of size v and assuming a uniform rate constant $k = 1$, the propensity of a unimolecular reaction $R_1 \rightarrow \ldots$, is $\frac{|R_1|}{v^0} = |R_1|$, where $|R_1|$ denotes the number of copies of signal R_1 in the current state. The propensity of a bimolecular reaction, $R_1 + R_2 \rightarrow \ldots$ is $\frac{|R_1||R_2|}{v^1}$, assuming $R_1 \neq R_2$, and is $\frac{2|R_1|-1}{v^1}$ otherwise. In general, the propensity of an i^{th} order reaction, assuming i distinct reactants, is $\frac{|R_1||R_2|\ldots|R_i|}{v^{i-1}}$. The expected time until the next reaction is an exponential random variable with a rate, r, equal to the sum of the propensities of reactions that can occur. The probability of a particular

reaction occurring next is equal to its propensity divided by r. Intuitively, for all reactions that can occur in a particular state (*i.e.,* all reactants are present in sufficient quantity), higher order reactions are less likely than lower order reactions. In this thesis, many of the CRNs we propose will initially contain higher order reactions. However, when implemented as DSDs, all reactions will be bimolecular reactions between two distinct species, each of a single copy. A further property of the CRNs we propose in this thesis is that in any given state, at most two reactions are possible. Thus, in the DSD realizations of our CRNs (see Section 1.2.5), the propensity of a reaction that can occur is $\frac{1}{v}$, and the expected time for the reaction to occur is $O(v)$.

CRNs as a means for computation

We will always define the CRNs we study in this work using the chemical reaction notation already introduced. However, it is helpful to have a formal definition of a CRN when proving certain results. This eliminates ambiguity that may arise, for example, when reasoning about reversible reactions and catalysts. In addition, a formal definition will permit us to define what it means to perform computation with a CRN in a reaction volume that is a closed system. We define a Chemical Reaction Network (CRN) to be a tuple $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}} \rangle$, where

- S is the set of all signal types (or species) of signal molecules used in any reaction.
- $\mathcal{R}$ is a set of chemical reaction equations, where each $R \in \mathcal{R}$ is an ordered pair of multisets of signal molecules. Intuitively, a reaction equation $R = (I, P)$ consumes the signal molecules in I as the input reactants and produces the signal molecules in P as products. Note that it is only the signal molecules in $I - P$ that are actually consumed. The others act as catalysts for the reaction. Our formalism is directional to allow modeling non-reversible reactions; a reversible chemical reaction is modeled as two separate elements of $\mathcal{R}$, *i.e.,* (I, P) and (P, I).
- S_0 is the initial signal multiset where $s \in S_0 \rightarrow s \in S$.
- $s_{\text{end}} \in S$ is a signal molecule denoting the end of computation.[10]

The *state of a CRN* is defined by its current signal multiset. We formalize computations in $\mathbf{C}$ in the natural manner: Let ρ be a sequence of reactions $R_1, R_2, \ldots, R_m$ from $\mathcal{R}$, where each $R_i = (I_i, P_i)$. We define ρ to be a *trace* of $\mathbf{C}$ if ρ induces a corresponding sequence of multisets $S_0, S_1, \ldots, S_m$, with S_0 being the multiset of initial signal molecules in $\mathbf{C}$, and for all $1 \leq i \leq m$, we have both $I_i \subseteq S_{i-1}$ and $S_i = S_{i-1} - I_i + P_i$. (We use "$-$" and "$+$" to denote multiset subtraction and union.) If ρ is a trace for a completed computation,

[10] A computation may have multiple final states. To model this situation, we can let s_{end} be produced in all final reactions, in addition to any other signal molecules that may indicate the result of the computation.

then $s_{\text{end}} \in S_m$ and $s_{\text{end}} \notin S_n$ for $n \neq m$. Note that throughout this thesis, we consider the computation to halt when the s_{end} signal is first produced. The traces we study for completed computations will reflect this fact.

Example 1.2.1 (continued)**.** Let us describe the 3-bit standard binary counter formally. The set of signal types is $S = \{0_1, 0_2, 0_3, 1_1, 1_2, 1_3, s_{\text{end}}\}$, where s_{end} denotes the end of computation. The initial signal multiset is $S_0 = \{0_3, 0_2, 0_1\}$. Finally, we have the following set of chemical reaction equations

$$\begin{aligned}\{R_{\text{1-for}} &= (\{0_1\}, \{1_1\}),\\ R_{\text{1-rev}} &= (\{1_1\}, \{0_1\}),\\ R_{\text{2-for}} &= (\{0_1, 1_2\}, \{1_1, 0_2\}),\\ R_{\text{2-rev}} &= (\{1_1, 0_2\}, \{0_1, 1_2\}),\\ R_{\text{3-for}} &= (\{0_1, 1_2, 1_3\}, \{1_1, 0_2, 0_3\}),\\ R_{\text{3-rev}} &= (\{1_1, 0_2, 0_3\}, \{0_1, 1_2, 1_3\}),\\ R_{\text{end}} &= (\{1_1, 1_2, 1_3\}, \{s_{\text{end}}\})\}\,.\end{aligned}$$

These reactions, with the exception of the last one, formally define the reactions shown in Figure 1.6(a). The shortest trace producing s_{end} is the sequence of reactions

$$R_{\text{1-for}}, R_{\text{2-for}}, R_{\text{1-for}}, R_{\text{3-for}}, R_{\text{1-for}}, R_{\text{2-for}}, R_{\text{1-for}}, R_{\text{end}}\,,$$

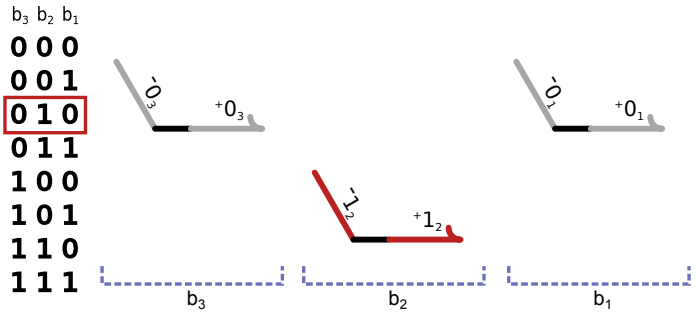
which induces the following sequence of multisets:

$$\begin{gathered}\{0_3, 0_2, 0_1\}, \{0_3, 0_2, 1_1\}, \{0_3, 1_2, 0_1\}, \{0_3, 1_2, 1_1\},\\ \{1_3, 0_2, 0_1\}, \{1_3, 0_2, 1_1\}, \{1_3, 1_2, 0_1\}, \{1_3, 1_2, 1_1\}, \{s_{\text{end}}\}\,.\end{gathered}$$

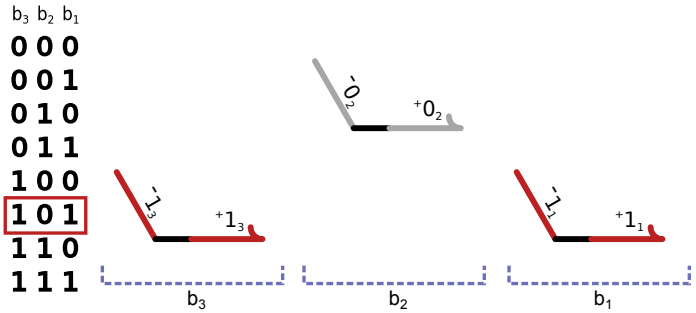
1.2.3 Tagged chemical reaction networks (tagged CRN)

As CRNs are an abstract description for molecular programs, we must consider how they can be realized by physical systems such as DSDs. Continuing with the 3-bit standard binary counter example, we can represent each 0_i and 1_i with a unique strand, for $1 \leq i \leq 3$. Figure 1.7 shows two states of the counter and the composition of signal strands representing those states. Thus, state representation is easily achieved, but how does one transition between states? For instance, how can reaction (1) of Figure 1.6(a) be implemented?

Unfortunately, we do not know how to change the signal strand 0_1 *directly* into the signal strand 1_1. However, we do know how to achieve the same result, *indirectly*. Figure 1.8 shows a strand displacement process—based on toehold mediated strand displacement as discussed in Section 1.2.1—implementing the reaction $0_1 \rightleftharpoons 1_1$ based on a construction proposed by Qian et al. [98]. From top to bottom, the 0_1 signal strand interacts with a *transformer* to first become consumed — sequestered on a double stranded complex — and ultimately the 1_1 signal strand is produced — released from a double stranded complex. The strands contained within a shaded box are the signal strands, while everything else forms the transformer for this reaction. The important point is that the



(a) The current signal strands denote that the counter has value 010.



(b) The current signal strands denote that the counter has value 101.

Figure 1.7: Representing the state of the CRN for a 3-bit standard binary counter can be achieved by the presence and absence of certain signal strands for each bit position. Long domains for bits representing a 1 value are coloured in **red** while those representing a 0 value are coloured in **grey**. Universal toehold domains are coloured **black**.

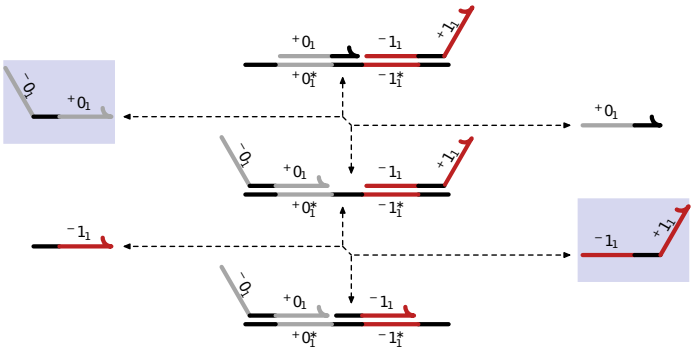


Figure 1.8: A strand displacement implementation of the reaction $0_1 \rightleftharpoons 1_1$ as proposed by Qian et al. [98]. From top to bottom, the input signal strand 0_1 (shown in a shaded box on the left) is consumed by the transformer (middle) which produces the signal strand 1_1 (shown in a shaded box on the right). Additional unbound strands are used in the process and are considered part of the transformer. The transformer can be applied next in the opposition direction (from bottom to top) to consume signal 1_1 and produce signal 0_1. In this and later figures, the Watson-Crick complement of a domain x is denoted by x^*.

transformer is not in the same state after producing signal 1_1 as it was prior to consuming signal 0_1. The transformer is no longer in a state that can consume a 0_1 signal and produce a 1_1 signal. It is however in a state that can perform reaction (1) in reverse (from bottom to top in Figure 1.8). Thus, while the same transformer can be used to perform both the forward and reverse of a reaction, it must strictly alternate between these directions.

Tags and tagged chemical reaction equations

To capture this notion of transformer orientation at the level of a chemical reaction network, we can *tag* each side of a reaction to represent the transformer and its required orientation that is necessary to perform a reaction in the respective direction. In the case of reversible reactions, when considered as two separate reactions, the forward tag of one will be the reverse tag of the other. We call these *tagged chemical reaction equations*. A *tagged Chemical Reaction Network (tagged CRN)* consists of an initial signal multiset, an initial tag multiset, and a set of tagged chemical reaction equations. Formally, we define a tagged CRN to be a tuple $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}}, T, T_0 \rangle$, where all members are defined the same as a CRN and additionally T is the set of all tag species, and T_0 is the initial tag multiset, containing one or more tags for each tagged chemical reaction equation $R \in \mathcal{R}$.

Space complexity of a tagged CRN

This simple concept of tags allows us to account for the required number of transformers and the minimum size of the reaction volume required to complete a computation. Given a trace ρ for a tagged CRN $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}}, T, T_0 \rangle$, let S^* be the largest signal multiset of the sequence of multisets induced by ρ. We define the *space complexity of a tagged CRN computation* with trace ρ of tagged CRN $\mathbf{C}$ to be $|S^*| + |T_0|$. Note that for every reaction, exactly one tag is consumed and one is produced, thus the number of tags present in any reachable state is equal to $|T_0|$. Intuitively, this corresponds to the minimum size of the reaction volume of a closed system to fit all molecules necessary to complete the computation specified by the trace ρ. We will often refer to this quantity as the *required space of a tagged CRN*.

$$
\begin{array}{lrcl}
(1) & \mathrm{T}^{\mathrm{f}}_{\mathbf{1}} + 0_1 & \rightleftharpoons & \mathrm{T}^{\mathrm{r}}_{\mathbf{1}} + 1_1 \\
(2) & \mathrm{T}^{\mathrm{f}}_{\mathbf{2}} + 0_2 + 1_1 & \rightleftharpoons & \mathrm{T}^{\mathrm{r}}_{\mathbf{2}} + 1_2 + 0_1 \\
(3) & \mathrm{T}^{\mathrm{f}}_{\mathbf{3}} + 0_3 + 1_2 + 1_1 & \rightleftharpoons & \mathrm{T}^{\mathrm{r}}_{\mathbf{3}} + 1_3 + 0_2 + 0_1
\end{array}
$$

Figure 1.9: Tagged chemical reaction equations for a 3-bit standard binary counter.

Example 1.2.1 (continued)**.** We can augment our 3-bit standard binary counter chemical reaction equations from Figure 1.6(a) with tags resulting in the tagged chemical reaction equations shown in Figure 1.9. Formally, we have the following set of tagged chemical reaction equations

$$
\begin{aligned}
\{R_{\text{1-for}} &= (\{T_1^f, 0_1\}, \{T_1^b, 1_1\}), \\
R_{\text{1-rev}} &= (\{T_1^b, 1_1\}, \{T_1^f, 0_1\}), \\
R_{\text{2-for}} &= (\{T_2^f, 0_1, 1_2\}, \{T_2^b, 1_1, 0_2\}), \\
R_{\text{2-rev}} &= (\{T_2^b, 1_1, 0_2\}, \{T_2^f, 0_1, 1_2\}), \\
R_{\text{3-for}} &= (\{T_3^f, 0_1, 1_2, 1_3\}, \{T_3^b, 1_1, 0_2, 0_3\}), \\
R_{\text{3-rev}} &= (\{T_3^b, 1_1, 0_2, 0_3\}, \{T_3^f, 0_1, 1_2, 1_3\}), \\
R_{\text{end}} &= (\{T_{\text{end}}, 1_1, 1_2, 1_3\}, \{T'_{\text{end}}, s_{\text{end}}\})\}\,.
\end{aligned}
$$

If we consider the sequence of reactions illustrated in Figure 1.6(b) to advance from count 000 to 111, then the initial signal multiset is still $\{0_3, 0_2, 0_1\}$ and the initial tag multiset required for the computation to reach the count 111 and finally produce s_{end} is $\{T_1^f, T_1^f, T_1^f, T_1^f, T_2^f, T_2^f, T_3^f, T_{\text{end}}\}$. Therefore, the required space or space complexity for this tagged CRN is eleven molecules as each signal multiset during the computation has the same size as the initial signal multiset. In general, we will reason about the space complexity of a tagged CRN asymptotically.

1.2.4 Proper chemical reaction networks (proper CRN)

We define one additional restricted class of CRNs to help simplify our space complexity analysis throughout this thesis. Given a (possibly tagged) CRN $\mathbf{C}$ having a reaction set $\mathcal{R}$, consider any $R = (I, P) \in \mathcal{R}$. We call the signal molecules consumed in $I - P$ *proper reactants* and those produced in $P - I$ *proper products*. R is a *k-proper chemical reaction equation* (or simply a proper chemical reaction equation) if and only if $|I - P| = |P - I| = k$. We say that $\mathbf{C}$ is a *k-proper Chemical Reaction Network (proper CRN)* (or simply a proper CRN) if all reactions are proper and k is the maximum number of proper inputs of all reactions in $\mathcal{R}$. We observe the following obvious, but useful results.

Lemma 1. A proper CRN with initial signal multiset S_0 will always have $|S_0|$ free signal molecules during a computation.

Lemma 2. The *space complexity of a tagged CRN* $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}}, T, T_0 \rangle$ with initial signal multiset S_0 and initial tag multiset T_0 that is also a proper CRN is $|S_0| + |T_0|$.

1.2.5 Realizing CRNs with DSDs

Soloveichik et al. [122] showed that arbitrary CRNs could be realized by using DNA signal strands to represent the signal molecules and by using a cascade of toehold mediated strand displacements to implement chemical reaction equations. Qian et al. [98] proposed an alternate construction — hereafter called the *QSW construction* — that is capable of simulating bi-molecular, and higher-order, chemical reactions. Specifically, the construction can exchange a multiset of signal strands (the reactants) for another multiset of signal strands (the products) through a sequence of toehold mediated strand displacements. Signal strands are of the same form: a negative recognition long domain ^{-}d, followed by a universal toehold t, followed by a positive recognition long domain ^{+}d. Signals and additional auxiliary strands that will be produced by a reaction are initially bound strands on a template strand. Additional unbound strands, consisting of a single long domain and a single universal toehold are used to effect the cascade of toehold mediated strand displacements. All toehold domains are universal and are therefore not labeled in the following figures. The one template strand for each reaction has the property that all of its long domains and all but one of its toehold domains are bound. We call these *saturated template strands*. We refer to the saturated template complex and associated auxiliary strands, collectively, as a transformer.

For example, Figure 1.10 shows an implementation of the chemical reaction equation $A + B \rightleftharpoons C + D$ using the QSW construction. The forward reaction is depicted from top to bottom and it can be seen that the signal strands C and D are initially bound to the template strand of the transformer. The reverse reaction is depicted from bottom to top. Realizing a reaction equation with more or less reactants and products is straightforward and involves modifying the template strand appropriately, adding the necessary auxiliary unbound strands,

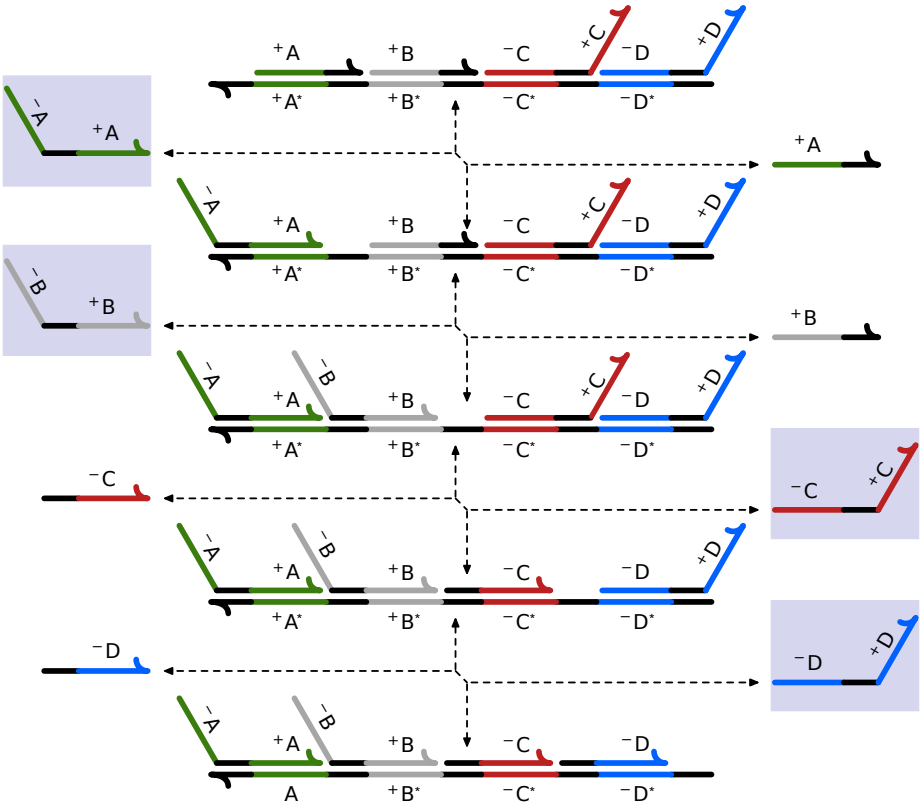


Figure 1.10: A strand displacement implementation of the bi-molecular chemical reaction equation $A + B \rightleftharpoons C + D$ using the construction proposed by Qian et al. [98].

and adding the necessary bound strands that will eventually be produced as signal strands. The QSW construction guarantees us the following result for the types of systems we study.

Theorem 1 (Qian et al. [98])**.** Any tagged CRN requiring $O(s)$ space can be realized and simulated by a DSD in $O(poly(s))$ space assuming all strand displacements are *legal*.

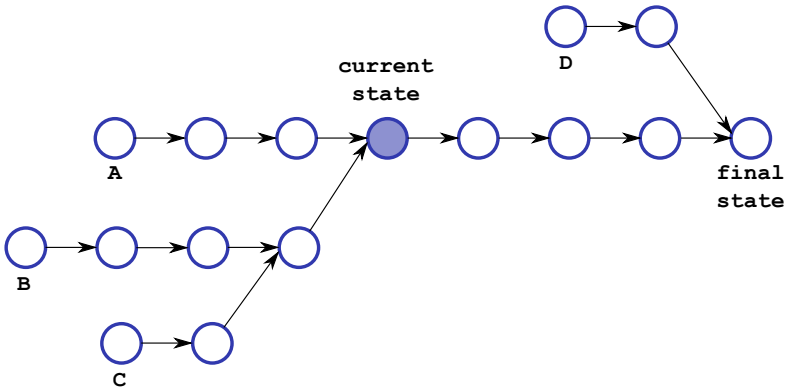
In related work, Cardelli [15, 72] has shown how primitives that support concurrent models of computation, such as fork and join gates, can be implemented using strand displacement systems. Many of the techniques used in the QSW construction are similar to those of Cardelli's constructions: for example, the signal strands share a common universal toehold while the long domains are distinct, and do not use branched structures. To effect an abstract chemical reaction equation with i reactants and i products, the QSW construction uses a cascading of toehold mediated strand displacements whereby the reactants are first consumed (by a transformer) and products are then produced by further strand displacements. This order of events is similar to an i-way join followed by an i-way fork of Cardelli. In this work, we will make use of a modified version of the QSW construction that we describe in Chapter 4.

1.2.6 Energy efficient computation

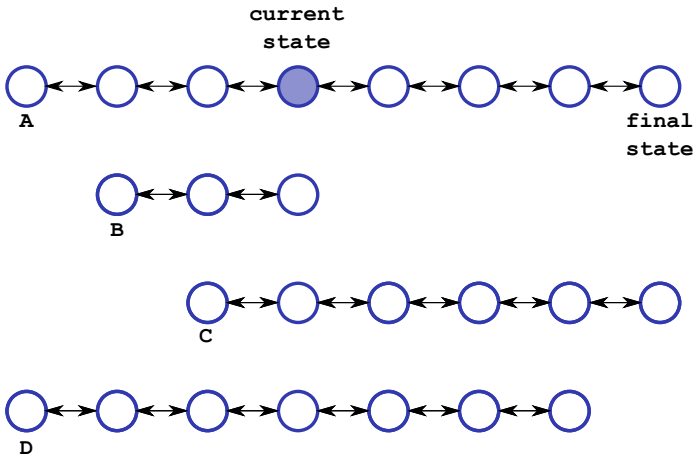
Aside from the potential biological and chemical applications, DSDs and CRNs are also of independent interest due to their promise for realizing *energy efficient computation*. Rolf Landauer proved that logically irreversible computation—computation as modeled by a standard Turing machine—dissipates an amount of energy proportional to the number of bits of information lost, such as previous state information, and therefore cannot be energy efficient [69]. Surprisingly, Charles Bennett showed that, in principle, energy efficient computation is possible, by proposing a universal Turing machine to perform *logically reversible computation* and identified nucleic acids (RNA/DNA) as a potential medium to realize logically reversible computation in a physical system [8].

A logically reversible computation is a form of deterministic computation. For our purposes, it suffices to understand the important difference distinguishing these two classes of computation. A *configuration graph* of a computation has a node for every possible state on every possible input for the underlying Turing machine being modeled. There is a directed edge from node i to node j if and only if state j is reachable from state i in a single state-transition of the Turing machine. An example of a deterministic configuration graph, for four different inputs (source nodes A-D) having a common final state is given in Figure 1.11a. Consider the path from A to the sink node labeled *final state*. Every node along the path has an out-degree of at most 1, making the computation deterministic. This is also true for the other source nodes.

The computation in this example is not reversible. However, if every directed edge is replaced with two edges, one for each orientation (or equivalently



(a) Deterministic computation



(b) Logically reversible computation

Figure 1.11: Example configuration graphs, induced on four different inputs, for (a) deterministic computation, and (b) logically reversible computation. Nodes represent possible states in a computation and directed edges denote valid state transitions.

by an undirected edge), we get a configuration graph for *symmetric* computation. With this change, we lose determinism. Consider a computation that has begun from node A, has reached the node labeled *final state*, and is now reversing towards its initial state. Once the computation reaches the node labeled *current state*, then a non-deterministic choice must be made. In essence, the computation does not currently have enough information to (deterministically) return to its initial state. Thus, information is lost.

Contrast this with the logically reversible configuration graph of Figure 1.11b, again shown for four different inputs (source nodes A-D). Importantly, a logically reversible computation for a particular input forms a chain which is unconnected to any state for any other possible input. This means any state along the chain can be deterministically reached from any other state along the chain. Thus, information is *not* lost. Even though non-terminal nodes along the chain have two possible choices of where to next proceed, the computation is still deterministic as one choice is always the previous state of the computation. (Retreating to the previous state is equivalent to the transition never having occurred.) The important point is that at any given node, the computation cannot proceed to more than one other node that is not the previous state. All of the molecular programs we propose in this thesis have this property.

1.3 Objectives

There is a need to find minimum-energy barrier folding pathways of nucleic acids not only in the context of biological processes, but also in the molecular programs which leverage them. When designing molecular programs, knowledge of folding pathways can help to debug the intended behaviour and also to formally verify the correctness of a program; for instance, by ensuring that certain states are unreachable by a low-energy barrier folding pathway. Still, the problem complexity of finding pathways with this property remained unknown at the outset of this research. Understanding the complexity of this problem could have a number of implications. If the problem is easy (in P), then an efficient and effective algorithm may be developed to better understand both naturally occurring and designed folding pathways. If the problem is hard, it may be possible to use folding pathways for non-trivial computation within chemical and biological systems. It is our first objective to elucidate the problem complexity of finding minimum-energy barrier folding pathways.

Regardless of problem complexity, there is a need for an exact algorithm that is efficient in practice. To date, all exact algorithms have time and space complexity that is exponential in the size of the input (length of nucleic acid strand(s)). Can this be improved?

It is also our aim to understand the computational power of deterministic molecular programs that leverage folding pathways. In this context, there was a particular need to understand space complexity of molecular programs. In the context of molecular programs operating in a reaction volume of a closed system, space can be thought of as the necessary size of that volume to fit all

molecules necessary to complete a computation. Can a *biological soup* of nucleic acids having total size $\Theta(n)$ perform a computation, by means of a folding pathway, of $\Theta(2^n)$ steps? If so, such a program would need to be space-efficient and reuse strands. At the outset of this thesis, DSD implementations did not efficiently reuse strands and therefore accumulated *waste*—inert strands that remain present in the reaction volume. We will discuss the details and consequences of strand re-use in Chapter 4. This question can be expanded to ask: what are the limitations to deterministic space-efficient molecular programming via folding pathways, if any? As base-pair formation is an inherently reversible process, we also explore the limits of logically-reversible, and thus energy-efficient computation of DSDs and folding pathways. Along the way, we are interested in understanding the complexity of a number of related problems. Can deterministic DSDs and their underlying folding pathways be verified to be correct (*i.e.* certain states are reachable within a certain energy barrier, while others are not)?

1.4 Contributions

We now describe the contributions of this part of the thesis, in the order they are discussed. By folding pathway, we mean pseudoknot-free nucleic acid folding pathway using the simple energy model.

1. We show that finding a direct folding pathway with minimum energy barrier is NP-complete for both the single strand and multiple strand cases.

2. We give a graph-theoretic algorithm for finding direct folding pathways with minimum energy barrier for the single strand case and discuss how it can be extended to the multiple strand case. For an instance having n arcs, the algorithm has a worst case time complexity exponential in n, but a space complexity only polynomial in n and is shown to be efficient in practice for most of the experimental instances evaluated. A feature of the algorithm is the ability to identify a succinct representation of all minimum free-energy structures between the initial and final structure of an instance in polynomial time.

3. We show that finding a direct-with-repeats folding pathway with minimum energy barrier is NP-complete for the single strand case and NP-hard for the multiple strand case.

4. We give the first example of a minimum energy barrier (indirect) folding pathway for multiple interacting strands whose length is exponential in the combined length of participating sequences. Our example is a DSD implementation of a binary-reflecting Gray code counter. An n-bit counter deterministically advances through 2^n states using only $poly(n)$ space. This demonstrates that deterministic DNA strand displacement (DSD) systems are capable, in principle, of space-efficient computation. An assumption

of this construction is that certain strand species exist as a single copy, rather than in an unbounded concentration.

5. We give the first proof that certain classes of chemical reaction networks (CRN), such as the underlying CRN implemented by our DSD Gray code counter, cannot be space efficient if all species of molecules are assumed to exist in concentration (multiple copies), rather than as a single copy. This implies the counter lacks determinism when all strands are present in concentration. We generalize this result to show that it is not possible to design any deterministic chemical reaction network that performs more than a linear number of deterministic computation steps in a reaction volume of a closed system (*i.e.,* as modeled by a tagged CRN), unless certain molecules exist as a single copy (*i.e.,* an exact count of the molecules is necessary to ensure determinism).

6. We demonstrate that any space-bounded computation can be solved by a space and energy efficient DNA strand displacement system, and thus low energy barrier (indirect) folding pathways of multiple interacting strands. We achieve the result by first giving a space efficient molecular program that can solve any arbitrary (unquantified) Boolean formula. We evolve the program to consider quantified Boolean formulas and further transformations to achieve our overall result. In the process we demonstrate a number of techniques useful for logically-reversible computation such as traversing a complete binary tree. Given our results of bullet 5, we must assume that certain molecules exist as a single copy to achieve these results.

7. We characterize the complexity of verification and model checking of deterministic chemical reaction networks and DNA strand displacement systems by showing that the reachability problems associated with these models are PSPACE-hard. We fully characterize restrictions of the models that are PSPACE-complete.

8. We relate our molecular programming results (bullets 6 & 7) with our earlier study of nucleic acid folding pathways by incorporating our quantified Boolean formula solver implementation (developed in Chapter 5) into a proof that predicting indirect folding pathways with minimum energy barrier for multiple interacting strands is PSPACE-complete.

9. We motivate and propose a refinement to the ReversibleSPACE complexity class to better model the inherent properties of current molecular programming domains.

A summary of the known complexity for the various folding pathway problems studied in this thesis is given in Table 1.1.

Folding pathway energy barrier problem		Hardness	Notes
Single strand	direct	NP-complete	Shown in Chapter 2. Solvable in practice by a graph theoretic algorithm given in Chapter 3.
	direct with repeats	NP-complete	Shown equivalent to the direct folding pathway problem in Chapter 3.
	indirect	open	If hard, could lead to novel molecular programming methods. Otherwise, an efficient algorithm could shed light on biological pathways and energy landscapes.
Multiple interacting strands	direct	NP-complete	Hard by restriction to the single strand case. Gives insight into direct folding pathways typical of DSD systems.
	direct with repeats	NP-hard	Hard by restriction to the single strand case. Not clear if problem is in NP.
	indirect	PSPACE-complete	Shown in Chapter 5. Gives many insights into DSDs, CRNs, and logically reversible computation.

Table 1.1: The complexity of folding pathway energy barrier problems for the simple energy model.

1.5 Outline

Chapter 2 formally introduces the minimum-energy barrier folding pathway problem and resolves the complexity of direct folding pathways. The hardness proof is quite technical. We note that reading the proof is unnecessary to understand the following chapters; knowing that finding minimum energy-barrier direct folding pathways is NP-complete is sufficient. The remaining chapters are much more accessible. In Chapter 3 we introduce an algorithm for finding direct folding pathways having minimum energy barriers. The algorithm makes interesting use of elegant graph decomposition techniques and may be of interest to theorists searching for open problems motivated by biological questions. In Chapter 4 we turn our attention towards molecular programming motivated by folding pathways and investigate the potential and peril of space efficient computation in these models. In Chapter 5 we explore how any deterministic computation that halts can be implemented by a space and energy efficient molecular program that leverages folding pathways. We also resolve the complexity of related problems, including prediction of folding pathways involving multiple interacting strands. In Chapter 6 we summarize our results and motivate the need for refined complexity models to more accurately characterize existing molecular programs.

Chapter 2

Complexity of predicting minimum energy barrier folding pathways

In this chapter, we study the computational complexity of the energy barrier problem for nucleic acids: what energy barrier must be overcome for one or more DNA or RNA molecule(s) to adopt a given final secondary structure, starting from a given initial secondary structure? The results presented here are a first step towards solving the energy barrier problem. Our results pertain to restricted types of folding pathways, namely *direct* folding pathways. Such pathways were introduced by Morgan and Higgs [83]. A folding pathway from secondary structure $\mathcal{I}$ to $\mathcal{F}$ is *direct* if the only arcs which are added are those from $\mathcal{F}-\mathcal{I}$ and the only arcs which are removed are those from $\mathcal{I}-\mathcal{F}$. Beyond the importance of this simple model for the study of biological folding pathways (see Section 1.1), we note that most designed nucleic acid folding pathway systems that we are familiar with are direct [116, 120, 146, 147]. However, there are examples of designed *indirect* folding pathways, including the catalytic system of Zhang et al. [150] and the binary-reflecting Gray code counter we present in Chapter 4.

The whole of this chapter is dedicated to the proof of our first main result: finding direct folding pathways with minimum energy barrier is NP-complete. In our proof, we consider the folding pathways of single strands. At the end of the chapter, we extend the result to consider the case of multiple interacting strands. We begin with formal definitions and by listing existing results necessary to support our claim.

2.1 Preliminaries

A *secondary structure* $\mathcal{T}$ for an RNA (DNA) molecule of length n is a set of base pairs $i.j$, with $1 \leq i < j \leq n$, such that (i) each base index i or j appears in at most one base pair and (ii) the bases at indices i and j form a Watson-Crick (*i.e.,* `C-G`, `A-U`, or `A-T`) base pair. Since we represent secondary structures

Content from this chapter appears in the proceedings of the 15th Annual International Conference on DNA computing and Molecular Programming (DNA 2009) [82] and the Journal of Natural Computing [81].

using arc diagrams, we use the word *arc* interchangeably with base pair (see Figure 1.1). Our main results pertain to pseudoknot-free secondary structures. That is, structures with no crossing arcs. We assume a very simple energy model for secondary structures in which each arc contributes an energy of -1. Thus, as is roughly consistent with more realistic energy models, the more base pairs in a structure the lower its energy. We denote the energy of secondary structure $\mathcal{T}$ by $E(\mathcal{T})$.

Fix initial and final pseudoknot-free secondary structures $\mathcal{I}$ and $\mathcal{F}$. A *direct pseudoknot-free folding pathway* from $\mathcal{I}$ to $\mathcal{F}$ is a sequence of pseudoknot-free secondary structures $\mathcal{I} = \mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_r = \mathcal{F}$, where each $\mathcal{T}_i$ is obtained from $\mathcal{T}_{i-1}$ by either the addition of one arc from $\mathcal{F} - \mathcal{I}$ or the removal of one arc from $\mathcal{I} - \mathcal{F}$. Thus, there are exactly $|\mathcal{I} \triangle \mathcal{F}|$ (the size of the symmetric difference of the two structures) steps along a direct folding pathway. We call each such addition or removal an *arc operation* and we let $+x$ and $-x$ denote the addition and removal of the arc x, respectively. The $\mathcal{T}_i$'s which are not the initial nor the final structure are called *intermediate structures*. A folding pathway can thus be specified by its corresponding sequence of arc operations; we call this a *transformation sequence*. A *direct pseudoknot-free transformation sequence* specifies a folding pathway which is both direct and pseudoknot-free.

The *energy barrier* of a folding pathway $\mathcal{I} = \mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_r = \mathcal{F}$ is the maximum of $E(\mathcal{T}_i) - E(\mathcal{I})$, where the max is taken over all integers i in the range $1 \leq i \leq r$. The *energy difference* of each intermediate configuration $\mathcal{T}_i$ is defined as $\mathrm{E}(\mathcal{T}_i) - \mathrm{E}(\mathcal{I})$. For instance, the entire folding pathway illustrated in Figure 1.2 has an energy barrier of 2; whereas the structure labeled B has an energy difference of 1. If Π is the transformation sequence for this pathway, then the *energy barrier* of transformation sequence Π, denoted as $\Delta\,\mathrm{E}(\mathcal{I}, \mathcal{F}, \Pi)$, is defined to be the energy barrier of the corresponding folding pathway.

In our result, it is convenient to work with weighted arcs. To motivate why, note that the union $\mathcal{I} \cup \mathcal{F}$ of two pseudoknot-free secondary structures may be pseudoknotted, *i.e.,* may have crossing arcs, even when both $\mathcal{I}$ and $\mathcal{F}$ are pseudoknot-free. In a pseudoknotted structure, we use the term *band* to refer to a set of nested arcs, each of which crosses the same set of arcs. In a folding pathway from $\mathcal{I}$ to $\mathcal{F}$ which minimizes the energy barrier, we can assume without loss of generality that when one arc in a band of $\mathcal{I} \cup \mathcal{F}$ is added, then all arcs in the band are added consecutively. Similarly, we can assume without loss of generality that when one arc in a band is removed, then all arcs in the band are removed consecutively. Thus, it is natural to represent the set of arcs in a band as one arc with a weight equal to the number of arcs in the band. An example showing two bands represented by weighted arcs is given in Figure 2.1.

Hence we generalize the notion of secondary structure as follows. A *weighted arc* $I = (I^b, I^e)^{I^w}$ is specified by start and end indices $I^b < I^e$ and a weight I^w. We say that two weighted arcs I and J are *crossing* if either $I^b \leq J^b \leq I^e \leq J^e$, or $J^b \leq I^b \leq J^e \leq I^e$. A *configuration* is a set of weighted arcs. Configuration $\{I_i\}_{i=1}^n$ is *pseudoknot-free* if for all $1 \leq i < j \leq n$, I_i and I_j are not crossing. The energy of configuration $\mathcal{I} = \{I_i\}_{i=1}^n$ is $\mathrm{E}(\mathcal{I}) = -\sum_{i=1}^n I^w$. The previous definitions can easily be generalized to weighted arcs. We can now formally

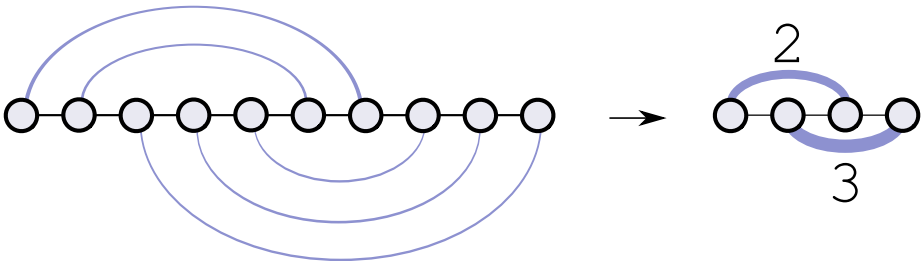


Figure 2.1: The three arcs on the bottom all conflict with the same two arcs on the top, and vice versa. Thus, each forms a *band* of arcs. Each band is collapsed into a single arc with weight equal to the size of the band.

define the main problem studied in this chapter.

Problem 1. EB-DPFP (Energy Barrier for Direct Pseudoknot-free Folding Pathway of a single strand)
Instance: Given two pseudoknot-free configurations $\mathcal{I} = \{I_i\}_{i=1}^{n}$ (initial) and $\mathcal{F} = \{F_i\}_{i=1}^{m}$ (final) of a single strand, and integer k.
Question: Is there a direct pseudoknot-free transformation sequence S such that the energy barrier of S, in the simple energy model, is at most k?

The reduction in our result begins with an arbitrary instance of the 3-PARTITION problem.

Problem 2. 3-PARTITION
Instance: Given $3n$ integers $a_1, \ldots, a_{3n}$ such that $\sum_{i=1}^{3n} a_i = nA$ and $A/4 < a_i < A/2$ for each i.
Question: Is there a partition of the integers $\{1, \ldots, 3n\}$ into n disjoint triples $\mathrm{G}_1, \mathrm{G}_2, \ldots, \mathrm{G}_n$ such that the sum of all a_j, where j belongs to G_i, is equal to A, *i.e.*, $\mathrm{c}(\mathrm{G}_i) = \sum_{j \in \mathrm{G}_i} a_j = A$, for each $i = 1, \ldots, n$?

Note the use of the notation $\mathrm{c}(\mathrm{G}_i)$. We will use this throughout to simplify our language. Importantly, we note the following result.

Theorem 2 (Garey, Johnson (1979) [41])**.** The 3-PARTITION problem is NP-complete even if A is polynomial in n.

The choice of the 3-PARTITION problem was not arbitrary. It is known as a *strongly* NP-complete problem, and therefore remains hard even when A is polynomial in n. This is important for our reduction as the number of arcs we create in the corresponding folding pathway instance will be proportional to A (in unary). Thus, if A were exponential in n, then the number of arcs would be exponential in n and the reduction from the 3-PARTITION problem instance would take exponential time, and not the requisite polynomial time. For this very reason, we did not make use of the *weakly* NP-complete PARTITION

problem—partition a set of integers into two sets with the same sum—as that problem is solvable in pseudo-polynomial time when the sum of all integers, and thus A, is polynomial in n. Note that the 3-PARTITION problem is in P if A is a constant; thus, we will ensure that A is polynomial in n in our reduction. For more information regarding this important distinction between *weakly* and *strongly* NP-complete problems, the reader is directed to Garey & Johnson [41].

2.2 Result

Theorem 3. The EB-DPFP problem, namely the energy barrier for direct pseudo-knot-free folding pathway problem, is NP-complete.

We note that the theorem does not require the energies of the initial and final structures to be minimum and indeed they can be different, as illustrated in Figures 2.1 and 2.3.

Proof. It is straightforward to show that the EB-DPFP problem is in NP. Given an instance $(\mathcal{I}, \mathcal{F}, k)$, it is sufficient to non-deterministically guess a direct folding pathway from $\mathcal{I}$ to $\mathcal{F}$, and to verify that the energy barrier of this path is at most k. Note that the length of any such pathway is at most $|\mathcal{I}| + |\mathcal{F}|$.

To show that the EB-DPFP problem is NP-hard, we provide a reduction from the 3-PARTITION problem. We first provide a formal description of the reduction, then provide some intuition as to why the reduction is correct, and then prove correctness in detail.

Consider an instance of the 3-PARTITION problem $A/2 > a_1 \geq \cdots \geq a_{3n} > A/4$ such that $\sum_{j=1}^{3n} a_j = nA$ and the value of A is bounded above by some polynomial in n. We define an instance $(\mathcal{I}, \mathcal{F}, k)$ of the EB-DPFP problem as follows. The initial configuration $\mathcal{I}$ contains weighted arcs $\{\bar{A}_{j,i};\ j = 1, \ldots, 3n,\ i = 1, \ldots, n\} \cup \{\tilde{A}_{j,i};\ j = 1, \ldots, 3n,\ i = 1, \ldots, n\} \cup \{\tilde{T}_i;\ i = 1, \ldots, n\}$. The final configuration $\mathcal{F}$ is $\{A_{j,i};\ j = 1, \ldots, 3n,\ i = 1, \ldots, n\} \cup \{T_i;\ i = 1, \ldots, n\}$. The arcs are organized as in Figure 2.2. Intuitively, the various T sets of weighted arcs are associated with the n triples, while the various A sets are associated with the $3n$ integers of the input. For each set of the weighted arcs corresponding to triples, there are weighted arcs corresponding to all $3n$ integers of the input. The reason for this is that a triple "chooses" its corresponding entries by adding arcs denoting the value of the entry, prior to a "validation" stage that occurs later in the folding pathway.

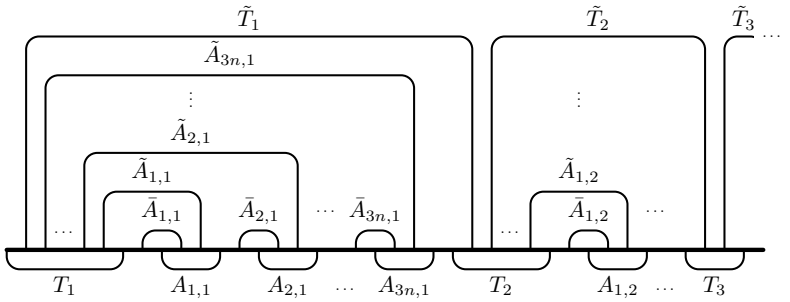


Figure 2.2: Organization of weighted arcs in the initial (top) and the final (bottom) configurations.

Formally, the arcs are organized as follows:

$$\begin{aligned}
T_1^b < \tilde{T}_1^b < \tilde{A}_{3n,1}^b < \cdots < \tilde{A}_{1,1}^b < T_1^e < \bar{A}_{1,1}^b, \\
T_i^b < \tilde{T}_{i-1}^e < \tilde{T}_i^b < \tilde{A}_{3n,i}^b < \cdots < \tilde{A}_{1,i}^b < T_i^e < \bar{A}_{1,i}^b, \ \forall i = 2, \ldots, n, \\
\bar{A}_{j,i}^b < A_{j,i}^b < \bar{A}_{j,i}^e < \tilde{A}_{j,i}^e < A_{j,i}^e, \ \forall i = 1, \ldots, n, \\
\forall j = 1, \ldots, 3n, \\
A_{j,i}^e < \bar{A}_{j+1,i}^b, \ \forall i = 1, \ldots, n, \\
\forall j = 1, \ldots, 3n-1, \\
A_{3n,i}^e < T_{i+1}^b, \ \forall i = 1, \ldots, n-1, \\
A_{3n,n}^e < \tilde{T}_n^e \,.
\end{aligned}$$

The weights of arcs are set up as follows. For all $i = 1, \ldots, n$ and $j = 1, \ldots, 3n$:

$$\begin{aligned}
\tilde{A}_{j,i}^w &= 4ia_j, \\
\bar{A}_{j,i}^w &= k - (j-1)A - 4ia_j, \\
A_{j,i}^w &= k - jA.
\end{aligned}$$

Also,

$$\begin{aligned}
\tilde{T}_1^w &= k - (7n-4)A, \\
\tilde{T}_i^w &= k - (6n+8)nA - 4(n-1)iA, \ \forall i = 2, \ldots, n, \\
T_i^w &= k - (6n+8)nA, \ \forall i = 1, \ldots, n-1, \\
T_n^w &= k \,,
\end{aligned}$$

where $k > 4(5n^2 + n + 1)A$ is the energy barrier.

Before getting into the details of the proof, we next describe intuitively the key properties of the construction. The weights are chosen to ensure that the folding pathway with minimum energy barrier has the following properties. Here

we list only the arcs that are added and assume without loss of generality that all arc removals happen only when needed.

1. Initially a (possibly empty) sequence of $A_{j,i}$'s are added to the folding pathway. Intuitively, this corresponds to "triple choosing" for the initial set of integers. The added $A_{j,i}$'s define a potential solution $\mathrm{G}_1, \mathrm{G}_2, \ldots, \mathrm{G}_n$ to the 3-PARTITION problem in a natural way: G_i contains j if $A_{j,i}$ is in this initial sequence. As we will prove later, the weights ensure that the addition of each $A_{j,i}$ raises the energy difference. After $3n$ such additions, the energy difference is so high that no other $A_{j,i}$'s can be added. As a result, the weights impose certain desirable constraints on the G_i's which will help ensure that they (or a slight perturbation of the G_i's) form a valid solution.

2. Following the initially-added sequence of $A_{j,i}$'s, the T_i's must be added in increasing order of i (with no interspersed $A_{j,i}$'s). This is in part because of the placement of the $\tilde{T}_i$'s: adding T_1 requires only the removal of $\tilde{T}_1$, whereas adding T_i, for $i > 1$, requires the costlier removal of both $\tilde{T}_{i-1}$ and $\tilde{T}_i$. Thus, it becomes feasible to add T_i without exceeding the energy barrier only after T_{i-1} is added because, at that point, $\tilde{T}_{i-1}$ has already been removed. In addition, after adding T_1, the energy difference increases to the level that none of the $A_{j,i}$'s can be added (and stays there until addition of T_n).

3. Moreover, the T_i's can be added without exceeding the energy barrier only if the G_i's defined by the initial sequence of $A_{j,i}$'s actually is a valid solution. Intuitively, this is a "triple validation" to ensure all chosen triples form a valid solution. That is, if the G_i's are valid then for each i, at least three of the $A_{j,i}$'s are in the initial sequence and so at least three of the $\tilde{A}_{j,i}$'s (whose weights sum to at least $4iA$) were removed in the initial part of the pathway described in 1 above. This means that at most $n-3$ of the $\tilde{A}_{j,i}$'s remain to be removed before T_i can be added. The total weight of the remaining $\tilde{A}_{j,i}$'s is just low enough to ensure that they can be added without exceeding the energy barrier. In contrast, if the G_i's are not valid then for some i the weight of the $\tilde{A}_{j,i}$'s which must be removed in order to add T_i causes the energy barrier k to be exceeded.

Let us illustrate the construction of the proof with the following example. Assume that we want to partition the multiset of integers $\{\{10, 9, 8, 7, 7, 7\}\}$ into two sets ($n = 2$). Figure 2.3a shows the corresponding instance of the energy barrier problem. For each triple, there are weighted arcs denoting all integers in the input set. We have labeled the associated weighted arcs by their integer value from the input set and have coloured those associated with triple 1 as black, and those associated with triple 2 as white. Figure 2.3b shows a correct pathway, which selected two triples, $T_1 = \{\{10, 7, 7\}\}$ (corresponding to the black labels along the pathway during triple-choosing) and $T_2 = \{\{9, 8, 7\}\}$ (corresponding to the white labels along the pathway during triple-choosing), both of which

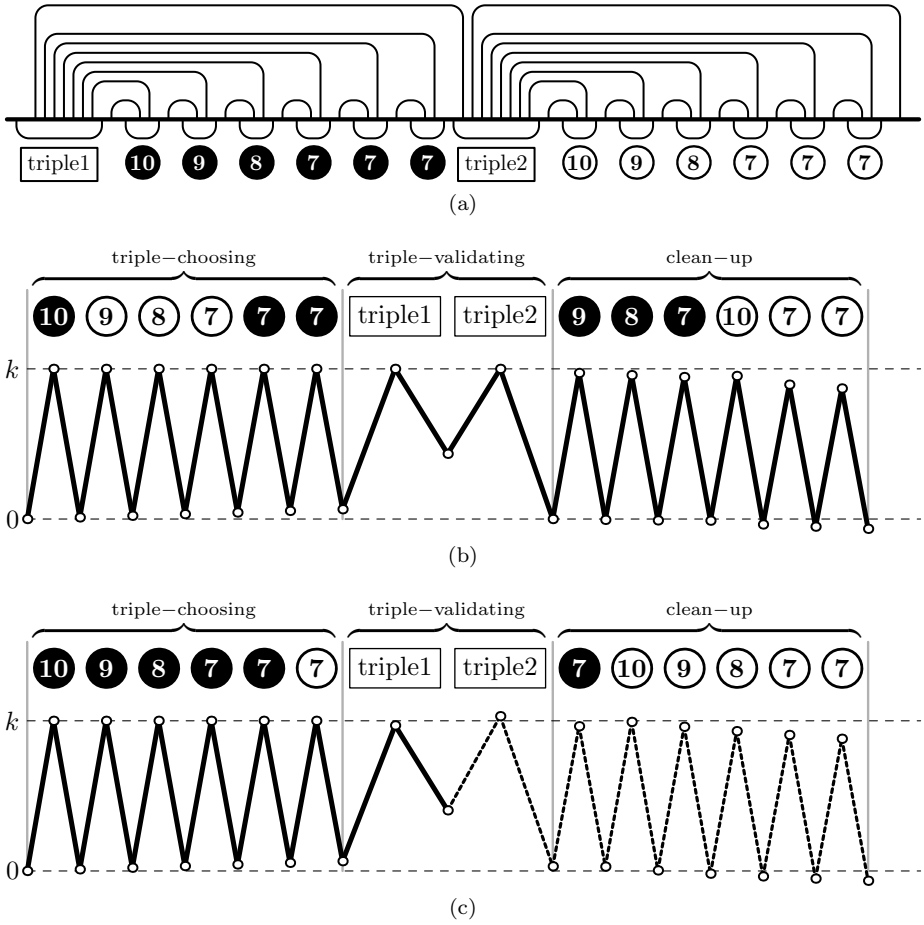


Figure 2.3: Illustration of the construction in the proof of Theorem 3: (a) The instance created for the set of integers $\{\{10, 9, 8, 7, 7, 7\}\}$. (b) The energy function stays within barrier k if and only if the partition sets are selected correctly ($T_1 = \{\{10, 7, 7\}\}$ and $T_2 = \{\{9, 8, 7\}\}$). (c) The energy function exceeds the barrier for an incorrect selection of partition sets ($T_1 = \{\{10, 9, 8, 7, 7\}\}$ and $T_2 = \{\{7\}\}$). The dashed lines depict hypothetical progress of the pathway for some energy barrier larger than k.

sum to 24. By construction of the proof, the portion of the folding pathway corresponding to the triple validation stage is able to proceed within barrier k. However, in the incorrect pathway shown in Figure 2.3c, where selection does not result in two equal triples (*i.e.,* $T_1 = \{\{10, 9, 8, 7, 7\}\}$ and $T_2 = \{\{7\}\}$), the triple validation stage fails, forcing the barrier above k.

The remainder of this chapter formally proves that the EB-DPFP instance has a solution with energy barrier at most k if and only if the 3-PARTITION instance $a_1, \ldots, a_{3n}$ has a solution. In showing this result, we must demonstrate a number of properties that the construction enforces, such as ensuring that elements of the input set are selected exactly once.

First, assume that the 3-PARTITION instance has a solution $\mathrm{G}_1, \ldots, \mathrm{G}_n$, where $\mathrm{G}_i = \{j_{i,1}, j_{i,2}, j_{i,3}\}$. Let $f(j) = i$ if $j \in \mathrm{G}_i$, for every $j = 1, \ldots, 3n$. We will show that the transformation sequence

$$-\bar{A}_{1,f(1)}, -\tilde{A}_{1,f(1)}, +A_{1,f(1)}, \ldots, -\bar{A}_{3n,f(3n)}, -\tilde{A}_{3n,f(3n)}, +A_{3n,f(3n)}, \tag{2.1}$$

$$\underbrace{-\tilde{A}_{1,1}, \ldots, -\tilde{A}_{3n,1}}_{\text{without } -\tilde{A}_{j_{1,1},1}, -\tilde{A}_{j_{1,2},1}, -\tilde{A}_{j_{1,3},1}} , -\tilde{T}_1, +T_1, \ldots,$$

$$\underbrace{-\tilde{A}_{1,n}, \ldots, -\tilde{A}_{3n,n}}_{\text{without } -\tilde{A}_{j_{n,1},n}, -\tilde{A}_{j_{n,2},n}, -\tilde{A}_{j_{n,3},n}} , -\tilde{T}_n, +T_n, \tag{2.2}$$

$$\underbrace{-\bar{A}_{1,1}, +A_{1,1}, -\bar{A}_{1,2}, +A_{1,2}, \ldots, -\bar{A}_{3n,n}, +A_{3n,n}}_{\text{without indexes } 1, f(1);\ 2, f(2);\ \ldots;\ 3n, f(3n)} \tag{2.3}$$

is pseudoknot-free with energy barrier exactly k. For clarity, the $-$ sign marks the arcs from the initial configuration which are being removed and the $+$ sign marks the arcs from the final configuration which are being added. It is easy to see that the sequence is pseudoknot-free, since

- each $A_{j,i}$ only crosses $\tilde{A}_{j,i}$ and $\bar{A}_{j,i}$ in the initial configuration and it is added only when these two arcs are already removed; and
- each T_i only crosses the following arcs in the initial configuration: $\tilde{T}_{i-1}$ (if $i > 2$), $\tilde{T}_i$ and $\tilde{A}_{1,i}, \ldots, \tilde{A}_{3n,i}$ and they are all removed before T_i is added.

Second, let us verify that the energy difference of each intermediate configuration is at most k. Figure 2.4 summarizes the sequence of energy differences along the pathway given in lines (2.1), (2.2) and (2.3) above; we next provide the details. First, in line (2.1), by induction, for each $j = 1, \ldots, 3n$, before removing $-\bar{A}_{j,f(j)}, -\tilde{A}_{j,f(j)}$ the energy difference is $(j-1)A$ and after removal it is k. Then after adding $+A_{j,f(j)}$ it decreases to jA. At the end of line (2.1), the energy difference is $3nA$. Next, we need to check that the sum of weights of arcs

$$\underbrace{-\tilde{A}_{1,1}, \ldots, -\tilde{A}_{3n,1}}_{\text{without } -\tilde{A}_{j_{1,1},1}, -\tilde{A}_{j_{1,2},1}, -\tilde{A}_{j_{1,3},1}} , -\tilde{T}_1$$

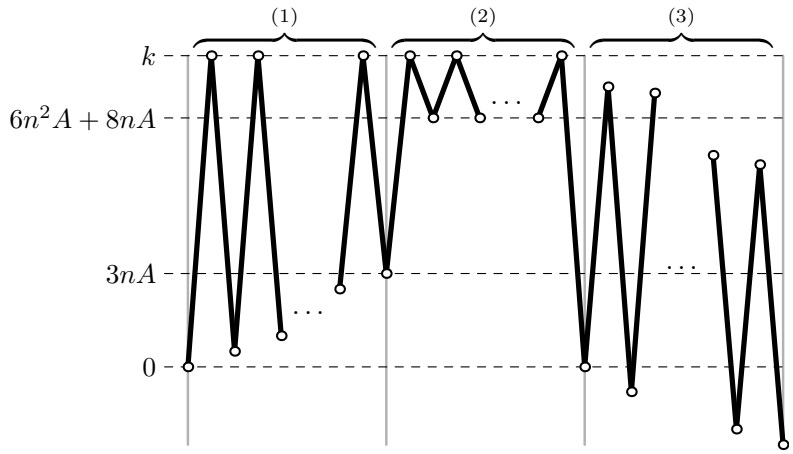


Figure 2.4: Illustration of the sequence of energy difference changes on the folding pathway described in lines (2.1), (2.2) and (2.3). Details are discussed in the text of the chapter.

is at most $k - 3nA$. The sum of weights of these arcs is exactly

$$\sum_{j=1}^{3n} \tilde{A}^w_{j,1} - \sum_{\ell=1}^{3} \tilde{A}^w_{j_{1,\ell},1} + \tilde{T}^w_1 = \sum_{j=1}^{3n} 4a_j - \sum_{\ell=1}^{3} 4a_{j_{1,\ell}} + k - 7nA + 4A$$
$$= 4nA - 4A + k - 7nA + 4A = k - 3nA\,.$$

Thus, just before adding $+T_1$, the energy difference is again exactly k. And after adding $+T_1$, it is $6n^2A + 8nA$. Similar calculations show that the energy difference will alternate between k, after each removal subsequence, and $6n^2A + 8nA$, after each addition of $+T_i$, in line (2.2) with the exception of the last addition, when the energy difference is 0. In line (2.3), all remaining arcs from the initial configuration ($-\bar{A}_{j,i}$) are removed and all remaining arcs from the final configuration ($+A_{j,i}$) are added—this is the *clean up* phase. Note that each removal is possible since $\bar{A}^w_{j,i} < k$ and after processing each pair $-\bar{A}_{j,i}, +A_{j,i}$, the energy difference only decreases since $\bar{A}^w_{j,i} - A^w_{j,i} = A - 4ia_j < 0$.

Now, assume that there is a pseudoknot-free transformation sequence S with the energy barrier at most k. From S, we will construct a solution for the original 3-PARTITION instance and show that it is a valid solution. We organize our proof into three parts, in line with the three properties described in the intuition at the start of the proof.

Consider the subsequence of S containing only additions, *i.e.,* arcs from the final configuration. Let S^+ denote this subsequence. We assume without loss of generality that all removals in S happen only when needed, *i.e.,* the next addition would not be possible without those removals. Hence, the subsequence S^+ determines the whole sequence S. By *processing* an arc $+I$ in S^+ we mean

(i) removal of all arcs $-J$ in S immediately preceding $+I$ (that is, $-J$ does not precede any other $+I'$ in S^+) and (ii) adding $+I$.

The first part of our proof considers the prefix of S^+ just before the first T_ℓ is added. Let this prefix be:

$$+A_{j_1,i_1}, +A_{j_2,i_2}, \ldots, +A_{j_M,i_M} \tag{2.4}$$

where M is the number of $+A_{j,i}$'s added before $+T_\ell$. We use this prefix to define a potential solution to the 3-PARTITION problem:

$$\mathrm{G}_i = \{j_\ell;\ i_\ell = i\},$$

for every $i = 1, \ldots, n$.

Ultimately we will show that the G_i's (or a slight perturbation of the G_i's) form a solution to the 3-PARTITION problem. Towards this goal, our first two lemmas below prove some useful properties of the G_i's that can be inferred from the weights of the arcs in the folding pathway prefix (2.4) and the corresponding removed arcs. Let $|\,\mathrm{G}_i\,|_j$ denote the number of elements in G_i (over all i) with value at most j. In order for the G_i's to be a valid solution, $|\,\mathrm{G}_i\,|_j$ should be exactly j for all $j, 1 \le j \le 3n$. Intuitively, this condition ensures that elements are selected exactly once. Moreover it should be the case that $\sum_{i=1}^{n} \mathrm{c}(\mathrm{G}_i) = nA$ where $\mathrm{c}(\mathrm{G}_i)$ denotes the sum of a_j for $j \in \mathrm{G}_i$ (see the definition of 3-PARTITION). The statements of the two lemmas below assert somewhat weaker properties of the G_i's.

Lemma 3. For every $j = 1, \ldots, 3n$, $\sum_{i=1}^{n} |\,\mathrm{G}_i\,|_j \le j$. Consequently, $M \le 3n$.

Proof. Let $+T_\ell$ be the first $+T_i$ in S^+. Consider an $+A_{j,i}$ appearing before $+T_\ell$. Recall that before adding $+A_{j,i}$, we need to remove both $-\tilde{A}_{j,i}$ and $-\bar{A}_{j,i}$. Since, $\tilde{A}^w_{j,i} + \bar{A}^w_{j,i} = k - (j-1)A$, the energy difference has to be at most $(j-1)A$ for $+A_{j,i}$ to be added. Note that processing of each $+A_{j,i}$ appearing in S^+ before $+T_\ell$ will increase the energy difference by A, as it requires both $-\tilde{A}_{j,i}$ and $-\bar{A}_{j,i}$ to be removed first and $\bar{A}^w_{j,i} + \tilde{A}^w_{j,i} - A^w_{j,i} = k - (j-1)A - 4ia_j + 4ia_j - (k - jA) = A$. For instance, an $+A_{1,i}$ can only appear at the first position of the part of the subsequence S^+ before $+T_\ell$, since it requires the energy difference to be at most 0 and after any $+A_{j,i}$ is added, the energy difference increases to A. Thus, starting from the second position, no $+A_{1,i'}$ can be added before $+T_\ell$. Similarly, $+A_{j,i}$ can appear only in the first j positions of the subsequence of S^+ before $+T_\ell$. Due to this condition imposed by the construction, the lemma easily follows. □

In the next lemma we use double brackets to denote multisets: for example $\{\{1, 2, 2\}\}$ is the multiset with elements 1, 2, and 2 and $\{\{1, 1, 2\}\} \neq \{\{1, 2, 2\}\}$.

Lemma 4. $\sum_{i=1}^{n} \mathrm{c}(\mathrm{G}_i) \le nA - (3n - M)A/4$, where the equality happens only if $M = 3n$ and $\{\{a_{j_1}, \ldots, a_{j_M}\}\} = \{\{a_1, \ldots, a_{3n}\}\}$.

Proof. Let $b_1 \geq b_2 \geq \cdots \geq b_M$ be the sorted elements of the multiset $\{\{a_{j_1}, \ldots, a_{j_M}\}\}$. Note that $\sum_{i=1}^{n} \mathrm{c}(\mathrm{G}_i) = \sum_{j=1}^{M} b_j$. We will show that $b_j \leq a_j$ for every $j = 1, \ldots, M$. Suppose to the contrary that $b_j > a_j$ for some j. Hence, elements $b_1, \ldots, b_j$ belong to $\{\{a_1, \ldots, a_{j-1}\}\}$, *i.e.*, $|\{\{a_{j_1}, \ldots, a_{j_M}\}\}|_{j-1} \geq j$. This is a contradiction with Lemma 3. Hence, we have

$$\sum_{i=1}^{n} \mathrm{c}(\mathrm{G}_i) = \sum_{j=1}^{M} b_j \leq \sum_{j=1}^{M} a_j = nA - \sum_{j=M+1}^{3n} a_j \leq nA - (3n - M)A/4\,.$$

The equality happens only if $M = 3n$ (since $a_j > A/4$) and $b_j = a_j$, for every $j = 1, \ldots, 3n$. □

We now turn to the second part of our proof: we show that, following the initially-added sequence of $+A_{j,i}$'s, the T_i's must be added in increasing order of i. That is, the arcs $+T_1, \ldots, +T_n$ appear in the subsequence S^+ consecutively (with no $+A_{j,i}$ in between) and in this order. The next lemma shows that the first $+T_i$ in the sequence S^+ must be $+T_1$ and the following lemma reasons about the rest of the sequence of $+T_i$'s.

Lemma 5. The first $+T_i$ in S^+ is $+T_1$.

Proof. Let $+T_\ell$ be the first $+T_i$ in S^+. As argued in the proof of Lemma 3, after each $+A_{j,i}$, the energy difference increases by A. Hence, before adding $+T_\ell$, the energy difference is non-negative. Second, if $\ell > 1$ then to add $+T_\ell$, both $-\tilde{T}_{\ell-1}$ and $-\tilde{T}_\ell$ has to be removed. After their removal the energy difference would be at least $2k - 2(6n+8)nA - 4(n-1)(2\ell-1)A > k$, a contradiction. The last inequality follows by $k > 4(5n^2+n+1)A = 2(6n+8)nA - 4(n-1)(2n-1)A$. □

Hence, by the above lemma, the subsequence S^+ has the following form

$$+A_{j_1,i_1}, +A_{j_2,i_2}, \ldots, +A_{j_M,i_M}, +T_1$$

followed by the addition of all remaining $+A_{j,i}$'s and $+T_i$'s. The following lemma gives more detailed insight into the order of arcs in S^+.

In the remaining lemmas we adopt notation which was introduced by Graham, Knuth and Patashnik [44]:

$$[i > j] = \begin{cases} 1, & \text{if } i > j; \\ 0, & \text{otherwise,} \end{cases} \qquad \text{and} \qquad [i = j] = \begin{cases} 1, & \text{if } i = j; \\ 0, & \text{otherwise.} \end{cases}$$

Lemma 6. All T_i's appear in S^+ in one sequence and in increasing order.

Proof. Assume to the contrary that subsequence $+T_1, +T_2, \ldots, +T_p$ is followed by an arc $+I$ different from $+T_{p+1}$ in S^+, where $p < n$. This arc could be either $+A_{j,i}$ or $+T_\ell$, where $\ell > p+1$. We will show that both cases lead to a contradiction by lower bounding the energy difference of the intermediate configuration after adding $+T_p$.

As argued in the proof of Lemma 3, processing of each $+A_{j_m,i_m}$ will contribute A to the energy difference. Hence, before adding $+T_1$, the energy difference is non-negative. We will lower bound contributions of processing $+T_1, \ldots, +T_p$ to the energy difference. For every $i = 1, \ldots, p$, to process $+T_i$, we need to remove $-\tilde{T}_i$ and all $-\tilde{A}_{j,i}$ which were not yet removed. This will add to the energy difference

$$\begin{aligned}\tilde{T}_i^w + \sum_{j \notin T_i} \tilde{A}_{j,i}^w &\geq k - 3nA - [i > 1](6n+5)nA \\ &\quad - 4(n-1)iA + 4i \sum_{j=1,\ldots,3n} a_j - 4|\,\mathrm{G}_i\,|iA/2 \\ &> k - 3nA + [i > 1](6n+5)nA - 2|\,\mathrm{G}_i\,|nA\,,\end{aligned}$$

since only $|\,\mathrm{G}_i\,|$ arcs $-\tilde{A}_{j,i}$ have been removed before processing $+T_i$ and each $\tilde{A}_{j,i}^w = 4ia_j < 2nA$. Hence, the contribution of processing $+T_1$ is at least $k - 3nA - 2|\,\mathrm{G}_1\,|nA - T_1^w = (6n+5)nA - 2|\,\mathrm{G}_1\,|nA$, and the contribution of processing $+T_i$, for $i = 2, \ldots, p$, is at least $-2|\,\mathrm{G}_i\,|nA$. Since $\sum_{i=1}^p |\,\mathrm{G}_i\,| \leq M$ and by Lemma 3, $M \leq 3n$, the total contribution of adding $T_1, \ldots, T_p$ is at least $6n^2A + 5nA - 6n \cdot nA = 5nA$. Hence, the energy difference of the intermediate configuration before processing $+I$ is at least $5nA$.

Now, let us consider two cases depending on the type of arc $+I$. First, assume that $+I$ is a $+T_\ell$, for some $\ell > p+1$. Since $+T_{\ell-1}$ appears in S^+ after $+T_\ell$, to add $+T_\ell$, we need to remove both $-\tilde{T}_{\ell-1}$ and $-\tilde{T}_\ell$. Since the energy difference before removing $-\tilde{T}_{\ell-1}$ and $-\tilde{T}_\ell$ is positive (at least $5nA$), the lemma follows by the argument used in the proof of Lemma 5.

Second, assume that $+I$ is an $+A_{j,i}$. Before adding $+A_{j,i}$, the arc $-\bar{A}_{j,i}$ needs to be removed. Since $\bar{A}_{j,i}^w = k - (j-1)A - 4ia_j > k - (3n-1)A - 2nA > k - 5nA$, the energy difference after removing $-\bar{A}_{j,i}$ would be greater than $5nA + k - 5nA = k$, a contradiction. □

Hence, by the above lemmas, the subsequence S^+ has the following form

$$+A_{j_1,i_1}, +A_{j_2,i_2}, \ldots, +A_{j_M,i_M}, +T_1, +T_2, \ldots, +T_n$$

followed by the all remaining $+A_{j,i}$'s.

Moving on to the last part of the proof: we show that the G_i's defined by the initial sequence of $+A_{j,i}$'s form a valid solution (or can be perturbed slightly to form a valid solution) by arguing that only in this case can all of the T_ℓ's be added without exceeding the energy barrier. Specifically, we will show that $M = 3n$ and $\{\{a_{j_1}, \ldots, a_{j_{3n}}\}\} = \{\{a_1, \ldots, a_{3n}\}\}$. For this purpose, the next two lemmas prove lower bounds on sums of the $\mathrm{c}(\mathrm{G}_i)$'s.

Lemma 7. For every $\ell = 1, \ldots, n$, $\sum_{i=1}^{\ell} i(\mathrm{c}(\mathrm{G}_i) - A) \geq (M - 3n)A/4$.

Proof. To process $+T_\ell$, $-\tilde{T}_\ell$ and all remaining $-\tilde{A}_{1,\ell}, \ldots, -\tilde{A}_{3n,\ell}$ need to be removed. Specifically, this corresponds to all $-\tilde{A}_{j,\ell}$'s for which $j \notin \mathrm{G}_\ell$. Hence, the total weight of arcs which need to be removed is

$$\begin{aligned}\tilde{T}^w_\ell + \textstyle\sum_{j\notin T_\ell} \tilde{A}^w_{j,\ell} &= k - 3nA - [\ell > 1](6n+5)nA - 4(n-1)\ell A + 4\ell(nA - \mathrm{c}(\mathrm{G}_\ell)) \\ &= k - 3nA - [\ell > 1](6n+5)nA + 4\ell(A - \mathrm{c}(\mathrm{G}_\ell))\,.\end{aligned}$$

After removing these arcs, the energy difference will increase by this amount and then decrease by $T^w_\ell = k - (6n+8)nA$. Hence, the total change of the energy difference for adding $+T_\ell$ is $[\ell = 1](6n+5)nA + 4\ell(A - \mathrm{c}(\mathrm{G}_\ell))$.

It is easy to see, by induction on ℓ, that the energy difference before removing the arc for $+T_\ell$ is $MA + [\ell > 1](6n+5)nA + \sum_{i=1}^{\ell-1} 4i(A - \mathrm{c}(\mathrm{G}_i))$, since after processing subsequence $+A_{j_1,i_1}, \ldots, +A_{j_M,i_M}$, the energy difference is MA. Since the energy difference, after removing the necessary arcs before adding $+T_\ell$, must be at most k, we have

$$\begin{aligned}MA + [\ell > 1](6n+5)nA + \sum_{i=1}^{\ell-1} 4i(A - \mathrm{c}(\mathrm{G}_i)) \\ + k - 3nA - [l > 1](6n+5)nA + 4\ell(A - \mathrm{c}(\mathrm{G}_\ell)) \le k\end{aligned}$$

which simplifies to

$$\sum_{i=1}^{\ell} i(\mathrm{c}(\mathrm{G}_i) - A) \ge (M - 3n)A/4\,.$$

□

Using the inequalities from Lemma 7, we will lower bound the sum of $\mathrm{c}(\mathrm{G}_i)$'s.

Lemma 8. We have $\sum_{i=1}^{n} \mathrm{c}(\mathrm{G}_i) \ge nA - (3n - M)A/4$, where the equality happens only if $\mathrm{c}(\mathrm{G}_1) = A - (3n - M)A/4$ and $\mathrm{c}(\mathrm{G}_i) = A$, for every $i = 2, \ldots, n$.

Proof. We will multiply each inequality of Lemma 7 with the positive constant $1/\ell - [n > l]/\ell + 1$ and sum the inequalities:

$$\begin{aligned}\sum_{\ell=1}^{n} (1/\ell - [n > l]/(\ell+1)) \sum_{i=1}^{\ell} i(\mathrm{c}(\mathrm{G}_i) - A) \\ \ge \sum_{\ell=1}^{n} (1/\ell - [n > l]/(\ell+1))\,(M - 3n)A/4\,.\end{aligned}$$

Changing the order of the sums on the left hand side and using the fact that $\sum_{\ell=i}^{n}(1/\ell - [n > \ell]/(\ell+1)) = 1/i$ we obtain:

$$\sum_{i=1}^{n}(\mathrm{c}(\mathrm{G}_i) - A) = \sum_{i=1}^{n} i(\mathrm{c}(\mathrm{G}_i) - A) \sum_{\ell=i}^{n}(1/\ell - [n > \ell]/(\ell+1)) \ge (M - 3n)A/4,$$

and the lemma easily follows. The equality, in the resulting inequality, happens only if we have equality in all inequalities used in the summation. This would

imply that

$$\sum_{i=1}^{\ell} i(\mathrm{c}(\mathrm{G}_i) - A) = (M - 3n)A/4, \tag{2.5}$$

for all $\ell = 1, \ldots, n$. For $\ell = 1$, we have $\mathrm{c}(\mathrm{G}_1) - A = (M - 3n)A/4$, *i.e.,* $\mathrm{c}(\mathrm{G}_1) = A - (3n - M)A/4$. Subtracting Equation (2.5) for ℓ and Equation (2.5) for $\ell - 1$, we obtain $\ell(\mathrm{c}(\mathrm{G}_\ell) - A) = 0$, *i.e.,* $\mathrm{c}(\mathrm{G}_\ell) = A$. □

By Lemmas 4 and 8, we have $\sum_{i=1}^{n} \mathrm{c}(\mathrm{G}_i) = nA - (3n - M)A/4$, *i.e.,* we have equality in both Lemma 4 and Lemma 8. Thus, by Lemma 4, we have that $M = 3n$ and $\{\{a_{j_1}, \ldots, a_{j_{3n}}\}\} = \{\{a_1, \ldots, a_{3n}\}\}$.

Although this does not imply that $\mathrm{G}_1, \ldots, \mathrm{G}_n$ immediately forms a decomposition of the set $\{1, 2, \ldots, 3n\}$. For instance, if $a_1 = a_2$, the multiset $\{\{j_1, \ldots, j_{3n}\}\}$ could contain zero 1's and two 2's. However, this is easily solved with a slight perturbation of the solution. The sets $\mathrm{G}_1, \ldots, \mathrm{G}_n$ could be mapped to the decomposition of $\{1, 2, \ldots, 3n\}$ just by a sequence of replacements i's with j's assuming $a_j = a_{j+1} = \cdots = a_i$. This transformation simplifies the correspondence between the solution of two problems. Most importantly, by Lemma 8, we have $\mathrm{c}(\mathrm{G}_1) = A - (3n - M)A/4 = A$ and also $\mathrm{c}(\mathrm{G}_i) = A$ for all $i = 2, \ldots, n$. Hence, the sets $\mathrm{G}_1, \ldots, \mathrm{G}_n$ (possibly modified as described above) are the solution to the 3-PARTITION problem.

The reduction is polynomial as the sum of weights of all arcs (which is the total number of arcs in the unweighted instance) is

$$\sum_{i=1}^{n} \left(\tilde{T}_i^w + T_i^w + \sum_{j=1}^{3n} (\tilde{A}_{j,i}^w + \bar{A}_{j,i}^w + A_{j,i}^w) \right) < n \cdot 2k + 3n^2 \cdot 2k = \mathcal{O}(n^2 k) = \mathcal{O}(n^4 A)\,,$$

and A is assumed to be polynomial in n.

□

The problem of predicting folding pathways for a single strand generalizes to multiple interacting strands in the natural way. In this problem a configuration generalizes to consider base pairs between strands in addition to within strands. If there are n bases in all strands of an instance, then bases can be identified uniquely as a number in $[1, n]$. Thus, the current definition of a configuration also works for the multiple strand variation of the problem. We formally define the problem as follows.

Problem 3. EB-DPFP-MULTI (Energy Barrier for Direct Pseudoknot-free Folding Pathway of Multiple interacting strands)
Instance: Given two pseudoknot-free configurations $\mathcal{I} = \{I_i\}_{i=1}^{n}$ (initial) and $\mathcal{F} = \{F_i\}_{i=1}^{m}$ (final), of multiple interacting strands, and integer k.
Question: Is there a direct pseudoknot-free transformation sequence S such that the energy barrier of S, in the simple energy model, is at most k?

Since the length of a direct pathway for multiple interacting strands from an initial configuration $\mathcal{I}$ to a final configuration $\mathcal{F}$ has maximum length $|\mathcal{I}| + |\mathcal{F}|$, then the problem is in NP. By restriction to the single strand version of the problem, EB-DPFP, we can conclude the following result.

Theorem 4. The EB-DPFP-MULTI problem, namely the energy barrier for direct pseudoknot-free folding pathway of multiple interacting strands problem, is NP-complete.

2.3 Chapter summary

We have shown that the energy barrier problem for direct pseudoknot-free folding pathways is NP-complete, via a reduction from the 3-PARTITION problem. Thus, unless NP = P , there is no polynomial-time algorithm for calculating the energy barrier of direct folding pathways. This justifies the use of heuristics for estimating energy barriers [38, 83, 125, 143] and leads to the interesting question of whether or not there is an algorithm that is guaranteed to return the exact energy barrier and which works well on practical instances of the problem (while not in the worst case). This is the focus of the next chapter.

Our proof can help shed insight on energy landscapes. Consider an instance $(\mathcal{I}, \mathcal{F}, k)$ of the EB-DPFP problem which is derived from a "yes" instance of 3-PARTITION according to our construction. There are exponentially many possible prefixes (of the type shown in (2.4)) which could precede the addition of T_1, all of which do not exceed the energy barrier k. Of these, it may be that only one defines a valid solution of G_i's. Thus, if pathways are followed according to a random process, it could take exponential time for the random process to find the pathway with energy barrier k. This is because there are exponentially many initial prefixes which could lead to such a pathway of which only one can be extended to a pathway with barrier k.

In this chapter, we do not fully resolve the computational complexity of the general energy barrier problem, in which the pathway need not be direct. Two challenges in understanding the complexity of this problem which need to be considered are repeat arcs—arcs added and removed multiple times in a pathway—and temporary arcs—arcs not specified in the initial or final structure. The following chapter sheds further light on pathways containing repeat arcs.

Chapter 3

Predicting minimum energy barrier folding pathways

In the previous chapter we established that the direct energy barrier folding pathway problem is NP-complete. Still, there is a need for an exact algorithm that performs well in practice. This is exactly the focus of this chapter. We first generalize the folding pathway problem of a single strand to one defined in terms of bipartite graphs, allowing us to exploit the rich knowledge and algorithms found in graph theory. Later in the chapter, we will discuss results and extensions to the case of multiple interacting strands. While the algorithm we develop has exponential running time in the worst case, it is the first exact algorithm that uses only polynomial space. As we will show by an empirical evaluation in Section 3.3, the algorithm performs well in practice. Furthermore, the algorithm is inherently parallel; a property that could be exploited when solving hard instances. In the process of proving the correctness of the algorithm, we will resolve the complexity of the direct with repeats energy barrier folding pathway problem (see Section 3.4).

3.1 Preliminaries

We find it convenient to model the problem in terms of bipartite graphs and first develop some useful notation. For a pair of pseudoknot-free structures for the same RNA sequence, we define the *conflict graph* to be a bipartite graph $G[A, B]$ where A is the set of arcs from the first structure, B is the set of arcs from the second structure, and there is an edge in $E(G)$ between $a \in A$ and $b \in B$ if and only if a and b are crossing. An example is given in Figure 3.1. Throughout, we denote the neighbours of a vertex v, or set of vertices X in G as $N_G(v)$ and $N_G(X)$, respectively. We denote the subgraph of G induced by subsets $A' \subseteq A$ and $B' \subseteq B$ by $G/[A', B']$. Also, we denote the stability number of G, *i.e.,* the size of a maximum independent set in G, as $\alpha(G)$. We need a notion analogous to that of a pair of minimum free energy (MFE) structures[11] in the context of bipartite graphs. We say that G is *pairwise-optimal* if $\alpha(G) = |A| = |B|$. If

Content from this chapter appears in the proceedings of the 15th Annual Pacific Symposium on Biocomputing (PSB 2010) [129].

[11] A minimum free energy (MFE) structure has the lowest energy of any possible structure for a given molecule. In the simple energy model proposed by Morgan and Higgs [83], where each arc contributes -1 to the energy score, a pseudoknot-free MFE structure is one with a maximum number of non-crossing arcs.

A and B are MFE structures then G must be pairwise-optimal; otherwise the largest independent set in the conflict graph G would be a set of arcs with lower free energy than either A or B.

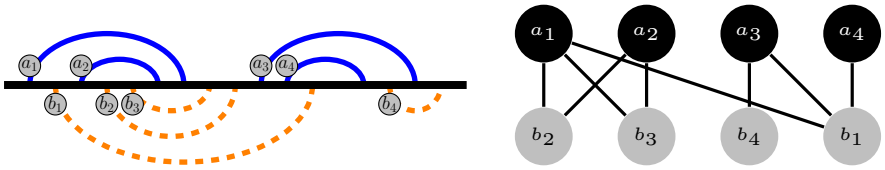


Figure 3.1: (left column) An example of an arc diagram representation of an initial and final structure of an RNA folding pathway, and (right column) the corresponding conflict graph. In the conflict graph, there is a node for every arc, and an edge between any pair of arcs that cross.

Let $G[A, B]$ be a pairwise-optimal bipartite graph. A *set pathway* for G is a sequence of independent sets $S_0, \ldots, S_m$, each of which is a subset of $A \cup B$, such that (i) $S_0 = A$, (ii) $S_m = B$ and (iii) for every $i = 1, \ldots, m$, $|S_{i-1} \triangle S_i| = 1$ (the size of the symmetric difference is one, *i.e.,* at each step one element is either added or removed). The *transformation sequence* corresponding to this set pathway is the sequence of singletons $S_0 \triangle S_1, \ldots, S_{m-1} \triangle S_m$. If an element appearing in the transformation sequence is not in the current set, then the element is meant to be added. If the element is in the current set, the element is meant to be removed. The set pathway is *direct* if its corresponding transformation sequence has no repeating elements. The *barrier* of the pathway (or its corresponding transformation sequence) is $k = \max_i |A| - |S_i|$. (Since A is a maximum independent set of G, it must be that $|A| - |S_i| \geq 0$ for all $i, 1 \leq i \leq m$.) We say that a set pathway is a ($\leq k$)-barrier set pathway or a k-barrier pathway if its barrier is $\leq k$ or $= k$, respectively. A *min-barrier* set pathway is a set pathway whose barrier is less than or equal to the barriers of any other set pathway for G. Consider the following problem:

Problem 4. EB-DSP (Energy Barrier for Direct Set Pathway)
Instance: Given a pairwise-optimal bipartite graph $G[A, B]$ and integer k.
Question: Is there a direct set pathway with barrier at most k for G?

An instance of the EB-DPFP (Energy Barrier for Direct Pseudoknot-free Folding Pathway) problem can be mapped to an instance of the EB-DSP (Energy Barrier for Direct Set Pathway) problem by constructing its conflict graph. We note that the mapping from an instance of EB-DPFP to an instance of EB-DSP is only immediate if both the initial and final secondary structures of the EB-DPFP instance are MFE structures. However, we demonstrate in Section 3.2.5 how this condition can be removed, such that any instance of EB-DPFP can be solved. See Figure 3.2 for an example which relates an instance of each problem. However, the Direct Set Pathway problem is actually a more general problem since

not every bipartite graph is realizable by a pair of pseudoknot-free structures. We characterize conflict graphs for the RNA direct folding pathway problem more accurately in Section 3.5.

3.2 An algorithm for the set barrier problem

Our algorithm for the Direct Set Barrier Problem uses two key ideas. The first is a *splitting strategy*: if for some proper non-empty subset B_1 of B the induced subgraph $G/[A_1, B_1]$ is pairwise-optimal, where $A_1 = N_G(B_1)$, then we can determine the solution for G by recursively solving the problem on the induced subgraphs $G/[A_1, B_1]$ and $G/[A \setminus A_1, B \setminus B_1]$ and combining their solutions. At some point, a subproblem $G'[A', B']$ cannot be split further as it contains exactly two maximum independent sets: A' and B'. In this case, we say that G' is *minimal pairwise-optimal*. Solving a minimal pairwise-optimal subproblem requires our second idea, a *cutting strategy* for reducing the size of minimal pairwise-optimal problem instances. After reducing the size, we can once again attempt to split the problem, and so on, until either a solution is found, or it is determined that one does not exist. In the following sections, we detail these strategies, the overall algorithm, its correctness and complexity, and the empirical performance of its implementation.

3.2.1 Splitting strategy

The hypothesis, in terms of the RNA folding pathway problem, that predicated the splitting strategy is simple: if one could identify a MFE structure C consisting of arcs from both the initial and final structures, A and B respectively, then there always exists an optimal pathway from A to B via C. The combination of Lemma 9 and Lemma 10 show that this hypothesis is correct, in terms of the direct set barrier problem; specifically, the resulting solution is (i) a valid set pathway, and (ii) optimal.

Lemma 9. Let $G[A, B]$ be a pairwise-optimal bipartite graph and let $G_1 = G/[A_1, B_1]$ be pairwise-optimal where B_1 is a proper non-empty subset of B and $A_1 = N_G(B_1)$. Let $G_2 = G/[A_2, B_2]$, where $A_2 = A \setminus A_1$ and $B_2 = B \setminus B_1$. Then

1. G_2 is pairwise-optimal, and
2. if T_1 and T_2 are $(\leq k)$-barrier transformation sequences for G_1 and G_2 respectively then T_1, T_2 is a $(\leq k)$-barrier transformation sequence for G.

Proof. Consider the first claim. Suppose to the contrary that G_2 is not pairwise-optimal. Then, in G_2, there must exist some maximum independent set $C \subseteq A_2 \cup B_2$, where $|C| > |B_2|$. Since $N_G(B_1) = A_1$, then $C \cup B_1$ is also a maximum independent set in G of size $|C| + |B_1|$, since $C \cap B_1 = \emptyset$. But $|C| + |B_1| > |B_2| + |B_1| = |B|$, contradicting that G is pairwise-optimal.

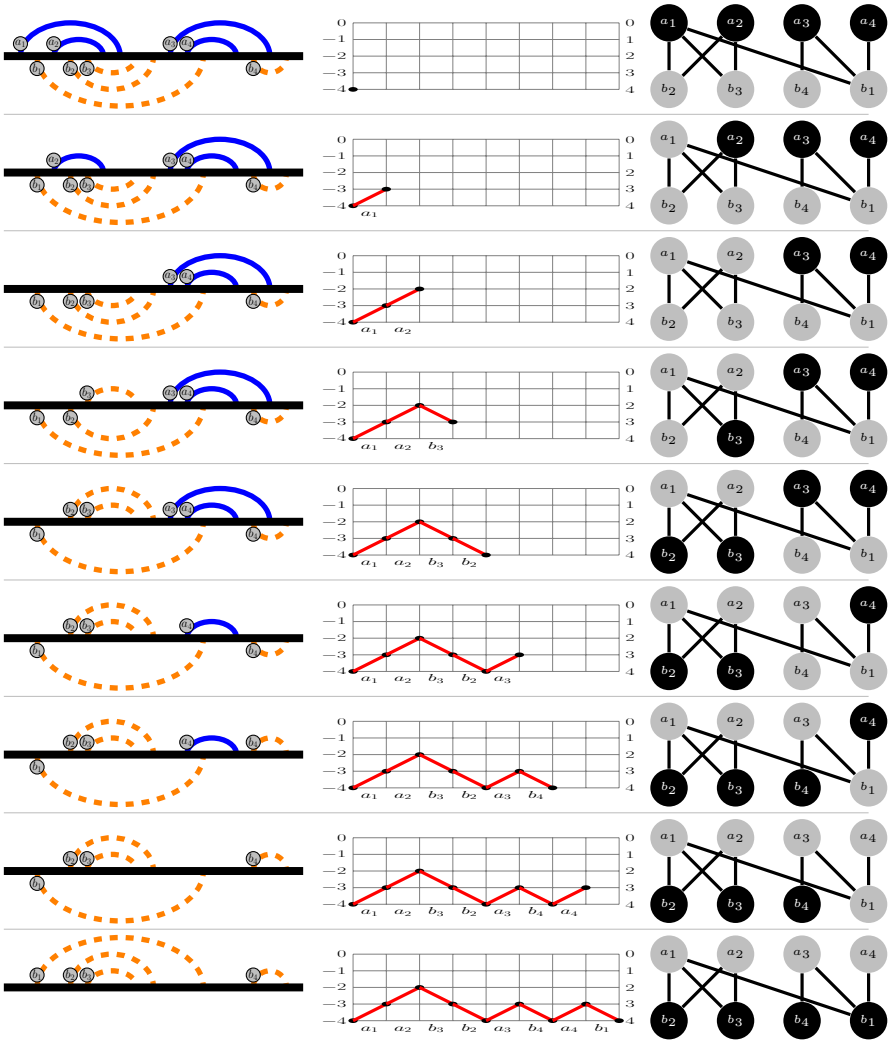


Figure 3.2: An example of a 2-barrier direct RNA folding pathway from an initial to final structure (left column), a corresponding set pathway (right column), and a graph showing the current folding pathway energy and the current barrier set size (center column). The set pathway instance (right) is specified by the conflict graph of the RNA folding pathway instance (left). The current set in the set pathway is denoted by **black** vertices while the current secondary structure in the folding pathway is indicated by the set of arcs on top.

Consider the second claim. Compare the set pathways that result from applying T_1 to G_1 and from applying T_1 to G. In the former, no set in the pathway contains an element of A_2, while in the latter all sets additionally contain A_2; otherwise, the set pathways are identical. Since barrier is relative to the initial set and all sets in the latter additionally contain A_2, then the barrier of both pathways is identical ($\leq k$). However, we must show that no element of A_2 conflicts with an element of $V(G_1) = A_1 \cup B_1$. Clearly this is true since $N_G(B_1) = A_1$, $A_1 \cap A_2 = \emptyset$ and A_1 is in the same partition. Next, consider that the final set after applying T_1 to G is $A_2 \cup B_1$. Importantly, note that $|A_2 \cup B_1| = |A|$ since G_1 is pairwise-optimal. To prove the claim, the remaining pathway can have barrier at most k. If T_2 is next applied, the remaining pathway is identical to the result of T_2 applied to G_2, except that each set will additionally contain B_1; as above, the barrier is identical ($\leq k$). Finally, no element of B_1 can conflict with any element of $V(G_2) = A_2 \cup B_2$ since $N_G(B_1) = A_1$, $A_1 \cap A_2 = \emptyset$ and B_2 is in the same partition. Therefore, T_1, T_2 must be a valid ($\leq k$)-barrier transformation sequence for G. □

Lemma 10. Let $G[A, B]$ be a pairwise-optimal bipartite graph and let $G' = G/[A', B']$ be pairwise-optimal where B' is a non-empty (not necessarily strict) subset of B and $A' = N_G(B')$. If the minimum barrier of any direct transformation sequence for G' is k, then the minimum barrier of any direct (possibly with repeats) transformation sequence for G is at least k.

Proof. Assume to the contrary that there exists a direct (possibly with repeats) transformation sequence T for G with barrier $k' < k$. Let X be the first set in the pathway specified by T which is missing k more elements from A' as from B'. Specifically, X is the first set such that $|X \cap B'| - |X \cap A'| = k$. Such a set must exist, otherwise G' would have a ($< k$)-barrier direct transformation sequence. We can determine the size of X relative to the initial set A, that is $|A \triangle X|$, as follows. Let $A_1 = (A \setminus X) \cap A'$ and $A_2 = (A \setminus X) \setminus A'$. Let $B_1 = (X \setminus A) \cap B'$ and let $B_2 = (X \setminus A) \setminus B'$. Informally, these are all elements removed ($A_1 \cup A_2$) and all elements added ($B_1 \cup B_2$), relative to the initial set A, partitioned by their inclusion ($A_1 \cup B_1$) or exclusion ($A_2 \cup B_2$) in $A' \cup B'$. Since T is a k'-barrier transformation sequence we have the following.

$$
\begin{aligned}
|A_1| + |A_2| - |B_1| - |B_2| &\leq k' && \\
k + |A_2| - |B_2| &\leq k' && \text{by definition of } X,\ |A_1| - |B_1| = k \\
k' + |A_2| - |B_2| &< k' && \text{since } k' < k \\
|A_2| &< |B_2| && (*)
\end{aligned}
$$

Case 1. $|B_2| = 0$. Contradiction with (*). Note that this case shows the claim holds when $G' = G$.

Case 2. $|B_2| > 0$. Consider that $N_G(B_2) \subseteq (A_1 \cup A_2) \subseteq (A' \cup A_2)$, otherwise X is not an independent set. Therefore, $N_G(B' \cup B_2) \subseteq A' \cup A_2$ since $N_G(B') \subseteq A'$

and we have that $B' \cup B_2 \cup (A \setminus (A' \cup A_2))$ is an independent set. Consider the size of this set.

$$\begin{aligned}
&|B' \cup B_2 \cup (A \setminus (A' \cup A_2))| && \\
&= |B'| + |B_2| + |A \setminus (A' \cup A_2)| && \text{mutually disjoint by definition} \\
&= |B'| + |B_2| + |A| - |A'| - |A_2| && \text{since } A' \cap A_2 = \emptyset \wedge (A' \cup A_2) \subseteq A \\
&= |A| + |B_2| - |A_2| && \text{since } |A'| = |B'| \text{ by assumption that} \\
& && G \text{ and } G/[A', B'] \text{ are pairwise-optimal} \\
&> |A| && \text{by (*)}
\end{aligned}$$

This contradicts that G is pairwise-optimal.

□

We are now faced with the task of identifying a pairwise-optimal subproblem. Intuitively, in terms of the folding pathway problem, this amounts to identifying a set of arcs from the final structure that can replace an equal number of arcs from the initial structure resulting in an intermediate structure that is (i) pseudoknot-free, and (ii) MFE. In terms of the set pathway problem, the resulting intermediate set is a maximum independent set; thus, our task is to identify a maximum independent set spanning both partitions. We can leverage Kőnig's Theorem to solve this problem in terms of matching.

Theorem 5 (Kőnig [61])**.** In a bipartite graph, the number of vertices in a maximum independent set equals the number of edges in a minimum edge covering.

Observation 1. If $G[A, B]$ is a pairwise-optimal bipartite graph, then G contains a perfect matching of size $|A|$ $(= |B|)$.

Proof. This follows immediately from the definition of pairwise-optimal and Theorem 5. □

For the sake of efficiency, it is desirable for a splitting strategy to always identify a minimal pairwise-optimal subproblem: one that cannot be split further. This is the behaviour of the `BasicSplit` algorithm that we now present.

Algorithm 1: `BasicSplit`

input : A non-null pairwise-optimal bipartite graph $G[A, B]$
output: (A_1, B_1) where $B_1 \subseteq B$, $A_1 = N_G(B_1)$, and $G/[A_1, B_1]$ is minimal pairwise-optimal

```
1 begin
2     M ⟵ MaximumMatching(G);
3     E′ ⟵ {(b, a) | b ∈ B ∧ (b, a) ∈ E} ∪ {(a, b) | a ∈ A ∧ (a, b) ∈ M};
4     (A_1, B_1), (A_2, B_2), ..., (A_p, B_p) ⟵ Tarjan(D = [A, B; E′]);
5     return (A_1, B_1);
```

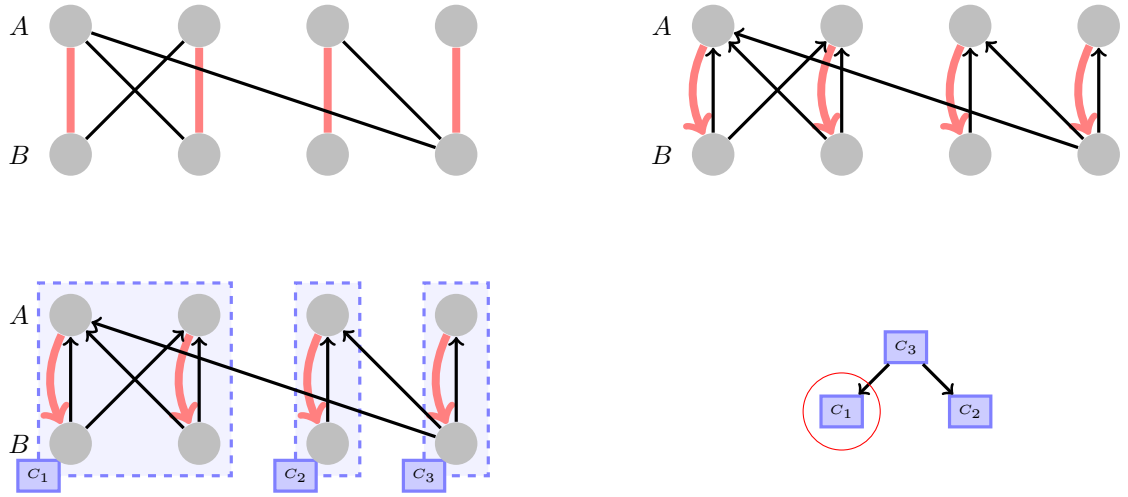


Figure 3.3: An example of the `BasicSplit` algorithm. For a pairwise-optimal bipartite graph G, a perfect matching is identified (top left), a directed *precedence* graph D is constructed (top right), strongly connected components in D are identified (bottom left), and one that is a sink in the condensation of D is returned (bottom right).

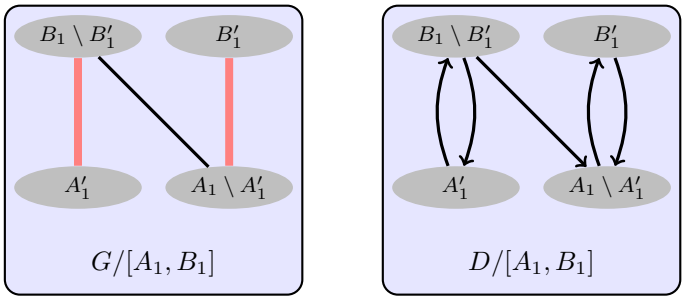
The algorithm itself is simple, and neatly summarized in Algorithm 1 and illustrated in Figure 3.3. First, find a maximum matching M in G (using for instance, the Hopcroft-Karp algorithm [52]). Second, create the *precedence graph* for G and M. By precedence graph, we mean the directed bipartite graph $D[A, B]$ where $E(D) = \{(b, a) \mid b \in B \wedge (b, a) \in E(G)\} \cup \{(a, b) \mid a \in A \wedge (a, b) \in M\}$. Third, find the strongly connected components (SSCs) of D (using for instance, Tarjan's algorithm [126]). Finally, return a SCC that is a sink in the *condensation* of D, *i.e.,* the directed acyclic graph in which each SCC is condensed into a single node. Note the Tarjan's algorithm, which we make use of in the `BasicSplit` algorithm, returns SSCs in depth-first order. Thus, the first SCC that is returned is a sink in the condensation of D.

The next lemma summarizes important properties of the `BasicSplit` algorithm.

Lemma 11. Given a pairwise-optimal bipartite graph $G[A, B]$ the `BasicSplit` algorithm returns a tuple (A_1, B_1) such that

1. $B_1 \subseteq B$,
2. $A_1 = N_G(B_1)$, and
3. $G/[A_1, B_1]$ is minimal pairwise-optimal.

Proof. Since the *precedence* graph D contains the same nodes as G then the first claim is trivially true. Consider the third claim. Since G is pairwise-optimal then by Observation 1, M must be a perfect matching. Consequently, each $b \in B_1$ is matched to a unique $a \in A$. By construction of the *precedence* graph D, a and b are strongly connected and therefore it must be the case that $a \in A_1$. Therefore, $|A_1| \geq |B_1|$. Since this is true for all p strongly connected components in D, specifically that $|A_i| \geq |B_i|$, for $1 \leq i \leq p$, and since $|A| = |B|$ (G is pairwise-optimal) we can conclude that $|A_1| = |B_1|$ by the pigeon hole principle. Therefore, there exists a perfect matching between A_1 and B_1 in $G/[A_1, B_1]$ and by Theorem 5, the size of a maximum independent set for $G/[A_1, B_1]$ is $|A_1|$ $(= |B_1|)$. Thus, $G/[A_1, B_1]$ is pairwise-optimal. Suppose it is not minimal pairwise-optimal. Then, there must exist some non-empty sets $A'_1 \subseteq A_1$ and $B'_1 \subseteq B_1$, such that $A'_1 \cup B'_1$ is also a maximum independent set in $G/[A_1, B_1]$.



With respect to G and M, A'_1 must be matched with $B_1 \setminus B'_1$ and B'_1 must be matched with $A_1 \setminus A'_1$. With respect to D, this implies that there are no arcs oriented from $A_1 \setminus A'_1$ to $B_1 \setminus B'_1$. Also, there cannot be any arcs from B'_1 to A'_1. Thus, for any $x \in B'_1 \cup (A_1 \setminus A'_1)$ there does not exist a path, in D, to any $y \in A'_1 \cup (B_1 \setminus B'_1)$ contradicting that (A_1, B_1) is strongly connected in D and proving the third claim.

Consider the second claim. Let $G' = G/[A_1, B_1]$. Since G' is pairwise-optimal, $A_1 = N_{G'}(B_1)$ and therefore $A_1 \subseteq N_G(B_1)$. If $A_1 = N_G(B_1)$ we are done. Otherwise, let a be any node in $N_G(B_1) \setminus A_1$ incident to some $b \in B_1$. Since $a \notin A_1$, a and b belong to different strongly connected components. Next, consider that Tarjan's algorithm finds SSCs of a graph by depth first search, returning a SSC only after all other reachable ones have been returned. This implies (A_1, B_1) must be a sink in the condensation of D. However, by its construction, D must contain an arc (b, a). Contradiction. □

3.2.2 Cutting strategy

In the original presentation of the algorithm [129], we presented two cutting strategies. Here, we detail the more intuitive (and efficient) of the two: the *two-sided cutting strategy*. For a problem instance consisting of a minimal pairwise-optimal bipartite graph $G[A, B]$ and barrier k, this strategy generates the subgraphs $G/(A \setminus \{a\}, B \setminus \{b\})$ for each choice of $a \in A$ and $b \in B$ and recursively solves each of the resulting subproblems with the barrier set to $k - 1$. The following lemma states that if we do this for all possible choices of a and b, we are guaranteed to find a $(\leq k)$-barrier set pathway for G if one exists.

Lemma 12. Let $G[A, B]$ be minimal pairwise-optimal. Then

1. $G/[A \setminus \{a\}, B \setminus \{b\}]$ is pairwise-optimal for all $a \in A$ and $b \in B$,
2. if $G/[A \setminus \{a\}, B \setminus \{b\}]$ has a transformation sequence T' with barrier $k-1$ then $T = \{a\}, T', \{b\}$ is a transformation sequence for G with barrier k and,
3. G has a transformation sequence with barrier k only if $G/[A \setminus \{a\}, B \setminus \{b\}]$ has a transformation sequence with barrier at most $k - 1$ for some $a \in A$ and $b \in B$.

Proof. Let $A' = A \setminus \{a\}$ and $B' = B \setminus \{b\}$ and let $G' = G/[A', B']$. Consider the first claim. Suppose not. Then, in G', there must exist some maximum independent set $C \subseteq A' \cup B'$, where $|C| > |B'|$ implying that C contains elements of both A and B. However, since $|B - B'| = 1$ we have that $|C| \geq |B|$ contradicting that G is minimal pairwise-optimal.

Consider the second claim. Let P' be the set pathway specified by T' on G'. Then the set pathway specified by T on G is exactly $P = A, P', B$. First, observe that P is a valid set pathway since (i) adding b last cannot introduce a conflict as the final set of P' is B' and (ii) removing a first ensures it cannot conflict with any set in P', B. Finally, since the barrier of P' is $k - 1$ relative

to A', it is $k - 1 + |A - A'| = k$ relative to A. Since adding b cannot raise the barrier, then T is a k-transformation sequence for G.

Consider the third claim. Suppose to the contrary that G has a k-transformation sequence and for all $a \in A$ and $b \in B$, any transformation sequence T' on G' has barrier at least k. But the barrier of T' is relative to A'. Relative to the initial set of G, the barrier is at least $k + |A - A'| = k + 1$. Since G' is pairwise-optimal (claim 1), then by Lemma 10 the barrier for G is at least as large as the barrier for G'. Contradiction. □

3.2.3 The overall algorithm

Algorithm 2: `DirectTransformation`

input : A pairwise-optimal bipartite graph $G[A, B]$ and a barrier k
output: A direct transformation sequence from A to B with barrier at most k, or $\emptyset$ if one does not exist

```
1  begin
       // trivial base case
2      if k ≤ 0 then return ∅;
3      ;
       // split
4      (A_1, B_1) ⟵ BasicSplit(G);
5      G_1 ⟵ G/[A_1, B_1];
6      G' ⟵ G/[A \ A_1, B \ B_1];
       // solve G' recursively
7      if G' is non-null then
8          T' ⟵ DirectTransformation(G', k);
9          if T' = ∅ then return ∅;
10         ;
11     else
12         T' ⟵ ∅;
       // base case for G_1
13     if |A_1| ≤ k then return A_1, B_1, T';
14     ;
       // otherwise, recursively solve G_1 with cutting strategy
15     foreach a ∈ A_1 and b ∈ B_1 do
16         T_1 ⟵ DirectTransformation(G_1/[A_1 \ {a}, B_1 \ {b}], k − 1);
17         if T_1 ≠ ∅ then
18             return {a}, T_1, {b}, T';
19     return ∅;
```

The `DirectTransformation` algorithm incorporates the splitting and two-

sided cutting strategy. First, a minimal pairwise-optimal subgraph G_1 is identified by the `BasicSplit` algorithm (line 3). Note that if G is minimal pairwise-optimal, then $G_1 = G$. If G', the remainder of the problem, is not null, *i.e.,* G was not already minimal pairwise-optimal, then it is solved recursively, if possible (lines 6-10). An overall solution is returned if G_1 is trivially solvable (line 11), otherwise, the cutting strategy reduces G_1 into smaller subproblems to be solved recursively (lines 12-15). Overall, if a solution for G_1 and G' is found, their concatenated pathway is returned as a solution to G. If no solution is found, then an empty transformation sequence is returned.

3.2.4 Algorithm correctness and complexity

Theorem 6. The `DirectTransformation` algorithm is correct.

Proof. To prove this claim we must show that for any arbitrary pairwise-optimal bipartite graph $G[A, B]$ the algorithm returns a valid $(\leq k)$-barrier direct transformation sequence for G, if and only if one exists. We prove this by induction on the order of G, *i.e.,* the number of vertices in G. Correctness is straightforward to show when $|A| = |B| = 1$. Suppose that `DirectTransformation` is correct on input graphs where $|A| = |B| \leq n - 1$.

By Lemma 11, G_1 is minimal pairwise-optimal and either G' is null or, by Lemma 9, G' is pairwise optimal. We will show correctness in the latter case which immediately implies correctness of the former. By Lemma 10 and Lemma 9 there exist $(\leq k)$-barrier transformation sequences T_1 and T', for G_1 and G', respectively, if and only if T_1, T' is a $(\leq k)$-barrier transformation sequence for G. Since the algorithm always returns the concatenation of their solutions, or $\emptyset$ if a solution for one does not exist, then it is sufficient to show the algorithm is correct for both G_1 and G'.

First, consider that G_1 cannot be null and consequently G' has a smaller order than G. Therefore, by assumption, `DirectTransformation` must be correct for G'. Next, consider how the algorithm solves G_1. The base case for G_1 (line 11) is clearly correct, so consider the recursive case (lines 12-15). Since G_1 is minimal pairwise-optimal then, by Lemma 12, G_1 has a solution if and only if there exists some $a \in A_1$ and $b \in B_1$ such that $G'_1 = G_1/[A_1 \setminus \{a\}, B_1 \setminus \{b\}]$ has a $(< k)$-barrier transformation sequence. Since the algorithm tries all pairs of a and b, and the algorithm is guaranteed to be correct for G'_1 by assumption (smaller order than G), then the recursive case must also be correct for G_1. □

Theorem 7. Given a pairwise-optimal bipartite graph $G[A, B]$ and maximum barrier k, `DirectTransformation` runs in $O(n^{2k+\omega})$ time and $O(n^2)$ space where $n = |A| = |B|$ and $\omega < 2.38$.

Proof. First note that the time and space complexity of the subsidiary algorithm `BasicSplit`, which is called once for each call to `DirectTransformation`, is dominated by finding a maximum matching; a problem that can be solved in $O(n^\omega)$ time and $O(n^2)$ space [47].

The worst case occurs when the input for each recursive call to `DirectTransformation` is already minimal pairwise-optimal; thus, `BasicSplit` simply returns the original input. In this case, the cutting strategy makes $O(n^2)$ recursive calls to `DirectTransformation` with allowable barrier one less than the current, and the recursion bottoms out when k reaches 0. We therefore have $O(n^{2k})$ calls to `DirectTransformation` each taking $O(n^\omega)$ time, since `BasicSplit` dominates the runtime at each step, for a total runtime of $(n^{2k+\omega})$. In all cases, the maximum matching algorithm dominates space usage resulting in $O(n^2)$ space overall. □

Comments on practical and theoretical runtime efficiency

The observant reader will have noticed a potential redundancy in the `DirectTransformation` algorithm. Specifically, that the sequence of SCCs returned by Tarjan's algorithm specifies a *safe splitting sequence*: a sequence of minimal pairwise-optimal subgraphs $G_1, G_2, \ldots, G_p$ such that a concatenation of optimal transformation sequences for these problems $T = T_1, T_2, \ldots, T_p$, where T_i is an optimal transformation sequence for G_i, is an optimal solution for the original graph G. (Thus, the redundancy arises as `BasicSplit` is called an extra $p - 2$ times to determine the same splitting sequence.) The correctness of this claim follows from a straightforward generalization of Lemma 9 and Lemma 11. In general, the condensation of the precedence graph specifies a partial order on minimal pairwise-optimal subgraphs. Any total order respecting this partial order (for instance, a depth first traversal), is a safe splitting sequence. Considering the example in Figure 3.3, C_1, C_2, C_3 and C_2, C_1, C_3 are both safe splitting sequences. Interestingly, the condensation of the precedence graph provides a succinct representation of all maximum independent sets. In terms of the folding pathway problem, this results in a succinct representation of all MFE structures, composed of arcs from the initial and final structures.

Next, consider the cutting strategy which must guess (through brute force enumeration) a first element to remove and a last element to add. If this reduced problem cannot be split, the procedure must be repeated recursively until it can (or a base case is reached). If G is a bi-clique, *i.e.,* all elements from the A must be removed before any elements from B can be added[12], then we witness the theoretical worst case behaviour of the algorithm. However, if we instead ask which element $b \in B$ we will first add, and which element $a \in A$ we will last remove, then we can fully determine the sequence of elements which must be removed first ($N_G(b)$), and which sequence of elements will be added last ($N_G(a)$). This seemingly minor observation leads to a major practical speed-up. Similarly, we need only consider pairs a and b which are not adjacent in G. For dense graphs (*i.e.,* $|E(G)| = \Theta(|A|^2)$), this can significantly cut the search space in practice. In theory, we may argue that the algorithm runtime is $O(n^{k+w})$ since there are $O(n)$ non-adjacent pairs in a dense bipartite graph.

[12]Recall the definition of a *band* of arcs in the folding pathway problem from Chapter 2. A bi-clique in the set pathway problem corresponds to a folding pathway instance containing exactly two bands, one for each structure, that cross.

The worst case analysis assumed the use of the most efficient algorithm known (in terms of worst case runtime) for finding a maximum matching [47]. However, this can be improved both in terms of practical and theoretical efficiency if the conflict graphs are known to be sparse. In this case, the matching algorithm due to Hopcroft and Karp [52] guarantees $O(m\sqrt{n})$ runtime, where m and n are the size (number of edges) and order (number of vertices) of G, respectively, and is known to be one of the most efficient in practice. For this reason, we have implemented the Hopcroft and Karp algorithm for finding a maximum matching within our `BasicSplit` algorithm. Therefore, the implementation we evaluate in Section 3.3 has runtime $O(n^{2k+2.5})$. If we know our conflict graphs are sparse (*i.e.*, $|E(G)| = O(|A|)$), we may argue that the algorithm runtime is $O(n^{2k+1.5})$.

Finally, consider that this algorithm has a trivial parallel implementation: each subproblem in a safe splitting sequence can be solved independently, in parallel; furthermore, each subproblem generated by the cutting strategy can be solved independently, in parallel.

3.2.5 Finding minimum barriers for non-pairwise optimal instances

The above algorithm maintains the invariant of operating on pairwise-optimal bipartite graphs. While this property has greatly simplified the algorithm description and proof of correctness, we now outline how these results can be extended to the more general case of solving the barrier problem for any bipartite graph $G[A, B]$. We accomplish this by giving a polynomial time reduction from G to a pairwise-optimal bipartite supergraph of G, denoted as $\texttt{PWO}(G)$. As we will show, a solution for $\texttt{PWO}(G)$ can be mapped to solution for G.

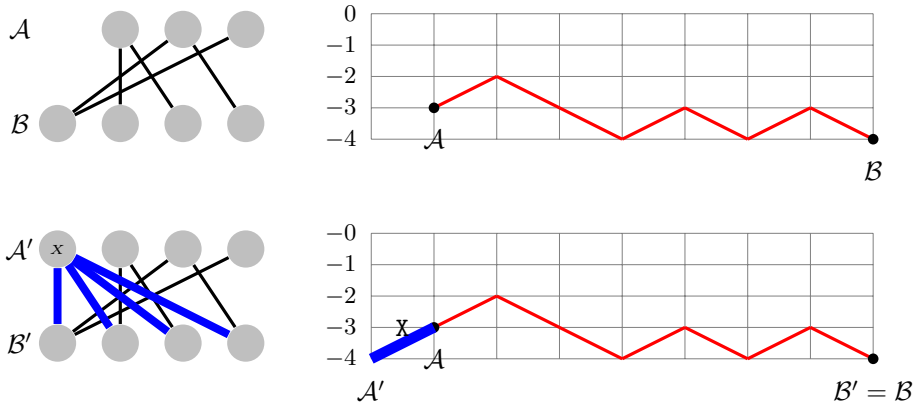


Figure 3.4: Creating a pairwise-optimal instance (bottom) from a non-pairwise-optimal instance (top).

Construction of $\texttt{PWO}(G)$

If G is not pairwise-optimal, then $\alpha(G) > |A|$ and/or $\alpha(G) > |B|$. We will construct a bipartite supergraph $G'[A', B']$ of G such that $\alpha(G') = |A'| = |B'| = \alpha(G)$ and is therefore pairwise-optimal. Let $A' = A \cup X$ and $B' = B \cup Y$ where $|X| = \alpha(G) - |A|$ and $|Y| = \alpha(G) - |B|$. Finally, let $E(G') = E(G) \cup E'$ where $E' = \{(x, b) \,|\, x \in X \wedge b \in B'\} \cup \{(a, y) \,|\, y \in Y \wedge a \in A'\}$. Note that since G is bipartite, $\alpha(G)$ can be determined in polynomial time [47, 52, 61] and therefore $\texttt{PWO}(G)$ can be constructed in polynomial time ($O(n^\omega)$). See Figure 3.4 for an example.

Theorem 8. There exists a $(\leq k)$-barrier transformation sequence for any bipartite graph $G[A, B]$ if and only if there exists a $(\leq k')$-barrier transformation sequence for $\texttt{PWO}(G)$, where $k' = k + \alpha(G) - |A|$.

Proof. Let $G'[A', B'] = \texttt{PWO}(G)$, where $A' = A \cup X$ and $B' = B \cup Y$. Let T' be a transformation sequence for G' having barrier k'. Since G' is a supergraph of G, then T' must contain as a subsequence a transformation sequence T for G. Let k be the barrier of T applied to G. Consider that no addition operation can appear in T' until all elements from X have been removed since every element of X dominates B'. Likewise, no element of Y' can be added in T' until all of A' is removed. Therefore, since reordering a consecutive sequence of remove (or addition) operations cannot affect the barrier, T' can be reordered as X, T, Y. Since the barrier for the prefix X, T is at most k' relative to the initial set A', and since the prefix X simply removes all elements of X thus resulting in the initial set for G, then we can express the barrier for X, T as $|X| + k = k + \alpha(G) - |A|$. Since the addition of elements of Y at the end of the transformation sequence cannot increase the barrier, we can conclude that $k' = k + \alpha(G) - |A|$. □

3.3 Empirical results

We implemented two versions of our algorithm for the direct set barrier problem, differing by their cutting strategy, in order to study their efficiency in practice on biologically motivated data. The first, referred to as the $O(n^{2k+2.5})$ algorithm, uses the two-sided cutting strategy as described above. The second, referred to as the $n^{O(n)}$ algorithm, uses a one-sided cutting strategy described in previous work [129]. Until this point, our algorithm has been described in terms of a decision problem; *i.e.,* can a problem instance be solved within barrier k? However, the implementation of our algorithm is terms of the more general optimization problem; *i.e.,* find the minimum barrier k that can solve a problem instance. The general strategy is to perform a binary search on values of k, using the decision algorithm as a subsidiary algorithm. As such, our empirical results report on the runtime required to identify the minimum barrier.

3.3.1 Implementation and experimental environment

Both algorithms were coded in C++ and compiled using g++ (GCC version 4.2.1). All experiments were run on our reference PCs with 2.4Ghz Intel Pentium IV processors with 256KB L2 cache and 1GB RAM, running SUSE Linux version 10.3.

3.3.2 Generation of problem instances

With the motivation of studying algorithm performance across a variety of problem instances, we randomly sampled five sequences for each of four different classes of non-coding RNA—*Transfer RNA, Transfer Messenger RNA, Ribonuclease P RNA,* and *5S Ribosomal RNA*—found in the RNA STRAND database [3]. For each sequence, five MFE structures—with respect to number of base pairs—were determined using a modified version of the Nussinov-Jacobsen algorithm [89]. The modified algorithm stored all optimal paths within the traceback matrix. In this way, we were able to randomly sample five different MFE structures for the same sequence. Identical structures were discarded. Every possible pairing of structures for the same sequence formed a new problem instance. Thus, ten problem instances were created for each sequence, resulting in 200 problem instances overall. The distribution of sequence length and the resulting number of conflicting base pairs between paired structures can be seen in Figure 3.5. In general, and as expected, the number of conflicting bases pairs increases with sequence length.

3.3.3 Algorithm runtime performance

Both algorithms were run for a maximum of 1 CPU hour on each of the 200 hundred problem instances. The $n^{O(n)}$ algorithm found solutions to 183 instances, while the $O(n^{2k+2.5})$ algorithm found solutions to 184 instances. Interestingly, the $n^{O(n)}$ algorithm found solutions to three instances not found by the $O(n^{2k+2.5})$ algorithm; likewise, four instances were found by the $O(n^{2k+2.5})$ algorithm not found by the $n^{O(n)}$ algorithm. Of the instances that were solved, optimal barriers were found within 1 CPU second by both algorithms in 90% of the cases with barrier height ranging from 1 to 8. The barrier of harder instances ranged from 6 to 11, with a mean of 9. In general, the $O(n^{2k+2.5})$ algorithm was the best performing for harder instances. However, as can be seen in Figure 3.6, both algorithms excelled for certain instances relative to one another.

The instances which failed to be solved within our cut-off time tended to have the highest number of conflicting base pairs; moreover, they tended to have the largest minimally pairwise-optimal subproblems generated by the `BasicSplit` algorithm. For each problem instance, we recorded the size of the maximum subproblem, as well as the average size of all subproblems, produced by the `BasicSplit` algorithm at the top level of recursion. We measured size as number of base pairs (elements in terms of the set barrier problem). Figure 3.7

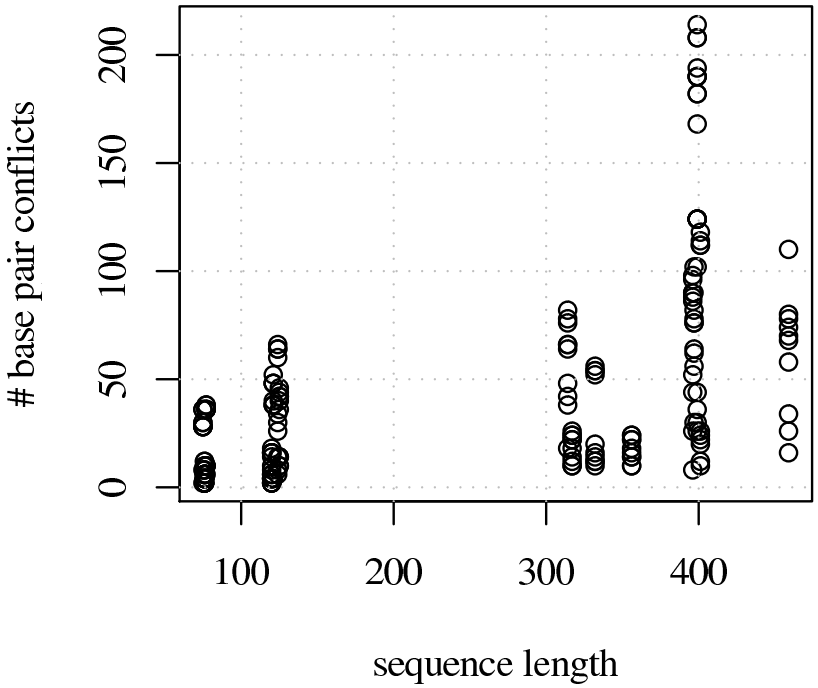


Figure 3.5: Distribution of conflicting base pairs for generated problem instances.

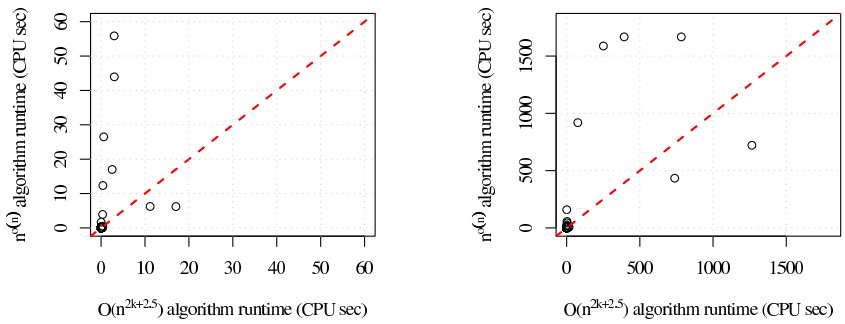


Figure 3.6: The required time to find an optimal barrier pathway is shown for two time scales.

shows the frequency of problem instances which have given maximum (left) and average (right) subproblem size. The problem instances which have maximum subproblems of size 200 or more were those that failed to be solved, within the allotted runtime. Alternative methods for splitting or recursing on such subproblems would clearly be valuable. Or, simply exploiting the inherent parallelism of the proposed algorithms could lead to a solution in reasonable wall-clock time, when many CPUs are employed.

3.4 Solving the direct with repeats barrier problem

Since any graph is a subgraph of itself, and since Lemma 10 correctly considers this case, we can immediately conclude that repeat operations cannot lower the barrier in direct transformation sequences for pairwise-optimal instances. Moreover, as an immediate consequence of Lemma 10 and Theorem 8, we can conclude the following, more general result.

Theorem 9. If there exists a direct-with-repeats k-barrier transformation sequence for a bipartite graph $G[A, B]$ then there must exist a direct transformation sequence for G with barrier at most k.

Theorem 9 has implications for the direct-with-repeats folding pathway problem that we formally define below.

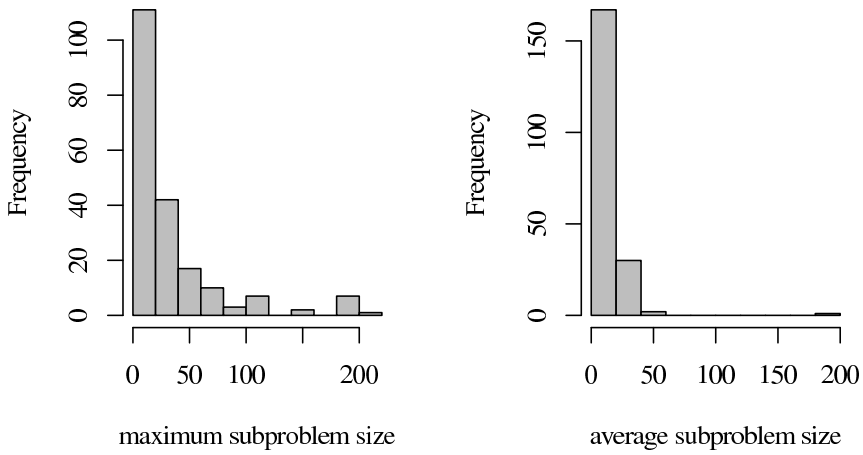


Figure 3.7: Frequency of maximum (left) and average (right) subproblem sizes, measured as number of base pairs in the subproblem produced by the first call to the `BasicSplit` algorithm for a given instance. The maximum and average are taken over all subproblems generated for a given instance.

Problem 5. EB-DRPFP (Energy Barrier for Direct-with-Repeats Pseudoknot-free Folding Pathway of a single strand)
Instance: Given two pseudoknot-free configurations $\mathcal{I} = \{I_i\}_{i=1}^{n}$ (initial) and $\mathcal{F} = \{F_i\}_{i=1}^{m}$ (final), of a single strand, and integer k.
Question: Is there a direct-with-repeats pseudoknot-free transformation sequence S such that the energy barrier of S, in the simple energy model, is at most k?

As repeats do not lower the energy barrier, then, by combining Theorem 3 and Theorem 9, we can conclude the following result.

Theorem 10. The EB-DRPFP problem, namely the energy barrier for direct-with-repeats pseudoknot-free folding pathway problem, of a single strand, is NP-complete.

We can also consider the implications for the multi-strand version of the problem that allows repeats in the folding pathway.

Problem 6. EB-DRPFP-MULTI (Energy Barrier for Direct-with-Repeats Pseudoknot-free Folding Pathway of Multiple interacting strands)
Instance: Given two pseudoknot-free configurations $\mathcal{I} = \{I_i\}_{i=1}^{n}$ (initial) and $\mathcal{F} = \{F_i\}_{i=1}^{m}$ (final), of multiple interacting strands, and integer k.
Question: Is there a direct pseudoknot-free transformation sequence S such that the energy barrier of S, in the simple energy model, is at most k?

It is unclear whether Theorem 9 can be applied to the multi-strand case. It leaves open the possibility that repeats may help to lower the energy barrier. Furthermore, we currently cannot bound the length of a minimum energy barrier pathway for this problem. Thus, we do not know if EB-DRPFP-MULTI is in NP. However, by restriction to the single strand case, we can conclude the following result.

Theorem 11. The EB-DRPFP-MULTI problem, namely the energy barrier for direct-with-repeats pseudoknot-free folding pathway of multiple interacting strands problem, is NP-hard.

We note that a constructive proof of Theorem 9 was previously given in terms of the more restricted folding pathway problem [129]. It is likely the same proof would hold for the more general set pathway problem as well. This alternative proof was given instead as it was noticed that a slight generalization of Lemma 10 proved this property in addition to the properties required to show algorithm correctness.

3.5 Chapter summary

In this chapter, we proposed an algorithm to exactly solve a generalized version of the direct energy barrier folding pathway problem that is defined in terms of

bipartite graphs. The algorithm has an exponential worst case time complexity, but uses only polynomial space. Because the algorithm is inherently parallel, this property could be exploited to help solve hard instances. As shown by an empirical study, the algorithm is practical for most instances, although it fails to solve some instances in a reasonable run-time. Due to the splitting algorithm, large sequences are not necessarily hard to solve. However, sequences that that do not decompose into small sub-problems can result in poor empirical performance by the algorithm, including the failure to solve the instance within a reasonable runtime. This is, of course, due to the exhaustive enumeration performed in order to guarantee a solution is optimal. However, the splitting algorithm could be used in conjunction with heuristic methods. For instance, it could be used to first partition the solution space into sub-problems, with the aim of improving both the efficiency and accuracy of the overall heuristic method used to solve each sub-problem.

We note that our algorithm is based on the *simple energy model*; however, there is potential to extend it to more complicated energy models. For instance, it may be straightforward to extend the algorithm to consider nearest neighbour base pairs as in the *Turner energy model* [79]. The resulting changes would still only consider *local* interactions when calculating the energy of particular structures. However, it seems unlikely that the current algorithm can be easily modified to consider *global* interactions found in the full Turner energy model, such as multi-branch loops. Fortunately, for the case of folding pathways resulting from strand displacement systems, such global interactions are not present by design. Furthermore, in the design of strand displacement systems, it is desirable that intended folding pathways have *low* energy barriers. Therefore, if our algorithm were extended to the multi-strand case (as discussed in Section 6.1), it would have a polynomial runtime in these important cases. As such, it may be a valuable tool in the design of strand displacement systems where unintended folding pathways should have a necessarily larger barrier than intended pathways.

Interestingly, by proving the algorithm is correct, we were also able to prove that repeat arcs do not help in a direct folding pathway. This establishes that the direct-with-repeats energy barrier folding pathway problem is NP-complete for the single strand case and NP-hard for the multiple interacting strand case.

Unfortunately, the algorithm does not seem applicable for indirect folding pathways. The design of the algorithm explicitly assumes that the graph modeling the conflicts, between the arcs of the initial and final structures of a problem instance, is bipartite. In an indirect folding pathway, where any non-crossing arc forming a Watson-Crick base-pair can be added at any point along a pathway, the conflict graph is not necessarily bipartite (and unlikely to be in general). Still, it is possible that a better understanding of the structure of conflict graphs for indirect pathways could lead to a similar result. The conflict graphs formed for indirect folding pathways can be characterized as circle graphs. The conflict graphs for direct pathways are 2-colourable circle graphs (see Figure 3.8 for an example).

For direct pathways, we were able to exploit the following property: if one

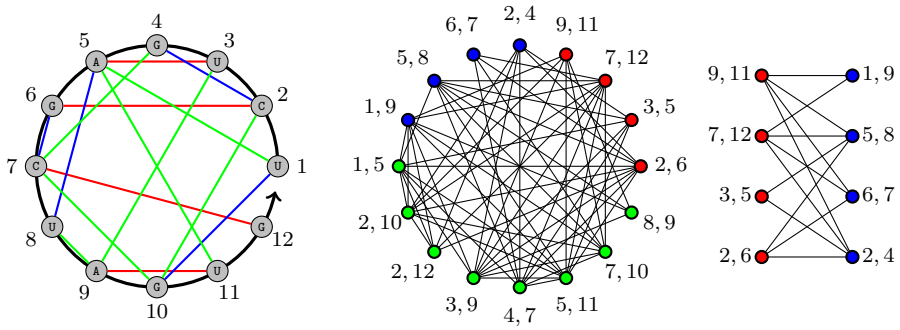


Figure 3.8: (left) An arc diagram representation for the RNA strand `UCUGAG CUAGUG`. Arcs (base pairs) in the initial structure are shown in red, those in the final structure are shown in blue and potential temporary arcs are shown in green. Also shown are the corresponding conflict graphs for the indirect folding pathway problem (center) and the direct folding pathway problem (right).

could identify an MFE structure C consisting of arcs from both the initial and final structures, A and B respectively, then there always exists an optimal pathway from A to B via C. We note that finding maximum independent sets in circle graphs, which correspond to MFE structures, is in P[13]. Could a generalized version of the algorithm proposed here be adapted for indirect pathways? Most properties exploited in the proofs are argued in terms of independent sets. Removing assumptions regarding the colourability of the graph would be a necessary first step. Finally, it remains possible that the indirect folding pathway problem (for a single strand[14]) is in P.

[13] While it is the case that the pseudoknot-free RNA/DNA structure prediction problem is in P, it is important to separately note the complexity for the corresponding circle graph problem, since the former is a restriction of the latter.

[14] We show in Chapter 5 that the indirect pathway problem is PSPACE-complete for multiple interacting strands.

Chapter 4

On recycling and its limits in molecular programs

While the previous two chapters focused exclusively on *finding* minimum-energy barrier folding pathways, we now turn our attention towards *designing* folding pathways to perform space efficient deterministic computation. Specifically, our aim is to design a low energy barrier folding pathway that deterministically transitions through a number of unique structures exponential in the length of the nucleic acid strand(s). We do not know how to design such pathways with a single strand, but in this chapter we show how this goal can be achieved using a set of multiple interacting strands. In such a pathway, subsets of strands will bind and unbind to other subsets, forming and breaking new strand complexes, multiple times over the length of the pathway. Thus, various strands will be actively reused, or *recycled* during the course of the folding pathway. This chapter explores the limits of strand recycling in folding pathways and the molecular programs that leverage them. More generally, we also consider the concept of recycling molecules within *Chemical Reaction Networks (CRNs)*. To our knowledge, we present the first example of a *DNA Strand Displacement system (DSD)* which significantly recycles strands. This also serves as the first example of a designed minimum energy barrier (indirect) folding pathway whose length is exponential in the combined length of participating strands. We also demonstrate a serious limit to recycling: recycling is not possible in deterministic CRNs, and their DSD realizations, when multiple copies of the initial state of the system are present in the same environment. In fact, we show that with just one extra copy of the initial signal molecules of a given CRN, it can perform at most a linear number of deterministic computation steps within a reaction volume of a closed system.

4.1 Introduction

We begin the chapter by illustrating the concept of recycling within molecular programs, discuss the benefits and possible limits, and give an overview of related work and our results in this context. The molecular programming models and terminology used throughout this and subsequent chapters are reviewed and

Content from this chapter appears in the proceedings of the 17th Annual International Conference on DNA computing and molecular programming (DNA 2011) [25] and the Journal of the Royal Society: Interface Focus [26].

introduced in Section 1.2 of the introductory chapter. Moreover, definitions of common molecular programming terms can be found in the Glossary.

4.1.1 On the need for strand recycling

Our goal is to determine whether or not molecular programs that leverage folding pathways can perform deterministic computations that are also space efficient. If a molecular program consists of $\Theta(n)$ molecules, could it deterministically advance through $\Theta(2^n)$ unique states? To answer this question, we initially set out to design a molecular program that simulates an n-bit standard binary counter. Recall the 3-bit standard binary counter given in Example 1.2.1 of Chapter 1. The chemical reaction equations for the counter are given in Figure 1.6(a). The counter is designed to begin at count 000, advance to 001, and so on, until reaching the count 111. Indeed, the corresponding CRN, with initial signal multiset $\{0_3, 0_2, 0_1\}$, simulates a logically reversible computation advancing correctly through the 2^3 unique states as illustrated in Figure 1.6(b). This counter can be generalized to n bits in the obvious way. It would seem that this simple example is sufficient to show that chemical reaction networks can perform deterministic computation exponential in their size (their largest signal multiset size, or number of reactions, for instance). Can we implement this CRN with a DSD?

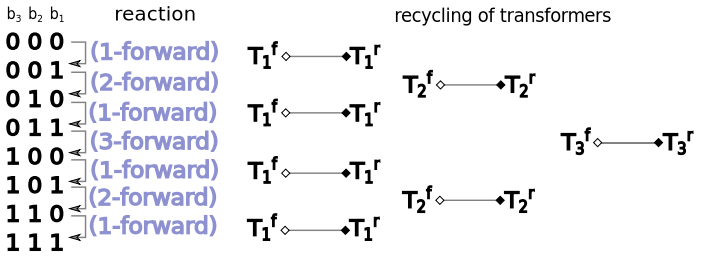


Figure 4.1: To reach the end state, the standard binary counter must perform a sequence of reactions that always occur in the forward direction, thus requiring a new transformer for every reaction as they are not recycled.

As discussed in Section 1.2.3, we do not know how to implement chemical reaction equations in a DSD without the use of transformers. (An example of a transformer implementing the chemical reaction equation $0_1 \rightleftharpoons 1_1$ is given in Figure 1.8.) While a transformer implementing a particular chemical reaction equation can be used to effect both the forward and reverse of the reaction, it must strictly alternate between these directions. To capture this notion, we introduced the concept of tagged chemical reaction equations and formally defined *tagged Chemical Reaction Networks (tagged CRNs)* in Section 1.2.3. The tagged equations for the 3-bit counter are given in Figure 1.9. Consider that the standard binary counter always performs these reactions in one direction. This means that a new transformer is required *at every reaction step*. This

is illustrated in Figure 4.1. While the generalized n-bit counter does advance deterministically through 2^n states, it would also require $2^n - 1$ transformers be present in the initial tag multiset when formally defined as a tagged CRN. The required space of a tagged CRN for an n bit standard binary counter is $\Theta(2^n)$. In general, the standard binary counter is *not* an example of space efficient deterministic computation that can be realized by a DSD.

The lack of transformer reuse in the standard binary counter is representative of other DSD programs in the literature. While some do use reversible transformers, such as the example transformer of Figure 1.8, the intended computation does not actively exploit this property. Appropriately, transformers are often referred to as *fuel*. The term captures the problem well: should the same reaction need to occur multiple times in the future, additional copies of fuel are required. In a reaction volume of a closed system, all fuel necessary to complete a computation must be present initially to avoid *fuel-depletion*. Therefore, the reaction volume becomes polluted with inactive fuel strands referred to as waste. *Active* recycling of transformers could avoid these problems.

4.1.2 On the potential for strand recycling

With the aim to avoid fuel depletion, waste, and to give an example of a space-efficient DSD, we now propose an alternate counter based on the binary reflecting Gray code sequence [111]. The sequence is a *Gray code* as each successive value differs from the previous in exactly one bit position. It is called a binary *reflecting* Gray code due to its elegant recursive definition: the n-bit Gray code sequence is formed by reflecting the $(n-1)$-bit sequence across a line, then prefixing values above the line with 0 and those below the line with 1. This is illustrated for $n = 1, 2, 3$ in Figure 4.2.

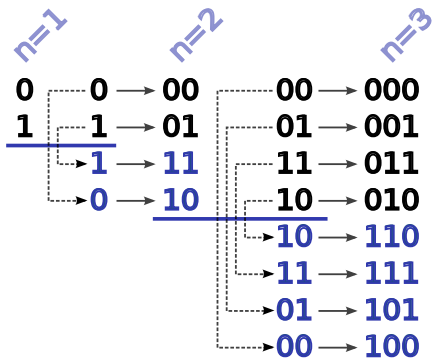
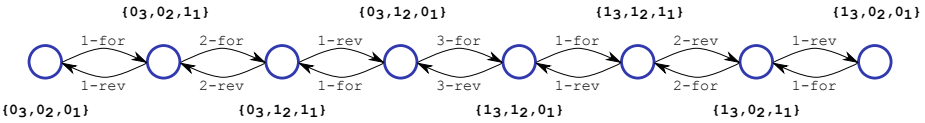


Figure 4.2: The 3-bit binary reflecting Gray code. The code for n digits can be formed by *reflecting* the code for $n-1$ digits across a line, then prefixing each value above the line with 0 and those below the line with 1.

Figure 4.3(a) gives the tagged chemical reactions for 3-bit version of this counter, which we call GRAY. The counter advances through application of the

$$
\begin{array}{lcc}
(1) & \mathrm{T}^{\mathrm{f}}_{\mathbf{1}} + 0_1 & \rightleftharpoons & \mathrm{T}^{\mathrm{r}}_{\mathbf{1}} + 1_1 \\
(2) & \mathrm{T}^{\mathrm{f}}_{\mathbf{2}} + 0_2 & \overset{1_1}{\rightleftharpoons} & \mathrm{T}^{\mathrm{r}}_{\mathbf{2}} + 1_2 \\
(3) & \mathrm{T}^{\mathrm{f}}_{\mathbf{3}} + 0_3 & \overset{1_2+0_1}{\rightleftharpoons} & \mathrm{T}^{\mathrm{r}}_{\mathbf{3}} + 1_3
\end{array}
$$

(a)



(b)

Figure 4.3: (a) Tagged chemical reaction equations for a 3-bit binary reflecting Gray code counter. (b) The configuration graph of the computation performed by the 3-bit binary reflecting Gray code counter forms a chain and is logically reversible. The nodes represent the state of the computation and the edges are directed between states reachable by a single reaction.

three reversible tagged chemical reaction equations (1-3) to produce the logically reversible computation chain shown in Figure 4.3(b). It is worth pointing out the correspondence between the tagged CRN and the recursive definition of the binary reflecting Gray code sequence. Consider the chain of Figure 4.3(b). The middle reaction (`3-for`) flips the third bit from a 0 to a 1 for the first and only time. To its left is the complete chain for the 2-bit sequence and to its right is the complete chain for the reverse of the 2-bit sequence. This can be seen in the left four chain nodes if one ignores the 0_3 bit in the signal sets. To create the 4-bit code, one additional reaction can be added ($\mathrm{T}^{\mathrm{f}}_{\mathbf{4}} + 0_4 \overset{1_3+0_2+0_1}{\rightleftharpoons} \mathrm{T}^{\mathrm{r}}_{\mathbf{4}} + 1_4$) which effectively results in the entire chain of Figure 4.3 forming the left half of the computation chain of the 4-bit sequence, and its reverse forming the right half, separated by a single reaction to flip the fourth bit.

The key idea to achieving this reaction sequence is for the reaction that alters the n^{th} bit to require as catalysts the last signal values in the $(n-1)$-bit sequence. This ensures the new reaction does not proceed until the $(n-1)$-bit sequence is complete. Moreover, by not consuming any of these signals, it forces the entire computation chain, until that point, to reverse. The reason for this is as follows. The new reaction has created a new signal, not previously seen (1_n). This creates a new state along the computation chain. Reversing the new reaction simply steps back in the computation chain; however, since the new state contains, as a proper subset, all signals at the end of the chain for the $(n-1)$-bit sequence, then the last reaction of that chain can proceed *in reverse*. Once this occurs, the reaction for the n^{th} bit cannot proceed in either direction as the necessary catalysts are no longer present. Moreover, since the $(n-1)$-bit sequence formed a logically reversible chain, then all of its reactions will be reversed. This is recursively true for its left and its right sub-chain. This

is a powerful technique that is also exploited in our designs of Chapter 5.

The key difference between the GRAY counter and the standard binary counter is that each particular reaction in GRAY occurs alternately in the forward and reverse direction, due to the recursive nature of the computation chain. This is illustrated in Figure 4.4. GRAY only requires a single copy of each transformer. For a 3-bit GRAY counter, the initial signal multiset is $\{0_3, 0_2, 0_1\}$ and the initial tag multiset is $\{\mathrm{T}_3^{\mathrm{f}}, \mathrm{T}_2^{\mathrm{f}}, \mathrm{T}_1^{\mathrm{f}}\}$. An n-bit GRAY counter has an initial signal multiset of size n and an initial tag multiset of size n. As GRAY is also a proper CRN, then by Lemma 2 it has space complexity $2n$. In general, GRAY *is* an example of a space-efficient deterministic computation that can be realized as a DSD.

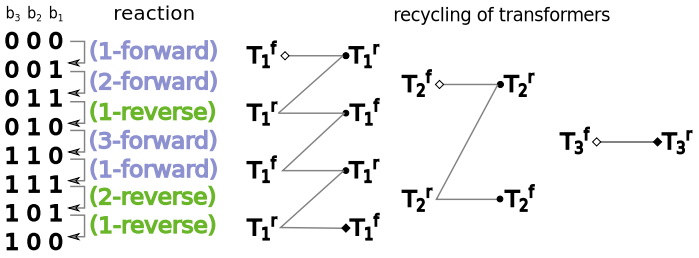


Figure 4.4: To reach the end state, the binary reflecting Gray code counter must perform a sequence of reactions that always alternate in the forward and reverse direction, thus requiring only one transformer for every reaction since they are actively recycled.

In summary, recycling in DNA strand displacement systems offers the potential of supporting space-efficient DNA computations in which the number of strands required to complete a computation in a reaction volume of a closed system is logarithmic in the length of the computation. Systems that recycle strands, and more generally molecules, do not use fuel, *i.e.,* large concentrations of certain transformer species that can bias reactions in one direction, and so are not prone to problems of fuel-depletion or waste. However, such advantages come at a price: as we will show, our counter proceeds somewhat more slowly than comparable fuel-driven strand displacement counters. The slowdown is due in part to the fact that reactions are used in both directions. Thus, our GRAY counter is not biased to advance towards the final state but rather performs an unbiased random walk on the logically reversible computation chain, both advancing and retreating, ultimately reaching the final state.

4.1.3 On the limits of strand recycling

Our n-bit GRAY counter advances correctly through 2^n states because only single copies of initial signals are present. That is to say, the computation relies on the fact that certain signal molecules are *absent* during certain steps in the computation. This property cannot be guaranteed, for instance, if the

initial signal multiset were duplicated multiple times within the same reaction volume. Consider a reaction that consumes a signal present in the initial signal multiset. It is not necessarily true that the same reaction will next be repeated, multiple times, in order to consume all of the additional copies present due to the duplication of the initial signal multiset. As we will show, ensuring that a signal is consumed in all copies present in the same reaction volume is not generally possible.

In Section 4.3 we show that if $\Theta(n)$ copies of the initial signal multiset are present, then the counter does not advance properly in a very strong sense: the final state of the counter can be reached in just $O(n^2)$ chemical reactions, rather than using the intended sequence of 2^n reactions. This result applies more generally and shows limits on molecule recycling when multiple copies of the initial signal multiset are present, under some restrictions on the allowable CRNs. In particular, if the size of the initial signal multiset of such a CRN is logarithmic in the length of a valid computation, then the CRN can produce any signal in a polynomial number of steps, when a linear number of copies of the initial signal multiset are present. We give a stronger result for tagged CRNs by showing that they cannot perform a computation super-linear in the size of their initial tag multiset when two copies of the initial signal and tag multisets are present in the same reaction volume.

4.1.4 Related work

Qian et al. [98] showed how to simulate a stack machine using strand displacement systems. A binary counter can be implemented via a stack machine; we call such a counter a QSW (Qian-Soloviechik-Winfree) counter and we compare its properties and resources with our counters in Section 4.2.5. Their result performs logically reversible computation and can also use fuel to bias the computation toward the final state. We compare our results to a fuel-biased QSW counter as the unbiased version is slower—it performs an unbiased random walk along the computation chain similar to our result. We also assume that all fuel, or transformers, must be initially present in the reaction volume that we assume is a closed system.

Building on models of Winfree and Rothemund [106, 141], Reif et al. [102] studied a tile-based graph assembly model in which tiles may both adhere to and be removed from a tile assembly. In their self-destructible graph assembly model, the removal of tiles allows for the possibility of tile reuse. The authors demonstrate that tile reuse is possible in an abstract tile model, via a PSPACE-hardness result. Doty et al. [30] showed a negative result on tile reuse for an irreversible variant of the model of Reif et al.

Kharam et al. [60] describe a DNA binary counter in which bit values are represented using relative concentrations of two molecule species. This is very different than our work, where the values of bits (0 and 1) are represented by the absence or presence of certain signal molecules. As their counter relies on concentrations of molecules, it cannot be space efficient as shown by our results in Section 4.3.

4.2 GRAY: a binary reflecting Gray code counter

Here we describe the *Chemical Reaction Network (CRN)* and *DNA Strand Displacement system (DSD)* implementation of our GRAY counter, provide a proof of its correctness, and analyze its expected time and space usage. We show how it can be modified to use only bi-molecular reactions, resulting in our fixed-order GRAY counter: GRAY-FO.

4.2.1 Chemical reaction network for the GRAY counter

We generalize the 3-bit GRAY counter in Section 4.1.2 to n bits. The counter state is represented by n signal molecules, one per bit. Presence of signal molecule b_i denotes that the i^{th} bit has value b_i, for $b = 0$ or $b = 1$. Initially, the state is $0_n \dots 0_2 0_1$. Each possible state of the counter represents a value in the Gray code sequence. The counter is described abstractly by the following chemical reaction equations:

$$\begin{aligned} &\text{(gc-1)} \quad \mathbf{0}_1 \quad \rightleftharpoons \quad \mathbf{1}_1 \\ &\text{(gc-}i\text{)} \quad \mathbf{0}_i \quad \overset{1_{i-1}+0_{i-2}+\cdots+0_1}{\rightleftharpoons} \quad \mathbf{1}_i, \ 2 \leq i \leq n \end{aligned}$$

All CRNs we propose are *tagged Chemical Reaction Networks (tagged CRNs)* and therefore account for the space of required transformers. However, to simplify the reaction equations and since each reaction has a unique transformer in our various implementations, we will omit the actual tag symbols.

Lemma 13. The above CRN ensures that the n-bit GRAY counter correctly advances through the 2^n states of the binary reflecting Gray code sequence, if each reaction is atomic[15] and all initial signal molecules exist as single copies. Furthermore, to advance through all counter states, each reaction is applied alternately in the forward and reverse direction.

Proof. Proof by induction on the number of digits. The claims are vacuously true for $n = 0$. Assume they are true for a counter with $i - 1$ bits and consider the construction of the i-bit counter. The initial signal multiset is the same as the $(i - 1)$-bit counter, except it contains the signal 0_i. It also contains one additional reaction, to flip the i^{th} bit. However, this reaction cannot occur for the first time until the signals $1_{i-1}, 0_{i-2}, \dots, 0_1$ are present as catalysts. These are exactly the signals for the last state of the $(i-1)$-bit counter. Thus, since the $(i-1)$-bit counter is correct by the induction hypothesis, the first 2^{i-1} reactions are exactly those of the entire $(i - 1)$-bit counter sequence. The i^{th} reaction can then be applied, otherwise the computation reverses to a previous state.

[15] "Atomic" is standard computer science terminology for something that occurs as if all at once, hearkening back to the original Greek etymology of an atom as an indivisible unit. Reasoning about chemical reactions as computational processes can unfortunately result in clashes in terminology.

Since the i^{th} reaction does not consume any signals for bits less than i, the entire reaction chain of the $(i-1)$-bit counter is reversed (as it does not interact with the i^{th} bit), otherwise the computation would reverse to a previous state. Importantly, the 0_i signal was consumed and since it was present as a single copy, the i^{th} reaction cannot be applied again once the $(i-1)$-bit reaction chain has begun reversing. Since the reactions alternated in the forward and reverse direction in the $(i-1)$-bit counter, they continue the alternation when the chain is reversed as the first reaction after the i^{th} bit is flipped is the reverse of the last reaction prior to flipping the i^{th} bit. Overall, the i-bit counter correctly advances through $2(2^{i-1}) = 2^i$ states and alternates the direction of reactions when signal molecules are present as single copies. □

4.2.2 DNA strand displacement implementation of the GRAY counter

Recall from Section 1.2.5 and Theorem 1 that the QSW construction is capable of simulating any tagged CRN by a space efficient DSD. Unfortunately, the construction does not simulate the higher-order reactions atomically, since some product signal molecules can initiate other reactions before all product signal molecules are produced. However, the toehold mediated strand displacements do occur in a fixed order and all reactant signal molecules are consumed before any product signal molecule is produced. We exploit this fact to simulate atomicity.

In particular, we borrow the concept of *transactions* from database and concurrency theory — a group of operations that either completes or does not complete in its entirety, and does not interfere with any other transaction. We implement transactions using a simple synchronization primitive: a *mutex*. A transaction must acquire the mutex in order to start, and releases it only when it completes. This is analogous to processes blocking when another process is in a critical section (which by definition must appear atomic). We consider the state of our counter to be defined only when the mutex is available. More precisely, let μ denote a single copy of a signal molecule species representing the mutex. In any sequence of strand displacements representing a chemical reaction, μ is the first reactant to be consumed and the last product to be produced. Therefore only one chemical reaction (transaction) can be in progress at any given time. When μ is next available, either all strand displacements in the sequence took place and the counter is in a new state—the transaction succeeded—or the counter is in the same state and the configuration of all molecules is exactly the same prior to the reaction beginning—the transaction failed. Since each reaction is implemented as a transaction, it appears atomic and cannot interfere with other reactions.

An example of the signal molecules and the transformer associated with the forward direction of the reaction $\mathbf{0_1} \rightleftharpoons \mathbf{1_1}$ which requires the availability of the mutex signal μ is given in Figure 4.5. Contrast this with the implementation of the same reaction that does not use a mutex signal in Figure 1.8.

As previously discussed, the reaction can only initiate if the signal molecule

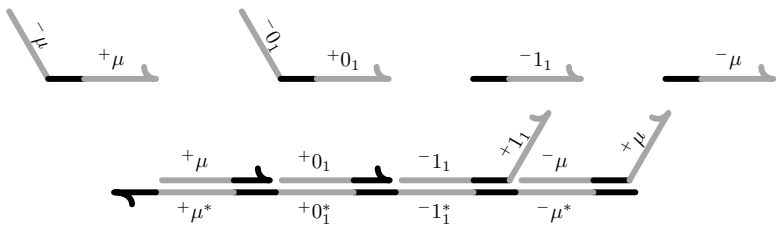


Figure 4.5: An example of signal molecules (top two left strands) and the transformer, consisting of auxiliary strands (top two right strands) and a saturated template strand (bottom complex) associated with the forward direction of reaction equation $\mathbf{0_1} \rightleftharpoons \mathbf{1_1}$ which requires a mutex. In this and later figures, the Watson-Crick complement of a domain x is denoted by x^*.

μ is present, and can only complete if all other reactants—in this case 0_1, assuming a forward reaction—are available. An example of the sequence of strand displacements for the reaction $\mathbf{0_1} \rightleftharpoons \mathbf{1_1}$ is given in Figure 4.6. The reaction proceeds from top to bottom in the forward direction and from bottom to top in the backwards direction.

The transformers that implement the i^{th} reaction (gc-i) are a straightforward generalization of the first reaction. As before, the signal molecule μ must initiate the first strand displacement, and is not produced until the last strand displacement. The number of required intermediate strand displacement reactions is dependent on the number of reactants and products. Specifically, the i^{th} reaction requires $2i + 2$ strand displacements to complete. An example of the transformer for the i^{th} reaction is given in Figure 4.7.

We now formally prove that the general DSD construction of Theorem 1 can be augmented with a mutex to ensure that all reactions occur as transactions: a sequence of toehold mediated strand displacements, the first of which consumes a *mutex strand* and the last of which produces a mutex strand, while no other no other displacement of the sequence produces a mutex strand. Our primary motivation in this CRN to DSD conversion is to ensure that (i) all reactions occur as transactions, and (ii) that the space complexity of the resulting DSD remains polynomially bounded by the space complexity of the original CRN. However, we note that this construction inherently enforces *serial* computation. That is, only a single reaction can occur at any one time. By design, our CRNs are meant to operate in this manner. However, this conversion could slow down other CRNs, designed to perform many parallel reactions, by a factor of $O(v)$ in the worst case, where v is the size of the reaction volume.

Theorem 12. Any logically reversible tagged CRN requiring $O(s)$ space can be simulated by a DSD in $O(poly(s))$ space, while ensuring that all chemical reactions occur as transactions (and therefore appear *atomic*) assuming all strand displacements are legal.

Proof. Let $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}}, T, T_0 \rangle$ be any logically reversible tagged CRN

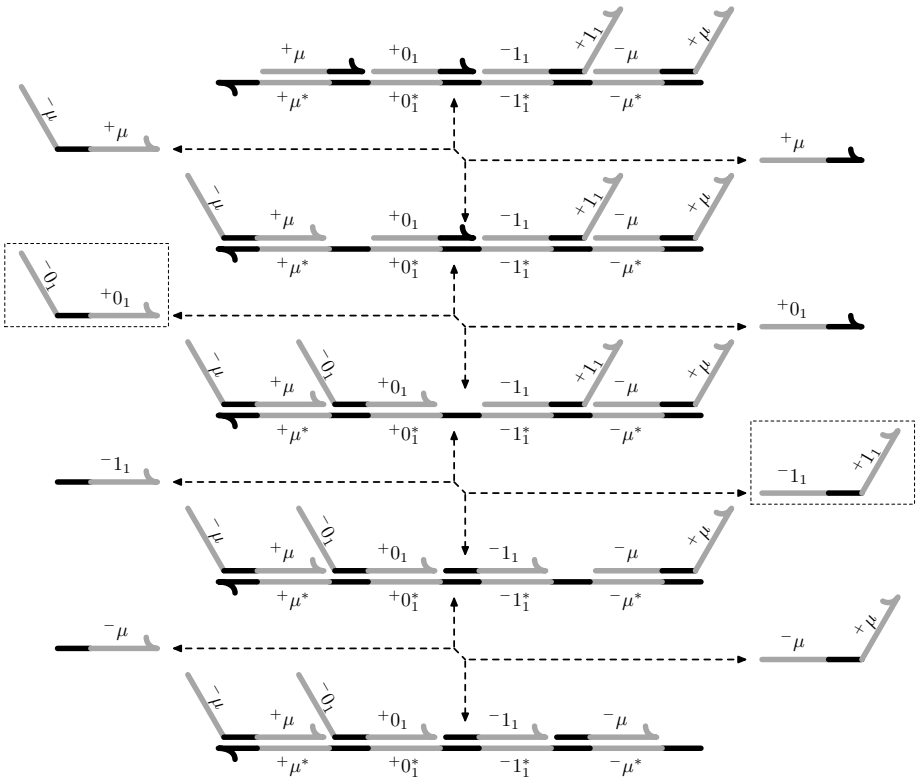


Figure 4.6: The sequence of strand displacement events for the reaction equation $\mathbf{0_1} \rightleftharpoons \mathbf{1_1}$ when a mutex signal μ is required. The mutex is the first signal to be consumed and the last to be produced, in either reaction direction. Otherwise, the reaction cascade proceeds exactly as before as dictated by the QSW construction.

requiring $O(s)$ space. We create an augmented logically reversible tagged CRN of $\mathbf{C}$ called $\mathbf{C}'$ as follows. We add the mutex signal species μ to S and one signal molecule of μ to S_0. For each reaction equation $R = (I, P) \in \mathcal{R}$, we add μ to both I and P. Note that we have only increased the number of reactants and products of a reaction by a constant and have only added a constant number of new signal molecules to the initial signal multiset. We construct a DSD $\mathbf{D}$ of $\mathbf{C}'$ using the QSW construction of Theorem 1 establishing most of the claim. All that remains is to show that the addition of the mutex signal forces each strand displacement cascade to occur as a transaction.

We argue by induction on the sequence of chemical reactions of the original CRN $\mathbf{C}$. Since $\mathbf{C}$ is logically reversible, then there is only one valid sequence of chemical reactions. Prior to any displacement simulating a chemical reaction, we will ensure the following invariant holds: (i) all template strands of all

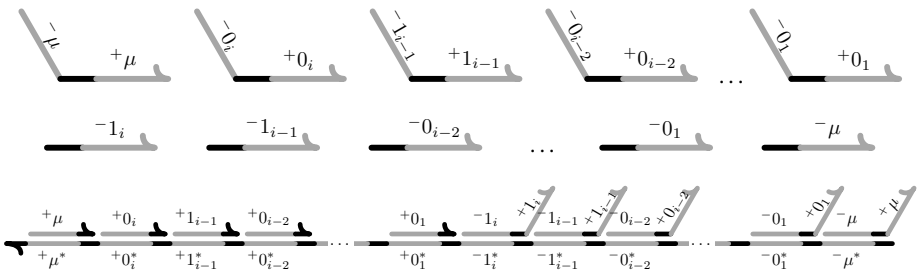


Figure 4.7: An example of the signal molecules and the transformer molecules for the i^{th} reaction. The counter is in state $b_n \ldots b_{i+1}\ 0_i\ 1_{i-1}\ 0_{i-2} \ldots 0_1$.

transformers are saturated and require the mutex signal molecule μ to initiate the first strand displacement, and (ii) there is exactly one available copy of μ. The invariant is trivially satisfied for the base case, when no reaction has yet occurred. Suppose the first $i-1$ reactions appear atomic, and the invariant is satisfied. Without loss of generality, suppose the next attempted reaction involves the k^{th} transformer.

Because we assume that all strand displacements are legal, no auxiliary strand or signal strand that is not μ can displace any strand in any transformer. Since there is exactly one available copy of the mutex signal species μ, that strand alone can initiate a reaction. Suppose the reaction is in the forward direction, as the reverse direction is symmetric. The signal molecule μ must initiate the first strand displacement by binding to the left end of the k^{th} transformer's template strand. This begins the transaction. Note that there is another copy of μ sequestered at the right end of the template. When the signal strand μ is once again produced, there are two cases to consider.

Case 1. If the copy on the right end of the transformer is released, then the transaction succeeded. It now appears that all input strands—the reactants—have been consumed, and all output strands—the products—have been produced. Furthermore, the invariant is preserved as (i) the k^{th} transformer is saturated, and only a signal strand μ can initiate a new reaction on the right end of the template, and (ii) exactly one signal strand μ was produced as the final strand displacement.

Case 2. Otherwise, the original copy of μ was released, the transaction failed, and the system is in the same state as before the reaction had begun, satisfying the invariant, as any intermediate strand displacements must have been reversed prior to the original μ signal strand being released.

Importantly, whether or not a transaction succeeds, while one is in progress no other reaction can be initiated since no other copy of the signal strand μ is available. Thus, all reactions are implemented as transactions and appear atomic. □

4.2.3 Space and expected time analysis of the GRAY counter

Here we analyze the space—the total number of nucleotide bases of all required strands in the reaction volume—and the expected time of the GRAY counter as it advances from initial to final states. We assume single copies of the initial signal, transformer, and mutex strands. Importantly, we note that since reactions occur alternately in the forward and reverse direction according to Lemma 13, then only a single copy of each reaction transformer is necessary. We note that our space analysis carries an assumption: $\Theta(n)$ domains of the signal species can be designed to have length $\Theta(n)$ such that only legal displacements occur for the duration of the counter. This seems to be a reasonable assumption one can make when considering existing results from coding theory. Schulman and Zukerman [114] show how to construct a set of $2^{\Theta(n)}$ domains (*i.e.,* binary strings in their code) of equal length $\Theta(n)$ such that the energy barrier (Levenshtein distance) between any pair of domains is at least cn, for any given constant c. However, we note that while a long domain of length $\Theta(n)$ may be sufficient to avoid illegal displacements, it may not be necessary. It may be the case that this bound is loose, and domains of length $\Theta(\log n)$ are sufficient.

Lemma 14. Assuming long domains have length $\Theta(n)$, the total number of nucleotide bases needed for a single copy of each initial signal, transformer, and mutex species of the n-bit GRAY counter is $\Theta(n^3)$.

Proof. Each signal strand 0_i and the initial mutex strand μ is composed of a toehold and two long domains. The same is true of the strands for states 1_i and the sequestered signal strands μ that are part of the initial transformer species. There are auxiliary transformer strands consisting of one toehold and one long domain for each type of signal species. We choose the toehold length to be $\Theta(1)$. Since the domain length dominates the toehold length, the total number of bases in all signal species and auxiliary strands is $\Theta(n^2)$.

The template strands for the i^{th} transformer have $\Theta(i)$ domains, which dominate their length, and therefore have length $\Theta(in)$. Thus, the total number of bases in all transformer template strands in the system is $\sum_{i=1}^{n} \Theta(in) = \Theta(n^3)$. □

Next consider the expected time for the counter to progress from its initial to its final state. Other than introducing the concept in Chapter 1, we have thus far ignored the *rate* of reactions in a chemical reaction network[16]. Briefly, in the DSD implementations of all networks proposed in this thesis, reactions always occur between two species present as single copies in the reaction volume. If the reaction volume has size V, the bimolecular reaction rate involving these two single copy species (*i.e.,* the time to find each other and interact) is $1/V$.

Lemma 15. Assuming a single copy of each initial signal, transformer, and mutex species, and that all strand displacements are legal and all reactions

[16]For a detailed overview of chemical reaction rates, particularly for strand displacement systems, the reader is referred to the PhD thesis of David Yu Zhang [149].

occur as transactions (appear atomic), the GRAY counter advances through the 2^n states of the binary reflecting Gray code sequence in $\Theta(n^3 2^{2n})$ expected time.

Proof. We assume that reactions occur in a volume of size $\Theta(n^3)$, since this is the total number of nucleotides required to represent all strands of the system. Each strand displacement step involves interaction between two strand species and thus the rate of each strand displacement step is $1/\Theta(n^3)$.

First, consider the *shortest path* from the initial state to the final state. On this path, each order-i reaction is applied 2^{n-i} times and involves $\Theta(i)$ strand displacements. Thus the total number of strand displacement steps along the shortest path is $\sum_{i=1}^{n} \Theta(i)2^{n-i} = \Theta(2^n)$.

Because each reaction is reversible, the system does not strictly follow the shortest path but rather proceeds as an unbiased random walk along the logically reversible computation chain. The expected number of steps for a random walk to reach one end of a length-$\Theta(2^n)$ path from the other is $\Theta((2^n)^2) = \Theta(2^{2n})$ [32]. Therefore, the expected number of strand displacement steps is $\Theta(2^{2n})$. Since each strand displacement step occurs at a rate of $1/\Theta(n^3)$, the overall expected time is $\Theta(n^3 2^{2n})$. Note that the expected time is polynomial in the $\Omega(2^n)$ steps required to proceed through all 2^n unique states of an n-bit binary reflecting Gray code counter. □

Combining Lemmas 13 through 15 and Theorem 12 we have the following result.

Theorem 13. An n-bit binary reflecting Gray code counter can be implemented as a DNA strand displacement system that proceeds through the 2^n unique states of the binary reflecting Gray code sequence in $\Theta(n^3 2^{2n})$ expected time and uses only $\Theta(n^3)$ nucleotides (space).

4.2.4 A fixed order implementation of the GRAY counter

An n-digit GRAY counter can perform a computation having length exponential in n, while only using space polynomial in n. However, it relied on template strands containing $O(n)$ domains, each of length $O(n)$, resulting in an overall length of $O(n^2)$ nucleotides. Synthesis of long nucleic acid strands is challenging, and the fidelity of synthesized strands generally decreases as sequence length increases. For this reason, it is desirable to bound the length of all strands in the system to $O(n)$ bases. We now briefly describe how a template strand from the GRAY counter consisting of $2i+2$ domains, can be split into $i+1$ template strands requiring 4 domains each, for any $i > 1$. The overall space will only be increased by a constant, resulting in the same volume, and thus the same expected time.

To simplify the description we introduce some notation. Consider the (gc-i) reaction of the GRAY counter which has, including catalysts, i reactants and i products:

$$\mathbf{0}_i \overset{1_{i-1}+0_{i-2}+\cdots+0_1}{\rightleftharpoons} \mathbf{1}_i$$

Theorem 12 demonstrated that by using the QSW construction and introducing a mutex species μ—thus creating an order $i+1$ reaction—chemical reactions occur as transactions and therefore appear atomic. Specifically to our counter, μ is first consumed, then 0_i, then, 1_{i-1}, and so on. Likewise, after all reactants are consumed, 1_i is first produced, then 1_{i-1}, and so on, until finally μ is produced. We denote a strand displacement implementation supporting a transaction of this type, which is initiated by consuming a mutex α, and terminated when producing a mutex β, by:

$$[\alpha + 0_i \overset{1_{i-1}+0_{i-2}+\cdots+0_1}{\rightleftharpoons} 1_i + \beta]$$

In the case of the GRAY counter, $\alpha = \beta = \mu$. Our goal is to convert this order $i+1$ reaction into a cascade of $i+1$ bi-molecular reactions, while preserving the appearance of atomicity. Using the above notation, we implement the following reaction cascade:

$$\left.\begin{array}{llll} [& \mu & \overset{1_{i-1}}{\rightleftharpoons} & \alpha_i^1 \quad] \\ [& \alpha_i^1 & \overset{0_{i-2}}{\rightleftharpoons} & \alpha_i^2 \quad] \\ & & \vdots & \\ [& \alpha_i^{i-2} & \overset{0_1}{\rightleftharpoons} & \alpha_i^{i-1} \quad] \end{array}\right\} \text{catalysts checked in sequence}$$

$$\left.\begin{array}{llll} [& 0_i & \overset{\alpha_i^{i-1}}{\rightleftharpoons} & 1_i \quad] \\ [& \alpha_i^{i-1} & \overset{1_i}{\rightleftharpoons} & \mu \quad] \end{array}\right\} i\text{-th bit flipped and } \mu \text{ released}$$

The overall transaction has been split into a cascade of sub-transactions. Each sub-transaction is implemented as a bi-molecular reaction using Theorem 12 (based on the QSW construction). The first $i-1$ sub-transactions check, in sequence, that all $i-1$ catalysts are present. The mutex signal molecule μ is consumed during the first check. The last two transactions will first perform the bit flip and then release the mutex signal molecule. Every sub-transaction, except the last two, produces a unique mutex signal species that is required to initiate the next sub-transaction in the cascade. Upon successful completion of the first i sub-transactions in the cascade, the final sub-transaction occurs, producing the original mutex signal species μ, and thus finalizing the overall transaction. Note that in all cases, once the transaction has begun and before it completes, the original mutex signal μ is absent, and therefore no other reaction cascade can commence. The implementation works in the reverse direction in a similar way with the exception that once the original mutex signal μ is consumed, the bit is flipped first and the mutex signal strand μ is released only after the presence of all catalysts have been verified. Note that as in the previous case, as the mutex strand μ is missing until either the entire transaction completes, or reverses, no other reaction in the system can occur. Thus, flipping the bit prior

to verifying all catalysts are present does not affect the correctness of the system. In the case that all catalysts are not present, the transaction cascade cannot complete, and will necessarily reverse. Using the above transformation for all higher-order reactions in the original GRAY counter implementation results in a new, fixed order counter, GRAY-FO.

4.2.5 Comparison with another molecular counter

Table 4.1 summarizes properties of our counters and compares with another counter, which we call QSW, based on work of Qian et al. [98] (see Section 4.1.4). The properties considered are (i) *order* or max number of reactants or products of chemical reactions that describe the counter, (ii) *space* or total number of nucleotides needed to implement the counter and (iii) *expected time* for the counter to reach a designated final state from its initial state when the volume equals the space. We describe how order, space and expected time grow as a function of n, the number of counter bits. In all cases, we make the assumption that the length, in bases, of long domains is $\Theta(n)$.

First, consider the QSW counter. Qian et al. [98] showed how to simulate a stack machine using strand displacements systems. A binary counter could be implemented via a stack machine. An n-bit implementation of the QSW counter advances deterministically through 2^n states and uses reactions of order 2 (some of which involve polymer extension reactions that realize the stack). The transformer molecules used in the strand displacement realizations of these reactions can serve as fuel, biasing the reaction so that the counter advances. We analyze the biased version of the counter; the unbiased version is slower. The expected number of reactions for the biased counter to advance to its final state is $\Theta(2^n)$. Each reaction consumes a constant number of molecules and so the overall expected consumption, or waste, is $\Theta(2^n)$. The expected time depends on the volume in which the reaction takes place. If all strands consumed are initially present in the reaction volume, then the volume is $\Theta(2^n)$ and thus each step takes expected time $\Theta(2^n)$, leading to an overall expected time of $\Theta(2^{2n})$.

Our n-bit binary reflecting Gray code counter, GRAY, uses reactions of maximum order $\Theta(n)$, generates only $\Theta(n^3)$ waste and uses expected time $\Theta(n^3 2^{2n})$ to reach the final state. Our GRAY-FO counter improves on the GRAY counter in that the reaction order is $\Theta(1)$. The QSW counter also has reaction order $\Theta(1)$ and has expected time $\Theta(2^{2n})$, which is somewhat better than the expected time needed by our counters. However, the QSW counter generates $\Theta(2^n)$ waste, exponentially worse than our counters. All three counters are deterministic in that they advance and retreat through a predetermined linear ordering of states (*i.e.,* they are logically reversible).

Properties	GRAY	GRAY-FO	QSW [98]
Reaction order	$\Theta(n)$	$\mathbf{\Theta(1)}$	$\mathbf{\Theta(1)}$
Space (in nucleotides)	$\mathbf{\Theta(n^3)}$	$\mathbf{\Theta(n^3)}$	$\Theta(2^n)$
Expected Time	$\Theta(n^3 2^{2n})$	$\Theta(n^3 2^{2n})$	$\mathbf{\Theta(2^{2n})}$

Table 4.1: Comparison of n-bit counter implementations. The GRAY and GRAY-FO counters described in this section are compared with the QSW counter which is based on the simulation of stack machines by strand displacement reactions of Qian et al. [98].

4.3 Limits on molecule recycling in chemical reaction networks

In this section, we show that all chemical reaction networks that efficiently recycle strands, or that can perform useful computations for a number of steps that significantly exceeds the number of signal molecules, are not deterministic when multiple copies of the initial signal molecules are present.

The underlying problem is the representation of the state of the network as specific *combinations* of signal molecules. If there are multiple copies of the network in the same reaction volume — as would typically occur in a laboratory setting[17] — then the states of the different copies may interfere with one another — a process we call *transaction*. To illustrate this point, we again consider the 3-bit GRAY counter. Initially, in a single copy of the construction, the signal molecules $\{0_3, 0_2, 0_1\}$ denote the state $0_3 0_2 0_1$. Consider a two-copy network where the initial multiset of present signal molecules is duplicated, yielding the multiset $\{0_3, 0_3, 0_2, 0_2, 0_1, 0_1\}$. (We also assume a duplicate multiset of transformers is available.) As in the single copy case, assume reaction (1) occurs in the forward direction, followed by reaction (2) in the forward direction. The resulting multiset of signal molecules is $\{0_3, 0_3, 0_2, 1_2, 0_1, 1_1\}$. In the single copy case, we intend that reaction (1) in the reverse direction will occur next; however, given the current multiset of present signal molecules in the two-copy case, reaction (3) in the forward direction could instead occur, resulting in the multiset $\{0_3, 1_3, 0_2, 1_2, 0_1, 1_1\}$. At this point, a copy of every signal molecule is present, and any reaction can occur, in either direction. Furthermore, the single copy case required *at least* seven reactions to produce the final state $1_3 0_2 0_1$, whereas the two-copy case can reach it in three. Crosstalk between the copies has broken the counter.

Recall the formal definition of a CRN $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}} \rangle$ and related concepts from Section 1.2.2. In addition, we use B_s to denote the *bandwidth* of signal species s, *i.e.,* the maximum number of copies of s that appears in a multiset I of any reaction $(I, P) \in \mathcal{R}$, and we use $B_{\mathbf{C}}$ to denote the bandwidth

[17] We note that while significantly more challenging, single molecule experiments are possible. This is discussed in more detail in Chapter 6.

of $\mathbf{C}$, *i.e.,* the sum of bandwidths of all signal species in S. An *x-copy* version of $\mathbf{C}$, denoted $\mathbf{C}^{(x)}$, is obtained by duplicating the initial multiset S_0 x times, *i.e.,* $\mathbf{C}^{(x)} = \langle S, \mathcal{R}, S_0^{(x)}, s_{\text{end}} \rangle$ where $S_0^{(x)}$ is a multiset consisting of x copies of S_0.

Theorem 14. Let $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}} \rangle$ be a 1-proper chemical reaction network. If there exists a trace that produces s_{end} in $\mathbf{C}$ then for the x-copy chemical reaction network $\mathbf{C}^{(\mathbf{x})}$ with $x \geq B_{\mathbf{C}}+1$, there exists a computation that produces s_{end} in at most $(B_{\mathbf{C}}+1)B_{\mathbf{C}}/2+1$ steps.

Proof. Let $\rho = R_1, \ldots, R_m$ be a trace for a computation that produces s_{end} in the last step in the (single-copy) network $\mathbf{C}$ and let $S_0, \ldots, S_m$ be the corresponding sequence of multisets of signal molecules. Let S' be the multiset of signal molecules obtained by including w_s copies of each $s \in S$, where $w_s \geq 0$ is the maximum number of copies of signal molecule s that appears in a multiset I of any reaction (I, P) of the sequence ρ. Note that $|S'| \leq B_{\mathbf{C}}$. Let $k = |S' - S_0|$ and note that $k \leq B_{\mathbf{C}}$. The goal is to produce all signal molecules in the multiset $S' - S_0$, so that we can apply the last reaction of ρ and produce s_{end}.

We construct a trace of the appropriate length for the multi-copy network from the trace ρ for the single-copy network. The high-level structure of the proof is as follows: First, we project out from ρ the k reactions, in order, that first produce each of the molecules in the multiset $S' - S_0$. From that sequence, we build a trace of the multi-copy network that is the concatenation of k phases. Each phase adds one more signal molecules to the multiset of signal molecules present, preserves the presence of all signal molecules previously produced, and "consumes" one copy of the initial signal molecules in S_0. We will show that the j-th phase is at most j reactions long, so the total length of the trace producing $S' - S_0$ is bounded by $\sum_{j=1}^{k} j = (k+1)k/2 \leq (B_{\mathbf{C}}+1)B_{\mathbf{C}}/2$.

We now formalize the construction of the k phases. Define the first appearance of the c-th copy of signal molecule s to be in S_i if there are at least c copies of s in multiset S_i and less than c copies of s in each of $S_0, S_1, \ldots, S_{i-1}$. Let $s_1, \ldots, s_k$ be the sequence of signal molecules (with multiplicities) from $S' - S_0$ in order of their first appearances in $S_1, \ldots, S_m$ and let $R_{\text{index}(s_j)}$ be the reaction in ρ which first produced this copy of s_j. In other words, $R_{\text{index}(s_j)}$ is the reaction that produced the first appearance of s_j (where s_j is the c-th copy of some signal molecule s, for some c). The k phases will produce the signal molecules in $S' - S_0$ exactly in this order: signal molecule s_j will be produced in phase j. Each phase j will consist of several reactions, numbering 0 to j, which will produce s_j without removing any other signal molecule from multiset $S' - S_0$, but they can remove one signal molecule from S_0. This is replenished by adding one new copy of S_0 at the beginning of this phase. To find the sequence of these reactions, we will work backwards. Assuming s_j is not already present in the current multiset of signal molecules, we use reaction $R_{\text{index}(s_j)}$ to produce s_j. As a result, we might have removed one of the signal molecules $s_1, \ldots, s_{j-1}$, say s_i. If that is the case, we repeat the process of producing signal molecule s_i, *i.e.,*, repeating reactions of phase i.

The k phases are constructed to maintain three invariants:

1. After the j-th phase, the multiset of signal molecules contains the multiset $\{s_1, \ldots, s_j\}$.
2. The trace constructed so far has not relied on the existence of more than j copies of the initial signal molecules S_0.
3. For each $i \leq j$, the i-th phase have used at most i reactions.

The invariants are vacuously true initially (before any phases). Assuming they are true after $j-1$ phases, we construct the j-th phase as follows. If s_j is already present in the current multiset of molecules, we do nothing; it was fortuitously produced in an earlier phase. Otherwise, the first reaction in the phase is $R_{\text{index}(s_j)}$, the reaction that produced s_j for the first time. We know this reaction can be applied because all of $\{s_1, \ldots, s_{j-1}\}$ are available, as well as the j-th copy of S_0. This guarantees that the multiset now contains s_j, and we have relied on only j copies of S_0. However, since the network is 1-proper, the reaction consumed at most 1 input signal molecule. If the reaction consumed 0 molecules, or if the 1 molecule is in S_0, the invariant is maintained and the phase ends. Otherwise, the reaction consumed some s_i, where $i < j$. To restore s_i to the multiset, we repeat the sequence of reactions of the i-th phase. Note that this is valid since the new copy of S_0 was not yet used and all signal molecules required for phase i are still present. The number of reactions of phase j can be bounded by the number of reactions of phase i plus one. By the induction invariant the i-th phase requires at most i reactions and since $i < j$, the j-th phase requires at most j reactions.

Concatenating the k phases produces a trace for the k-copy chemical reaction network $\mathbf{C}^{(k)}$, which produces all of $\{s_1, \ldots, s_k\}$ within $(k+1)k/2$ reactions. If s_{end} is not in $S' - S_0 = \{s_1, \ldots, s_k\}$, then S' contains all inputs needed for the last reaction in ρ that produces s_{end}. Thus to produce s_{end} we might need one additional copy of the initial multiset and one additional step. Since $k \leq B_{\mathbf{C}}$, the result follows. □

Note that Theorem 14 is much stronger than our intuitive notion of crosstalk short-circuiting a computation. It states that with only a linear number of copies, *any* signal molecule can be produced in at most a quadratic length computation. Although it applies only to 1-proper networks, it is sufficient to show that the GRAY counter does not work correctly when there are enough copies present. Furthermore, since there is a direct transformation between GRAY-FO and GRAY, it also demonstrates that our GRAY-FO counter is not robust in the multi-copy setting.

We can formalize the intuitive notion of short-circuiting. A network $\mathbf{C}$ is *x-copy-tolerant* if, for all $s \in S$, the length of the shortest trace to produce s in $\mathbf{C}$ and in $\mathbf{C}^{(x)}$ is the same. A network is *copy-tolerant* if it is *x-copy-tolerant* for all x. The result of Theorem 14 only pertains to a restricted class of chemical reaction networks. This raises the question of whether it is possible

to design a *copy-tolerant* network, which is not 1-proper and that performs a computation exponential in its size. As we are only interested in deterministic chemical reactions networks that properly account for space when representing DSDs (*i.e.,* tagged chemical reaction networks), we can achieve a tighter bound by focusing on the specific class of networks we study in this thesis.

Theorem 15. For any tagged chemical reaction network $\mathbf{C} = \langle S, \mathcal{R}, S_0, s_{\text{end}}, T, T_0 \rangle$, if there is a deterministic computation that produces s_{end} in $t > |T_0|$ steps, then the 2-copy network $\mathbf{C^{(2)}}$ is not deterministic after $2|T_0| - 2$ steps.

Proof. Let $\rho = R_1, R_2, \ldots, R_t$ be the deterministic trace of reactions from the original network $\mathbf{C}$ which induces a corresponding sequence of multisets $S_0, S_1, \ldots, S_t$, with $s_{\text{end}} \in S_t$. For convenience, we let R_k denote the reaction applied at step k, for $1 \leq k \leq t$.

Since reaction tags are counted in the initial tag multiset T_0, then there must exist some reaction, previously applied in the forward direction[18] at step i, that is the first to be applied in the reverse direction at some later step $j \leq |T_0| + 1$. Otherwise, all the tags would be consumed and the computation would halt within $|T_0|$ steps. Let $R_i = (I_i, P_i)$ denote the forward version of this reaction which consumes the multiset $X = I_i - P_i$ and produces the multiset $Y = P_i - I_i$. Thus, R_j consumes Y and produces X. Note that $P_i \subseteq S_i$.

Next consider that $i \neq j - 1$. Suppose it were. The multiset of signals prior to applying R_{j-1} is S_{j-2}, and it is $S_{j-2} - X + Y$ afterward, if $i = j - 1$. Next applying reaction R_j, which consumes Y and produces X by definition, results in the multiset S_{j-2}. The computation is now stuck in a length-2 cycle and has only advanced to at most $j-1 = |T_0|$ new states. Therefore, $R_{j-1} = (I_{j-1}, P_{j-1})$ is not the reverse of R_j. Note that $I_{j-1} \subseteq S_{j-2}$.

Now, we will construct a trace ρ' for the network $\mathbf{C^{(2)}}$. Let the first $j - 2$ reactions be the same as in ρ. Thus, the resulting multiset after $j - 2$ steps is $S'_{j-2} = S_{j-2} + S_0$; otherwise, the original trace ρ is not valid. Next, append the first i reactions of ρ. By the same reasoning, this will result in the multiset $S'_{j-2+i} = S_{j-2} + S_i$. Since $I_{j-1} \subseteq S_{j-2} \subseteq S'_{j-2+i}$ then reaction R_{j-1} can be applied. Since $P_i \subseteq S_i \subseteq S'_{j-2+i}$ then reaction R_j, the reverse of R_i, can be applied. Since one is not the reverse of the other, the computation is not deterministic after step $j - 2 + i \leq 2j - 4 \leq 2|T_0| - 2$. □

The result of Theorem 15 states that no tagged chemical reaction network will be deterministic for more than a linear number of steps when a second copy of the initial signal and tag multisets are present. This is formally stated in the following corollary.

Corollary 1. It is not possible to design a tagged chemical reaction network that performs deterministic computation within a number of steps that is super-linear in the size of the initial tag multiset and that is also 2-copy-tolerant.

[18] Without loss of generality, we can assume all reactions in a chemical reaction network are always applied first in the forward direction.

4.4 Chapter summary

In this chapter we have introduced the concept of recycling, or molecule reuse, in strand displacement systems and chemical reaction networks. Our n-bit GRAY counters effectively use recycling to deterministically step through 2^n states while requiring space, or total number of nucleotides, of just $O(n^3)$. Our GRAY counter strand displacement constructions also introduce the use of a *mutex* strand to ensure that higher-level chemical reactions are executed atomically. Finally, we show limits to recycling: for example, any signal molecule of our n-bit counter can be generated using just $O(n^2)$ reactions when $\Theta(n)$ copies of the initial signal molecules share the same volume. We have also shown that even having a second copy of the initial tag and signal multisets ensures that no computation which uses transformers can be deterministic after a linear number of steps.

One weakness of our counter construction is that the number of distinct domains needed is polynomial in n, the number of bits of the counter. In contrast, a QSW binary counter that is implemented via the stack machine of Qian et al. [98] uses just a constant number of distinct domains independent of n. Is it possible to construct an n-bit counter that combines the best of the GRAY and QSW counters, *i.e.,* uses space that is polynomial in n and uses $O(1)$ distinct domains? More generally, can *all* computation be realized by strand displacement systems whose space and expected time are within a (small) polynomial factor of the space and time of the computation? Our negative results suggest that any such systems must rely on exact molecular counts if they must have determinism.

Chapter 5

Space and energy efficient molecular programming

In the previous chapter, we demonstrated that space-efficient molecular programming with chemical reaction networks (CRN) and DNA strand displacement systems (DSD) is possible, in principle, by giving an implementation of a Gray code counter that performed a computation with a number of steps exponential in the required space. In this chapter, we ask the question: can any space efficient computation be realized by a space efficient CRN and DSD? We answer in the affirmative by showing how any problem in PSPACE can be solved by a logically reversible tagged CRN using polynomial space. We also demonstrate how this result can be extended to solve any space-bounded computation (*i.e.,* all of SPACE). Our CRN can be realized by a space and energy efficient DSD implementation. Not only do our results further characterize the computational power of CRNs and DSDs, they shed light on the complexity of a number of important related problems such as CRN and DSD model checking and verification [64, 65]. We show that even determining if an arbitrary state is reachable from an initial state of a CRN or DSD—a question that must be solved when verifying the correctness of a CRN or DSD—is PSPACE-hard. We show that the problem is PSPACE-complete for restricted classes of CRNs and DSDs. In this chapter we also return to reasoning at the sequence level by showing how our new results can be used to establish that the minimum energy barrier indirect folding pathway for multiple interacting strands problem (EB-IPFP-MULTI) is PSPACE-complete.

5.1 Related work

As with the previously mentioned results related to CRNs and DSDs, we now highlight results related to the limits of logically reversible computation. An introduction to logically reversible and energy-efficient computation is given in Section 1.2.6. This is a topic that will be explored in this chapter in the context of molecular programming. Charles Bennett's seminal work, that showed how any $T(n)$ time-bounded Turing machine can be simulated by a logically reversible Turing machine, was space inefficient as his reversible Turing machine simulation required $\Theta(T(n))$ space [8]. He later improved the result to

Content from this chapter appears in the proceedings of the 18th Annual International Conference on DNA computing and molecular programming (DNA 2012) [131].

show that any Turing machine computation using $T(n)$ time and $S(n)$ space can be simulated by a logically reversible Turing machine to use $O(S(n) \log T(n))$ space [9]. This proved that PSPACE equals ReversiblePSPACE [9]—the class of problems solvable by a logically reversible Turing machine that uses polynomial space. The result has since been generalized to prove SPACE equals ReversibleSPACE [70] demonstrating that, in principle, any space-bounded computation can be solved by a space and energy efficient computation. Until recently it remained unclear if a physical system could realize logically reversible computation. In perhaps one of the most important theoretical results in the field of molecular programming, Qian et al. [98] gave a DSD implementation of a stack machine capable, in principle, of energy efficient Turing universal computation. However, as with Bennett's seminal work, their implementation requires space proportional to the number of steps in the computation as it consumes *fuel* (transformer) molecules to drive the overall process forward.

5.2 Preliminaries

Definitions of *DNA Strand Displacement systems (DSDs)* and *Chemical Reaction Networks (CRNs)* are given in Section 1.2. In this chapter, as with the last, we will reason exclusively about tagged CRNs. However, to simplify the presentation, we will omit tags from reaction equations.

CRNs can be implemented by DSDs in a number of ways [72, 98]. We will leverage the implementation from Theorem 12 which relies on the assumption that certain signals only occur as a single copy within the reaction volume. A single copy mutex species is used to ensure that a strand displacement cascade which implements any particular reaction will occur as a transaction and therefore appear *atomic*. Specifically, either the entire cascade implementing a reaction will succeed, or it will return to the state prior to beginning the cascade. Importantly, the mutex molecule is sequestered during the cascade and therefore another reaction cannot begin.

Finally, we formally define problems we study in this chapter. The first two problems ask whether certain states are reachable within a CRN or DSD and are the basis for formal verification of these systems. Recall that the *state* of a CRN (DSD) is the current composition of free signal molecules (strands).

Problem 7. CRNR (CRN reachability)
Instance: A chemical reaction network with initial state S_{init} and an arbitrary state S'.
Question: Is S' reachable from S_{init}?

Problem 8. DSDR (DSD reachability)
Instance: A DNA strand displacement system with initial state S_{init} and an arbitrary state S'.
Question: Is S' reachable from S_{init}?

In this chapter we will construct a CRN that can solve any instance of the totally quantified 3-satisfiability problem defined below.

Problem 9. Q3SAT (Totally quantified 3-satisfiability)
Instance: A totally quantified Boolean formula ψ of n variables in prenex normal form with strictly alternating quantifiers, $\forall x_n \exists x_{n-1} \forall x_{n-2} \dots Q_1 x_1 \ \phi$, where Q_1 is the quantifier $\forall$ if n is odd and the quantifier $\exists$ otherwise, and where ϕ is an unquantified Boolean formula of m clauses in conjunctive normal form, each containing a literal for 3 distinct variables.
Question: Is the formula ψ satisfiable?

Finally, we will resolve the complexity of predicting indirect folding pathways for multiple interacting strands. We will reason about this problem using the simple energy model formally defined in Section 1.1.1.

Problem 10. EB-IPFP-MULTI (Energy Barrier for Indirect Pseudoknot-free Folding Pathway of Multiple interacting strands)
Instance: Given two pseudoknot-free structures $\mathcal{I}$ (initial) and $\mathcal{F}$ (final), of multiple interacting strands, and integer k.
Question: Is there an indirect pseudoknot-free folding pathway from $\mathcal{I}$ to $\mathcal{F}$ such that the energy barrier in the simple energy model is at most k?

5.3 Space efficient CRN simulation of PSPACE

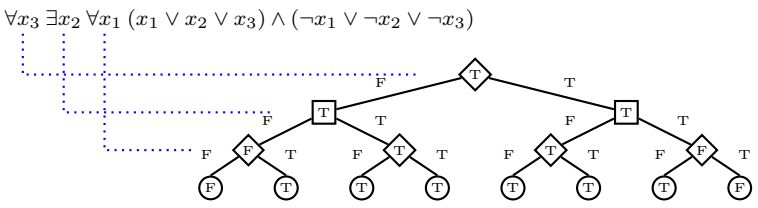


Figure 5.1: Solving a Q3SAT instance. Edge labeled paths from root to leaf denote variable assignments. Nodes are satisfied based on quantifier and satisfiability of left and right children.

Our goal is to demonstrate that any problem in PSPACE can be *solved* by a space efficient, logically reversible, tagged CRN. By solved, we mean that the CRN will produce a special accept signal for an instance of a problem

if and only if a Turing machine simulated with that same problem instance ends in an accepting state. Otherwise, the CRN will produce a special reject signal. To that end, we will show how a CRN with those properties can be constructed to solve any arbitrary instance of the Q3SAT problem, which is a complete problem for the class PSPACE. We present our solution in three logical parts. In Section 5.3.1, we demonstrate how to construct a CRN for verifying if a 3SAT formula is satisfied. In Section 5.3.2, we present an elegant solution for traversing a perfect binary tree in post-order that is both space efficient and logically reversible. In Section 5.3.3, we demonstrate how the two CRNs can be integrated and then modified to capture the semantics of strictly alternating variable quantifiers in the Q3SAT instance. To simplify the presentation of our result, we will add new reactions, tags, and signal molecules when needed as we refine our construction towards its final form.

To understand the intuition behind our construction, consider that a perfect binary tree of height n, with each level of the tree representing a variable, has 2^n leaves, each with a unique path from the root specifying a unique variable assignment. A tree defined in this manner can be used to express the semantics of strictly alternating quantifiers in the Q3SAT instance (see Figure 5.1). Leaf nodes are considered satisfied, or true, if and only if the current variable assignment satisfies the unquantified 3SAT formula of the Q3SAT instance. For example, the first leaf node from the left of the tree in Figure 5.1 is not satisfied, and is therefore considered false, as the variable assignment $x_1 = F, x_2 = F, x_3 = F$ does not satisfy the formula $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$; however, the second leaf node from the left with assignment $x_1 = T, x_2 = F, x_3 = F$ does satisfy the formula so it is considered to be true. Internal nodes can be used to propagate satisfiability of a partially solved instance up the tree. If an internal node represents a universally quantified variable, then it is marked as true if and only if both of its children are true. Therefore, the parent node of the first two leaf nodes from the left in Figure 5.1 is false as it is a universally quantified node and only one of its children is true. Similarly, a node representing an existentially quantified variable is marked false if and only if both children are false. In this straightforward manner, the overall quantified formula can be determined to be true or false, once the root is marked. Since the satisfiability of a node can immediately be determined once that of its two children is known, we perform a post-order traversal[19] of the tree. Furthermore, we exploit the fact that once the satisfiability of a child is marked, the satisfiability of its descendants is irrelevant and can be *forgotten*. This allows us to smartly reuse space in our tree traversal procedure.

5.3.1 Verifying a 3SAT instance variable assignment

We first demonstrate how the formula ϕ can be verified as satisfied or unsatisfied for a particular variable assignment. A variable assignment ensures exactly

[19] In a *post-order* tree traversal, a node is *processed* only after its children have been processed.

one signal for each variable x_i is present: x_i^T for a true assignment, and x_i^F otherwise. We first introduce the necessary reactions to verify an individual clause and demonstrate how the overall formula can be determined to be true or false.

Verifying an arbitrary clause

Recall that in a 3SAT instance, each clause consists of exactly three literals, each for a distinct variable[20]. As such, there are exactly eight possible truth assignments and we create a reversible reaction for each. The reactions for verifying the i^{th} clause, containing literals for variables x_j, x_k and x_l are given in Figure 5.2 (left). When the clause signal molecule $C_i^?$ is present, exactly one of the eight reactions can be applied, specified by the current variable assignment. The variable signals act as catalysts and the $C_i^?$ signal is consumed producing either a C_i^T signal if the clause is satisfied, or C_i^F otherwise.

$$
\begin{array}{lll|lll}
(1) & C_i^? \overset{x_j^F+x_k^F+x_l^F}{\rightleftharpoons} C_i^{[F/T]} & , 1{\le}i{\le}m & (2) & \phi^? \rightleftharpoons C_1^? & \\
 & C_i^? \overset{x_j^F+x_k^F+x_l^T}{\rightleftharpoons} C_i^{[F/T]} & , 1{\le}i{\le}m & (3) & C_i^T \rightleftharpoons C_{i+1}^? & , 1{\le}i{<}m \\
 & \vdots & & (4) & H_i^? + C_i^F \rightleftharpoons H_i^{\checkmark} + \phi^F & , 1{\le}i{\le}m \\
 & C_i^? \overset{x_j^T+x_k^T+x_l^T}{\rightleftharpoons} C_i^{[F/T]} & , 1{\le}i{\le}m & (5) & C_m^T \rightleftharpoons \phi^T &
\end{array}
$$

Figure 5.2: (left) Eight chemical reaction equations to verify an arbitrary 3SAT clause C_i for each combination of variable assignments. The product of the reaction is C_i^T for assignments that satisfy the i^{th} clause, and C_i^F otherwise. (right) Reaction equations to verify the overall 3SAT formula ϕ, consisting of m clauses.

For example, suppose C_i represents the following clause: $(x_j \vee \neg x_k \vee x_l)$. The reaction having catalysts x_j^F, x_k^T, and x_l^F will produce C_i^F. The other seven reactions will produce C_i^T. Note that for a particular variable assignment, only one reaction will apply in both the forward and reverse direction, ensuring the process is logically reversible.

Verifying the overall formula

The overall process of verifying the formula ϕ can be thought of as a subroutine that is initiated by consuming the signal $\phi^?$ and completes by producing either

[20]We assume this form to simplify our description. Note that when two or more of the literals in a clause are for the same variable, it is always possible to simplify the clause, even when they are negations of each other. It is also always possible to add dummy variables to ensure every clause has exactly three literals.

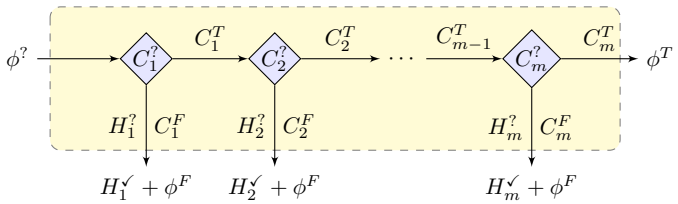


Figure 5.3: Flow control when verifying a formula ϕ having m clauses.

the signal ϕ^T, if ϕ is satisfied, or ϕ^F otherwise. The variable assignment signals are catalysts, and their values are maintained after the process completes.

For the formula to be true, all clauses must be satisfied. However, any combination of unsatisfied clauses will result in ϕ being false. For this reason, care must be taken that clauses are checked systematically to ensure reversibility. The overall process is depicted in Figure 5.3 and the reactions are given in Figure 5.2 (right). The process checks each clause, in sequence, and if the current clause is unsatisfied then reaction (4) occurs, immediately producing the ϕ^F signal denoting that the formula is unsatisfied. This reaction consumes a history signal $H_i^?$ and produces another history signal $H_i^\checkmark$. The sole purpose of the history signal is to ensure the reversibility of the computation, should the ϕ^F signal be produced as it uniquely identifies which clause was the first to be unsatisfied. Otherwise, all clauses are satisfied, and thus the signal ϕ^T can be produced and is sufficient to ensure the computation is reversible.

Lemma 16. A 3SAT Boolean formula of m clauses over n variables can be verified by a logically reversible tagged CRN in $O(m)$ reaction steps using $\Theta(m+n)$ space.

Proof. Importantly, we must now establish that the process is logically reversible. We first argue by induction on m, the number of clauses of the 3SAT formula ϕ, that C_m^T is eventually produced by a logically reversible sequence of reactions if and only if the first m clauses are satisfied and otherwise ϕ^F is produced by a logically reversible sequence of reactions along with a history signal denoting the first unsatisfied clause. In addition to the clause history signals $H_i^?$, we assume initially that the signal $\phi^?$ is present and exactly one signal for each variable x_i denoting its truth assignment—x_i^T or x_i^F. Suppose the inductive hypothesis holds for $m-1$ clauses and consider the case when ϕ has m clauses. We have two cases:

Case 1. The first $m-1$ clauses of ϕ are satisfied. By the inductive hypothesis, signal C_{m-1}^T will eventually be produced, by a logically reversible sequence of reactions. Other than the reverse of the previous reaction, only the reaction to produce $C_m^?$ can be applied. Next, other than reversing, only one clause reaction will be applicable and will produce either C_m^T or C_m^F. If C_m^T is produced, we are done. If C_m^F was produced, either the reverse of the previous reaction can be applied, or $\phi^F + H_m^\checkmark$ is next produced, ending the process. Thus, C_m^T or ϕ^F (in

addition to a history signal denoting the first unsatisfied clause) is eventually produced by a sequence of logically reversible reactions.

Case 2. At least one of the first $m-1$ clauses of ϕ are unsatisfied. By the inductive hypothesis, this case will correctly produce ϕ^F and a history signal denoting the first unsatisfied clause. The new reactions pertaining to clause m are not applicable and thus inconsequential.

To complete the process, if C_M^T was produced, other than reversing the previous reaction, the signal ϕ^T can next be produced. It is easy to see that in the worst case, $O(m)$ reactions steps are required. Finally, we establish the space claim. The initial signal multiset has size $\Theta(m+n)$ as it consists of the n variable signals, m clause history signals and the signal $\phi^?$. The CRN has $\Theta(m)$ reactions since there are a constant number for each of the m clauses and the overall formula verification. Since each reaction is applied at most once when verifying a formula, one tag per reaction is sufficient, therefore establishing the size of the initial tag multiset to be $\Theta(m)$. As the CRN is proper, then by Lemma 2 the required space to complete the computation is $\Theta(m+n)$. □

5.3.2 A space efficient post-order tree traversal

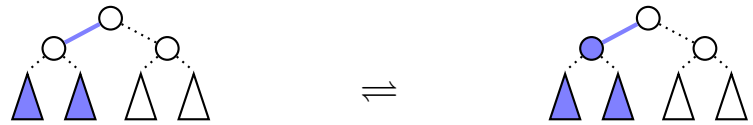
Next we demonstrate how to perform a post-order traversal of a perfect binary tree in a space-efficient manner. Importantly, the procedure must be logically reversible. The intuition and chemical reaction equations are captured in Figure 5.4. For any node with a left and right child, once the descendants of the left child have been recursively traversed (Figure 5.4 (a)), the left child can be marked (Figure 5.4 (b)) using reaction (6) *mark left*. Any information stored in those descendant nodes is no longer required and the whole traversal of that subtree can be reversed (Figure 5.4 (c)), the traversal can move to the right child (Figure 5.4 (d)) using reaction (7) *move right*, the right subtree can be recursively traversed (Figure 5.4 (e)), and finally the right child marked (Figure 5.4 (f)) using reaction (8) *mark right*.

Lemma 17. Given a perfect binary tree of height h, all descendants of the root can be traversed in post-order, by a logically-reversible tagged CRN in $\Theta(3^h)$ reaction steps, using $\Theta(h)$ space.

Proof. We construct the logically reversible tagged CRN of Figure 5.4 adding reactions (6), (7), and (8), for each $1 \leq i \leq h$. As each reaction of the CRN is reversible, after every reaction step, the reverse of the previous reaction can always be applied. To demonstrate the CRN is logically reversible, we need to demonstrate that at any point there is at most one other reaction that can be applied. We will further establish the invariant that each reaction strictly alternates in being applied in the forward and reverse direction, ensuring at most one tag is required for each type of reaction. We will argue by structural induction. Let s_h denote the number of reaction steps required for a tree of height h.

(6) *mark left*

$$x_i^F + r_i^? + \bigcup_{j=1}^{i-1}\{r_j^\checkmark\} \rightleftharpoons l_i^? \rightleftharpoons l_i^\checkmark \quad , 1 \le i \le h$$

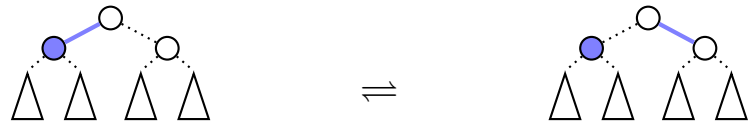


(a) recursively solve tree rooted at l_i

(b) mark l_i

(7) *move right*

$$l_i^\checkmark + r_i^? + \bigcup_{j=1}^{i-1}\{l_j^?\} \rightleftharpoons x_i^F \rightleftharpoons x_i^T \quad , 1 \le i \le h$$

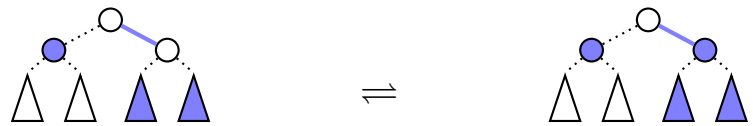


(c) reverse all steps from (a)

(d) move to right sub-tree

(8) *mark right*

$$r_{i+1}^? + l_i^\checkmark + x_i^T + \bigcup_{j=1}^{i-1}\{r_j^\checkmark\} \rightleftharpoons r_i^? \rightleftharpoons r_i^\checkmark \quad , 1 \le i \le h$$



(e) recursively solve tree rooted at r_i

(f) mark r_i

Figure 5.4: A logically reversible post-order traversal of all descendants of the root of a height h perfect binary tree can be achieved using three reactions: (6) *mark left*, (7) *move right*, and (8) *mark right*. Below each reaction is an illustration of the action it performs on the tree.

Consider the base case when $h = 1$ with initial signal multiset $\{r_2^?, x_1^F, l_1^?, r_1^?\}$. Reaction (8) cannot be applied until the signal x_1^T is present which is produced by reaction (7). Similarly, reaction (7) cannot be applied until signal $l_1^✓$ is present. Thus, it is easy to see that reaction (6) must first be applied—marking the left child—followed by reaction (7)—moving to the right child—and finally reaction (8)—marking the right child and completing the traversal in $s_1 = 3$ reaction steps. Each reaction was only applied once, in the forward direction, so the strictly alternating invariant is trivially maintained.

Suppose the traversal completes in s_{h-1} reactions steps, is logically reversible, and the strictly alternating invariant is maintained for a tree of height $h - 1$. Consider a tree of height h, having initial multiset of signals $S = \{r_{h+1}^?\} \cup \bigcup_{1 \leq i \leq h} \{x_i^F, l_i^?, r_i^?\}$. Before reaction (6) (and thus reaction (7) and (8)) can be applied, the signals $\bigcup_{1 \leq j < h} \{r_j^✓\}$ must be present. As the left subtree is selected, the signal $r_h^?$ is present, and by the induction hypothesis, the only available action is to produce these signals in s_{h-1} logically reversible reaction steps, that maintain the strictly alternating invariant, by traversing the subtree rooted at l_h (see Figure 5.4 (a)). Importantly, the signals $\bigcup_{1 \leq j < h-1} \{r_j^?\}$ are now absent and therefore no reaction affecting levels $1, \ldots, h-2$ can occur. Other than reversing the previous reaction, which produced signal $r_{h-1}^✓$, only reaction (6) can be applied for level h, thus producing $l_h^✓$ (see Figure 5.4 (b)). Next, observe that reaction (7) cannot be applied until the signals $\bigcup_{1 \leq j < h} \{l_j^?\}$ are present. Other than reversing the previous reaction, only a reversal of all s_{h-1} reaction steps that traversed the left subtree can be applied next, yielding the required signals to next apply reaction (7), producing signal x_h^T, denoting a move to the right subtree (see Figure 5.4 (c) and (d)).

Note that the reversal of the left subtree will maintain the strictly alternating invariant as it ensures all lower level reactions have been reset to their initial state, in order to be used again in the right subtree. Similar to reaction (6), reaction (8) cannot be applied at level h until the right subtree is traversed in s_{h-1} logically reversible reaction steps (see Figure 5.4 (e)). Other than reversing the previous reaction, only reaction (8) can next be applied at level h producing the signal $r_h^✓$ and ensuring no further reactions on lower levels can occur. The traversal is complete and no reaction, other than the reverse of the previous, can occur. Thus, the overall traversal is logically reversible, and is clearly in post-order. As the strictly alternating invariant was maintained for all lower level reactions, and all reactions at level h have been applied for the first time, and only once, the invariant is maintained for a tree of height h.

Exactly 3 reactions occurred at level h, and $3s_{h-1}$ reactions were required for the two traversals and one reversal of the height $h - 1$ subtrees, giving us the recurrence $s_h = 3s_{h-1} + 3$. Solving s_h with $s_1 = 3$ gives us the closed form expression $\frac{3}{2}(3^h - 1)$, establishing the claimed $\Theta(3^h)$ reaction steps.

Finally, consider the space claim. As we have shown that reactions strictly alternate being applied in the forwards and reverse direction, at most one tag for each of the $\Theta(h)$ reactions is sufficient. Consider that the initial multiset of signals for a tree of height h is $S = \{r_{h+1}^?\} \cup \bigcup_{1 \leq i \leq h} \{x_i^F, l_i^?, r_i^?\}$ and therefore

$|S| = 3h+1$. Since the CRN is proper, we immediately establish the space claim by Lemma 1. □

5.3.3 Solving a Q3SAT instance

We now have the means to verify if a variable assignment satisfies a 3SAT formula ϕ. We can also traverse a perfect binary tree in post-order, and in the process enumerate all possible variable assignments for ϕ. What remains is to combine these processes together in order to determine if a Q3SAT instance can be satisfied. We approach the integration in two parts. First, we will demonstrate how the formula verification process can be triggered immediately prior to the tree-traversal marking of a leaf node and how the verification reactions can be entirely reversed, prior to the next time the verification procedure must run. This effectively demonstrates how any problem in NP can be solved by a logically reversible CRN in polynomial space, if we specify the end of computation as the presence of the signal ϕ^T, or the signal ϕ^F in conjunction with the signals for the final variable assignment to be enumerated. Finally, we demonstrate how the tree traversal reactions of Figure 5.4 can be augmented in order to capture the semantics of alternating universal and existential quantifiers, thus demonstrating how any problem in PSPACE can be solved in polynomial space by a logically reversible CRN.

Integrating formula verification and tree traversal

Recall the sequence of logical steps in traversing level 1 of the tree, *i.e.,* the leaves: *mark left* leaf, *move right*, *mark right* leaf. We augment the reactions for level 1 to force the following sequence: (i) *verify* ϕ, (ii) *mark left* leaf, (iii) *reverse reactions of step (i)*, (iv) *move right*, (v) *verify* ϕ, (vi) *mark right* leaf. This new sequence ensures two invariants: first, the current variable assignment is verified prior to marking the current leaf, and second, the verification procedure is fully reversed prior to the next verification.

The augmented reactions are given in Figure 5.5. Both reactions marking a leaf have been split into two variants, each ensuring the verification procedure has completed by requiring as a catalyst one of the two possible outcomes of the verification process. In addition, we add new signals to record whether or not the variable assignment for a particular leaf is a satisfying assignment for ϕ. These signals will be used later to propagate satisfiability up the tree, once quantifiers have been integrated. Note that the reaction to move to the right leaf now requires the signal $\phi^?$ as a catalyst. This forces all steps performed in the previous verification to reverse. After moving to the right leaf, and thus swapping the value of variable x_1, the verification process can again run immediately prior to marking the right leaf. Importantly, we want to ensure that the verification procedure is completely integrated into the leaf level reactions and cannot perform any reactions while the traversal is marking higher level nodes. This is easily accomplished by augmenting reactions (2)-(5) to require $r_2^?$ as a catalyst. Note that the augmented variants of the tree traversal reactions

are also fully distinguishable by their catalysts (and products), thus ensuring the process is logically reversible.

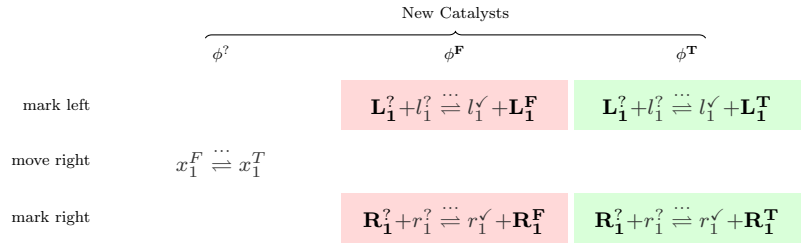


Figure 5.5: Integrating the 3SAT verification procedure into the leaf level reactions of the tree traversal procedure. Two reaction variants are created for marking leaf nodes as either satisfied or unsatisfied based on the result of the verification procedure. One reaction variant can proceed if the signal ϕ^F is available and the other variant requires ϕ^T. As these are the only two reaction variants, the formula for the current variable assignment must be verified before the leaf node can be marked. The *move right* reaction requires $\phi^?$ as a catalyst, thus ensuring the verification procedure is reversed prior to the next verification step. Existing catalysts listed in Figure 5.4 remain and are omitted above for space.

Integrating quantifiers into the tree traversal

Integrating quantifiers in non-leaf levels of the tree is relatively straightforward. Recall that the levels of the tree strictly alternate between universal and existential quantification. For each level, we create four variants of the correct quantifier for both the left and right node marking reactions to additionally produce a signal indicating if the current subtree is satisfied. The reaction variants for marking a left node are given in Figure 5.6. These reactions require as catalysts the signals indicating if the left and right children of the current node are satisfied and therefore four variants are sufficient to consider all cases, for each type of quantifier. As with the leaf level reactions, the augmented reaction variants can be fully distinguished by their catalysts ensuring the computation remains logically reversible, and the correct reactions are reversed.

Ending the computation

Once both children of the root have been solved the output signal can be produced based on the satisfiability of the children and on the quantifier imposed on the root level variable x_n. The reaction equations for the universal quantifier are shown in Figure 5.7. Modifying the reactions for an existential quantifier is straightforward.

Recall that reactions at level $n-1$ cannot proceed unless the signal $r_n^?$ is present. We could have the reaction producing the solution signal also con-

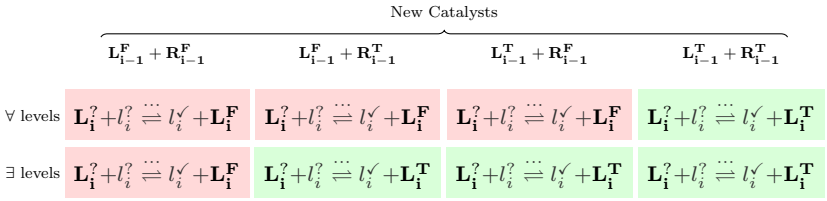


Figure 5.6: Integrating quantifiers to non-leaf levels of the tree traversal. For both universal and existential levels, four variants of the left node reactions are created to process the four combinations of left and right children satisfiability. The integration is identical for right node reactions. Existing catalysts remain the same as listed before and are omitted for space.

sume $r^?_n$. This would end the computation chain as only reversing the previous reaction would be possible next. However, for reasons we will make clear in Section 5.5, the signal $r^?_n$ is never altered and therefore after the solution signal is produced, the entirety of the tree traversal steps will be reversed before reaching the end of the computation chain. The entire configuration of the CRN system will appear identical to the initial configuration, with the exception that the output has been written (*i.e.,* the $\psi^?$ signal has been consumed and been replaced by ψ^F or ψ^T). See Figure 5.8 for a schematic of the logically reversible computation chain.

(9)
$$\begin{aligned}
\psi^? &\overset{L^F_{n-1}+R^F_{n-1}+\bigcup_{j=1}^{n-1}\{r^✓_j\}}{\rightleftharpoons} \psi^{\mathbf{F}} \\
\psi^? &\overset{L^F_{n-1}+R^T_{n-1}+\bigcup_{j=1}^{n-1}\{r^✓_j\}}{\rightleftharpoons} \psi^{\mathbf{F}} \\
\psi^? &\overset{L^T_{n-1}+R^F_{n-1}+\bigcup_{j=1}^{n-1}\{r^✓_j\}}{\rightleftharpoons} \psi^{\mathbf{F}} \\
\psi^? &\overset{L^T_{n-1}+R^T_{n-1}+\bigcup_{j=1}^{n-1}\{r^✓_j\}}{\rightleftharpoons} \psi^{\mathbf{T}}
\end{aligned}$$

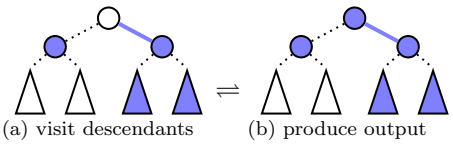


Figure 5.7: After both children of the root have been solved a solution can be determined based on the quantifier of the root level. Equations are shown assuming the root variable x_n is universally quantified.

Theorem 16. Any arbitrary instance of Q3SAT with n variables and m clauses can be *solved* by a logically reversible tagged CRN in $O(m\,3^n)$ reaction steps using $\Theta(m+n)$ space.

Proof. Let ψ be the totally quantified Boolean formula of the instance and ϕ be the unquantified 3SAT formula. By Lemma 16 a set of $\Theta(m)$ reactions can be created to verify if ϕ is satisfied, or not, for a particular variable assignment. By Lemma 17, a set of $\Theta(n)$ reactions can be created to traverse the height n tree

representing all possible assignments of the n variables. Furthermore, the above modifications demonstrate how these two processes can be integrated into one logically reversible computation chain, and how quantifiers can be added to the non-leaf levels to determine if there is a satisfying solution for ψ by propagating satisfiability of subtrees up to higher levels. Importantly, the modifications only increase the number of reactions by a constant factor and are designed to maintain the property that the computation is logically reversible. Consider that the number of reaction steps acting on a tree node, prior to reaching the root, has not increased. However, prior to marking every leaf in the traversal, the verification procedure is run for the current variable assignment (and reversed in between). Therefore, by Lemmas 16 and 17, the root of the height n tree can be reached, and a solution signal produced, within $O(m\,3^n)$ reaction steps. As forcing the entire tree traversal to reverse prior to the end of computation only doubles the number of reaction steps, the claim on computation length is established.

Next, consider the space required of the combined CRN. The modified verification procedure requires the following initial multiset, where $\mathcal{T}_{3sat}$ is the multiset of required tags:

$$\mathcal{S}_{3sat} = \bigcup_{1\leq i\leq m} \left\{C_i^?, H_i^?\right\} \cup \left\{\phi^?, r_2^?\right\} \cup \mathcal{T}_{3sat}$$

The augmented tree traversal procedure requires the following initial multiset, where $\mathcal{T}_{tree}$ is the multiset of required tags:

$$\mathcal{S}_{tree} = \bigcup_{1\leq i<n} \left\{l_i^?, x_i^?, r_i^?, L_i^?, R_i^?\right\} \cup \left\{r_n^?, \phi^?, \psi^?\right\} \cup \mathcal{T}_{tree}$$

The space required for the initial multiset of the combined CRN is therefore $|\mathcal{S}_{q3sat}| = |\mathcal{S}_{tree} \cup \mathcal{S}_{3sat}|$. As the combined CRN maintains the property that reactions strictly alternate being applied in the forward and reverse direction, then one tag for each of the $\Theta(m+n)$ reactions is sufficient and $|\mathcal{S}_{q3sat}| \in \Theta(m+n)$. □

As Q3SAT is a complete problem for PSPACE [94], we immediately have the following.

Corollary 2. Any problem in PSPACE can be *solved* by a logically reversible tagged CRN using polynomial space.

5.4 Space efficient CRN simulation of SPACE

We have so far shown how to simulate any problem in PSPACE with a space-efficient CRN. In this section, we extend our result to show how any $S(n)$ space-bounded computation can be simulated by a logically reversible tagged CRN using at most $poly(S(n))$ space.

The first result that we leverage, summarized in Theorem 17, states that any $S(n)$ space-bounded computation, with an input of size $n \leq S(n)$, can be simulated by an alternating Turing machine in $O(S(n)^2)$ steps.

Theorem 17 (Chandra, Kozen and Stockmeyer [18])**.** If $S(n) \geq n$, then $\mathsf{NSPACE}(S(n)) \subseteq \bigcup_{c>0} \mathsf{ATIME}(c \cdot S(n)^2)$.

We will also make use of the following transformation, from a nondeterministic Turing machine to a propositional formula, due to Cook [28].

Theorem 18 (Cook [28])**.** Let M be a $T(n)$ time-bounded nondeterministic Turing machine. For each input x there is a conjunctive normal form propositional formula $F(x)$ of length $\Theta(T(n) \log T(n))$, *[containing at most three literals per clause and that can be produced in poly(n) time]*, such that $F(x)$ is satisfiable if and only if M accepts x within $T(n)$ steps.

As communicated by Williams [140], this same transformation can be used to derive a quantified Boolean formula ψ of length $\Theta(T(n) \log T(n))$, for an alternating Turing machine M, such that ψ is satisfiable if and only if M accepts its input within $T(n)$ steps. Intuitively, the universal states of M are represented by universal variables, while existential states are represented by existential variables. Importantly, Cook's reduction ensures that the unquantified propositional formula is in conjunctive normal form and that clauses have at most three literals. Without loss of generality, we can assume that the quantified formula ψ is in prenex normal form and that existential and universal quantifiers strictly alternate[21].

We summarize this result due to Theorems 17 and 18 in Corollary 3.

Corollary 3 (Chandra, Kozen and Stockmeyer [18] & Cook [28])**.** Given any $S(n)$ space bounded Turing machine M and an input x of length n, with $S(n) \geq n$, it is possible in *poly*$(S(n))$ time to construct a totally quantified Boolean formula ψ, in prenex normal form, having $\Theta(S(n)^2 \log S(n))$ clauses in conjunctive normal form, where each clause contains at most three literals (*i.e.,* ψ is an instance of Q3SAT), such that ψ is satisfiable if and only if M accepts x within $S(n)$ space.

In Corollary 3 there is a condition that the space complexity is at least as large as the size of the input. When this is not case, such problem instances are necessarily in PSPACE and therefore our result of Corollary 2 applies. Otherwise, by Theorem 16 and Corollary 3 we can conclude the following result.

Corollary 4. Any problem solvable in $S(n) \geq n$ space can be solved by a logically reversible tagged CRN using $O(S(n)^2 \log S(n))$ space.

[21] Our CRN construction works without the assumption that quantifiers strictly alternate, but it simplifies the presentation of the result to assume that they do.

5.5 Space and energy efficient DSD simulation of SPACE

The remarkable consequence that Bennett's work demonstrates is that energy consumption is not necessarily an intrinsic cost of computation. In particular, if the computation is logically reversible, there is no inherent lower bound on energy expenditure, due to the computation. However, there must be a reasonable probability the actual solution can be observed. This can be problematic in a logically reversible computation which is free to immediately reverse once reaching a solution state. Qian et al. [98] solved this problem by using fuel (transformers) to provide a slight bias for remaining in a solution state once the computation completes. However, in our result, since reactions must be reused efficiently in both directions to maintain a polynomial space bound, they cannot be biased in general.

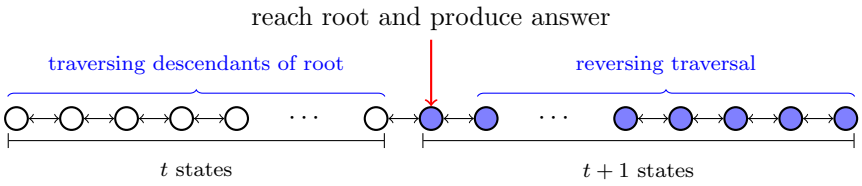


Figure 5.8: The logically reversible computation chain of the Q3SAT CRN. In more than half of the states, the output signal is present (shown shaded).

To overcome this, we have designed our reactions that produce an output signal to ensure the next logical step in the computation is to reverse the tree traversal. This effectively doubles the length of the logically reversible computation chain and established the important property that the output signal can be observed in strictly more than half of the states (see Figure 5.8). Notice that this was also the case for Bennett's original reversible Turing machine implementation[22]. As the computation performs an unbiased random walk along the logically reversible computation state space, the steady state probability of observing the output signal is $p > 0.5$.

This probability can be further increased in a number of ways. For instance, by adding one additional reaction that produces a new signal and requires the final signal multiset of the original computation chain as catalysts, we can once again double the number of reactions in the new chain. In this case, the probability of observing the output signal is $p' > 0.75$. In this manner, for every new reaction added to the CRN, the probability of not observing an output signal is cut in half. Formally, the probability of observing an answer becomes $p'' > 1 - 2^{-(1+c)}$ when $c \geq 0$ number of new reactions are added to extend the computation chain. Thus, we can make the steady state probability of observing

[22]The forward traversal of the tree, production of the output signal, and reversal of the traversal are analogous to the *compute*, *copy output*, and *retrace* phases of Bennett's original reversible Turing machine simulation [8].

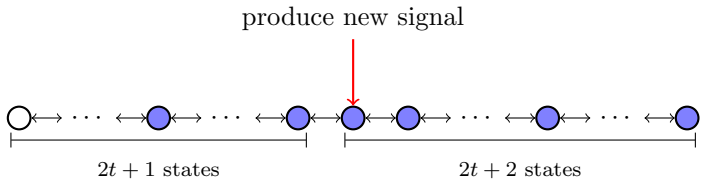


Figure 5.9: Extending the logically reversible computation chain of the Q3SAT CRN. Extending the chain is achieved by adding an additional reaction that produces a new signal and requires the final signal multiset of the original computation chain as catalysts. States where the output signal is present are shown shaded.

a solution signal arbitrarily high. Furthermore, at the DSD level, we could design the gates which implement the reaction producing the output signal to have a slight bias in the forward direction, by manipulating relative toehold lengths, effectively biasing our overall computation towards the end of the chain containing an answer shown in Figure 5.8 [142]. As this reaction is only performed once, the gate implementing the reaction is not reused and therefore, the bias is not problematic for the overall computation to complete.

We note that the CRN and DSD description given here is a non-uniform model of computation. Specifically, the CRN description is dependent on, and encodes, a particular problem instance. Therefore, different problem instances will result in different CRN descriptions and thus a different DSD implementation. Particularly at the DSD level, where synthesizing strands and gates is challenging, it would be desirable if only the input strands differed between unique instances. This can be achieved by constructing a more general quantified Boolean formula that is within a polynomial size of the original encoding described here. In such a construction, part of the input would describe which clauses are active for the particular problem instance. The generalized formula would be for a fixed number n of variables and could be used to solve any instance having at most n variables[23].

While the computation chain can be extended to increase the probability of observing the output to some fraction of the total length of the chain, as currently described, there is only one position on the chain (the initial position) where the input can be changed. Changing the input signals in the middle of the chain would mean the computation is no longer logically reversible—the chain would be missing necessary signals to reverse. This can be overcome by extending the number of states where the input can change, without issue, to be a constant fraction of the entire chain length [123]. In particular, we can use the n-bit GRAY counter implementation from Chapter 4 to form the first fraction of states of the overall chain. The reactions of the counter are orthogonal to the reactions of the Q3SAT solver and thus the input can be changed at any position when the counter is active. We would make the first reaction of the

[23]This was suggested in an anonymous review of an earlier version of this work [131]

Q3SAT solver consume the high order bit of the GRAY counter, which is only produced by its last reaction. Thus, the overall chain is logically reversible, the input can be written during a constant fraction of the chain length, and the output can be read for a constant fraction of the chain length [123].

Combining Theorem 12, Corollary 2 and Corollary 4 we have the following.

Theorem 19. Any problem in PSPACE can be *solved* by a space and energy efficient DSD.

5.6 Complexity of verifying CRNs and DSDs

Next we show there exists a polynomial time and space reduction from an arbitrary Q3SAT instance I into an instance I' of the CRN reachability problem (Q3SAT $\leq_p$ CRNR), such that I can be solved if and only if I' can be solved.

Theorem 20. The reachability problem for CRNs (CRNR) is PSPACE-hard.

Proof. Given an arbitrary Q3SAT instance, construct the CRN of Theorem 16 which is of polynomial size and can be constructed in polynomial time and space by following the steps described in our construction. Ask the question of whether the state $S_{init}/\{\psi^?\} \cup \{\psi^T\}$ can be reached from S_{init}, where S_{init} is the initial state of the CRN. □

By Lemma 1 it is easy to see the reachability problem for proper CRNs is in PSPACE. Whether other forms of CRNs are in PSPACE is dependent on their definition and how the required space to complete a computation is accounted for. Any tagged CRN accounts for the necessary transformers as part of the size of the reaction volume and therefore, by this interpretation, is in PSPACE.

Corollary 5. The reachability problem for proper/tagged CRNs is PSPACE-complete.

We note that other results are known for unrestricted CRNs which are not studied here. (Un-tagged) reversible CRNs correspond to reversible Petri nets where the reachability problem is EXPSPACE-complete [16]. CRN reachability has also been studied for the probabilistic case [121, 148] and nondeterministic case [148] and the connection with Petri nets was also explored [148].

By Theorem 12 and Theorem 20 we immediately have the following analogous results for DSDs.

Corollary 6. The reachability problem for DSDs (DSDR) is PSPACE-hard.

Clearly the reachability problem is PSPACE-complete for the set of DSDs implementing a proper CRN. When transformer (fuel) molecules are considered part of the space usage, as would be the case for closed volumes that are studied here (*i.e.,* tagged CRNs), then the reachability problem is PSPACE-complete.

5.7 A reduction from Q3SAT to EB-IPFP-MULTI

We next show that a DSD instance created by the above chain of reductions (*i.e.,* Q3SAT $\leq_p$ CRNR $\leq_p$ DSDR) can be adapted to show the minimum energy barrier indirect folding pathway problem for multiple interacting strands is PSPACE-complete when using the simple energy model defined in Section 1.1.1.

5.7.1 The reduction

We begin with an arbitrary instance ψ of the Q3SAT problem consisting of n variables and m clauses. By Theorem 16, we can construct a CRN of size $\Theta(m+n)$ that will output a special acceptance signal if and only if ψ is satisfiable. By Theorem 12, this CRN can be implemented by a DSD that uses $poly(m+n)$ space. Thus, the DSD will produce a special signal strand we call s_{yes} if and only if ψ is satisfiable.

Now, there are two issues we must address. First, for the folding pathway problem, we must reason at the sequence level and not at the abstract domain level specified by a DSD. Second, there is an assumption in the DSD construction that only *legal* toe-hold mediated strand displacements occur. We must design our sequences to ensure this assumption is maintained in order to conclude any meaningful result. To simplify our argument, we will use a modified version of the QSW construction (*i.e.,* the DSD construction of Theorem 12). The modification is straightforward: for every long domain in the original construction that is initially free (unbound), bind it to a new complementary strand (see Figure 5.10). Note that the resulting DSD still uses space $poly(m+n)$. This modified construction has two effects. First, a legal displacement of a strand now requires four-way branch migration (described below), in contrast to three-way branch migration required in the original construction. Second, prior to any displacement and immediately after any sequence of legal displacements, all long domains are fully bound to a complementary domain. This latter property greatly simplifies our correctness argument.

A legal toe-hold mediated strand displacement (legal displacement) that uses four-way branch migration involves four strands: an *invading* strand, a *complementary* strand bound to the displacing long domain of the invading strand, a *template* strand with a free toehold, and an *evading* strand currently bound to the template strand. The process can be summarized in five steps: (i) the invading strand / complementary strand complex associates to the complex containing the template strand by forming a first base pair, (ii) additional base pairs are formed between the free toehold of the template strand and the toehold complement domain on the invading strand, (iii) a long domain of the invading strand displaces an identical long domain of the evading strand using four-way branch migration, where the evading strand forms base pairs with the complementary strand as the invading strand forms base pairs with the template strand, (iv) all but one of the toehold base pairs of the evading strand are broken, and (v) the evading strand, now fully bound to the complementary strand, breaks the last base pair and disassociates from the complex containing the

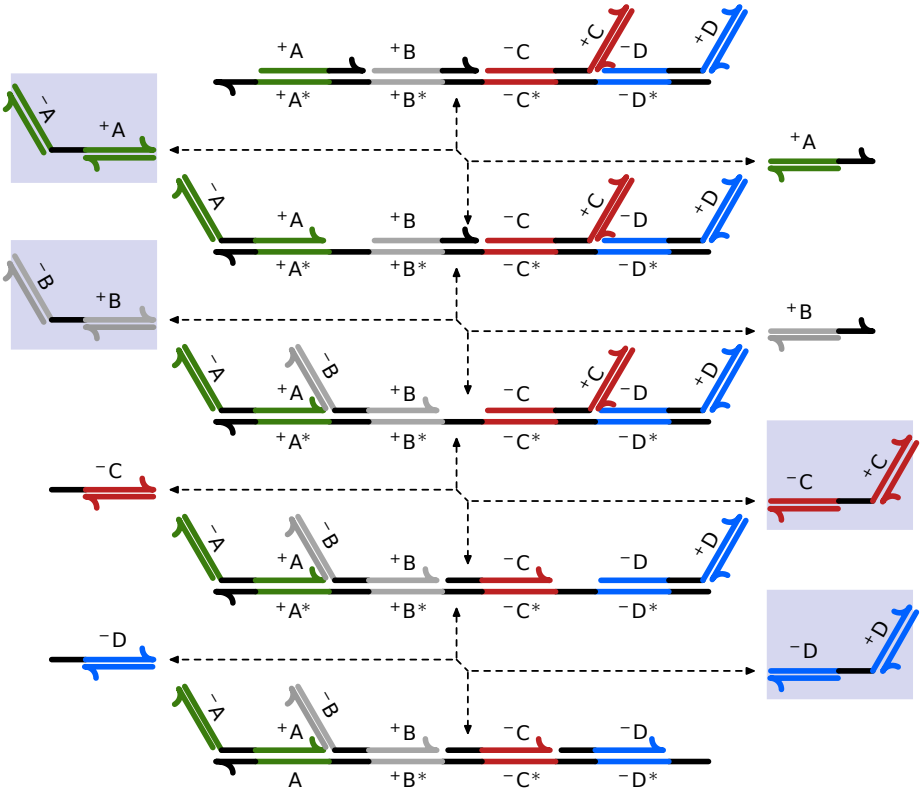


Figure 5.10: A strand displacement implementation of the bi-molecular chemical reaction equation $A + B \rightleftharpoons C + D$ using a modified construction from that proposed by Qian et al. [98]. In this construction, four-way branch migration is used to displace strands, in contrast to three-way branch migration from the original construction.

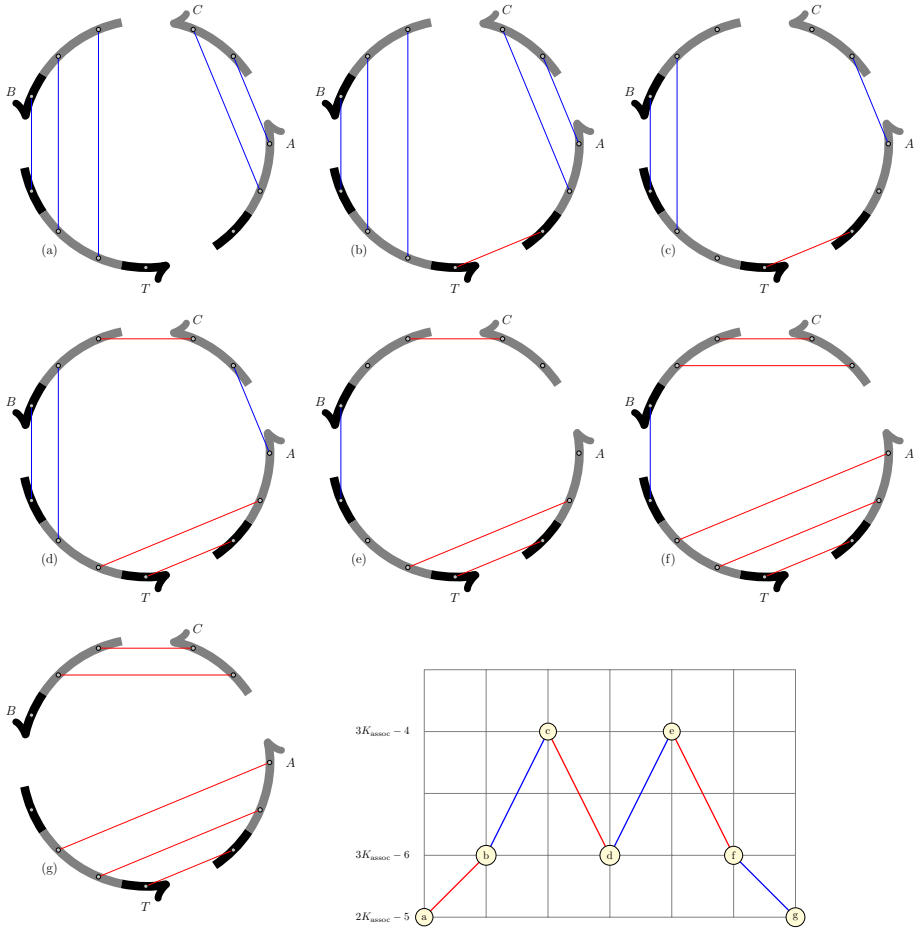


Figure 5.11: A folding pathway is shown for a strand displacement using four-way branch migration. A simple sequence design is assumed where toehold domains have one base and long domains have two bases. The displacement of strand B by strand A is shown in seven steps, from (a) to (g). Initially, the long domain of A is bound to strand C. During the displacement, C will form base pairs with B while A forms base pairs with T. In the figure, base pairs are shown as edges between strands. The energy changes between each structure, assuming $K_{\text{assoc}} = 2$, are shown in the bottom right. The energy barrier of the underlying folding pathway, relative to (a), is $K_{\text{assoc}} + 1$. Note that for toehold length $L_{\text{T}} > 2$, where $K_{\text{assoc}} > L_{\text{T}}$, the energy barrier would be $K_{\text{assoc}} - 1$.

template strand. The main difference with three-way branch migration is that the complementary strand forms base pairs with the evading strand whenever possible. An example folding pathway for a legal displacement using four-way branch migration is given in Figure 5.11.

Now let us consider how an evading strand can be displaced (*i.e.,* produced) from a template other than by a legal displacement. In the simplest case, it is possible that the evading strand simply breaks all base pairs with the template strand and disassociates. We call this a *spontaneous* displacement. Now suppose one or more other strands, that possibly have base pairs with one or more other complexes, are used to perform the illegal displacement. We partition this possibility into two cases. In the first case, suppose all the invading strands have a different long domain than the evading strand. We call this a *mismatch* displacement. In the second case, suppose at least one of the invading strands does have a long domain that is the same as the evading strand, but it is either the case that there are no free toeholds adjacent to the evading strand, or if there are, all invading strands with the correct long domain have the toehold on the wrong side. We call this *blunt-end* displacement. Note that if there were at least one free adjacent toehold, and at least one of the invading strands with the correct long domain had a toehold on the correct side, then it could simply perform a legal displacement. Thus, these three cases cover all possible events for an illegal strand displacement.

We begin by designing the sequences. Let all domains on template strands, inclusive of toehold domains, and all domains on complementary strands be formed of sequences using the bases `T` and `G`. Let all other strands be formed of sequences using the bases `A` and `C`. Thus, it is not possible for *intra*-strand base pairs to form[24]. Suppose all toeholds on template strands use the same sequence and are of identical length L_{T}, with $K_{\mathrm{assoc}} > L_{\mathrm{T}} > 2$; recall that K_{assoc} is the entropic penalty for each strand association that results in fewer strand complexes. Therefore all toehold complement domains on other strands also have identical sequences (the complement of the common toehold sequence) and have length L_{T}. Further suppose all long domains and complement strands have a common length $L_{\mathrm{L}} > 2K_{\mathrm{assoc}} > L_{\mathrm{T}} > 2$. Create sequences for long domains, in polynomial time, such that distinct domains have an edit distance of at least $2K_{\mathrm{assoc}}$ bases[25]. Let B be the *baseline* energy of the initial DSD, prior to any displacements, using any sequence design satisfying the above constraints. We now argue formally that this reduction will result in a folding pathway that can displace the acceptance strand s_{yes} within barrier $K_{\mathrm{assoc}} - 1$, relative to the baseline energy B, if and only if the Q3SAT instance ψ is satisfiable.

Lemma 18. From a particular configuration having energy B, with all template strands saturated, a *legal* displacement can complete within barrier $K_{\mathrm{assoc}} - 1$,

[24]The simple energy model assumes only Wastson-Crick base pairs can form (*i.e.,* `A-T` and `C-G`).

[25]Many trivial code word designs are sufficient for this purpose. For example, assign a unique multiple of $2K_{\mathrm{assoc}}$ `T` (`A`) bases to long domains on the template and complementary (other) strands. Such a code word design has polynomial size and is trivially created in polynomial time.

resulting in a new configuration with all template strands saturated, having energy B.

Proof. Let I, C, E, and T be the invading strand, complementary strand, evading strand, and template strand, respectively. The association of I to T decreases the number of complexes by one, and increases the number of base pairs by one, therefore increasing the energy to $B + K_{\text{assoc}} - 1$. As toeholds have length $L_{\text{T}} > 1$, then $L_{\text{T}} - 1$ new base pairs can immediately form, resulting in energy $B+K_{\text{assoc}}-L_{\text{T}}$. Identical long domains of I and E will perform four-way branch migration (T and C being the third and fourth strand). Thus, as one base pair between E and T is broken and another between C and I is broken, raising the energy to $B+K_{\text{assoc}}-L_{\text{T}}+2$, a new base pair between I and T and another between E and C can form next, lowering the energy to $B+K_{\text{assoc}}-L_{\text{T}}$. This oscillation of two base pair differences happens for each of the L_{L} bases in the common long domain of I and E. Once the common long domain of E is displaced (and C is fully bound to E), $L_{\text{T}} - 1$ toehold base pairs bonding E to T can break, raising the energy to $B + K_{\text{assoc}} - 1$. As strand E disassociates from the template, the number of complexes increases by one, and the number of base pairs decreases by one, lowering the energy back to B. The original free toehold of T is now paired to I. However, the toehold previously paired with E is now free. All other bases on T are paired and therefore T is saturated. As $L_{\text{T}} > 2$, then the highest energy of any configuration during displacement is $B + K_{\text{assoc}} - 1$, achieving the claimed energy barrier $K_{\text{assoc}} - 1$. □

To simplify the remaining argument, the combination of the next two lemmas show that any displacement of a strand within the prescribed energy barrier must be by exactly one additional displacing complex. That is to say, no combination of multiple complexes can cooperate to displace a strand from another complex, nor can the strands of the same complex be used, possibly with additional complexes, to displace a strand without exceeding the energy barrier.

Lemma 19. From a particular configuration having energy B, with all template strands saturated, a displacement involving more than two complexes cannot complete within barrier $K_{\text{assoc}} - 1$.

Proof. Since the template strand that binds the evading strand is saturated, then exactly one toehold of length L_{T} is unbound. Therefore, after the first complex associates with the complex containing the template strand, and leaving open at least one base for a second association (on either of the original complexes), the minimum possible energy is $B + K_{\text{assoc}} - L_{\text{T}} + 1$. When a second complex associates, the minimum possible energy cannot be lower than $B + 2K_{\text{assoc}} - L_{\text{T}} > B + K_{\text{assoc}} - 1$ as $K_{\text{assoc}} > L_{\text{T}}$. □

Lemma 20. From a particular configuration having energy B, with all template strands saturated, a displacement from a template strand using other strands from the same template complex cannot complete within barrier $K_{\text{assoc}} - 1$.

Proof. First, we note that any domain from the same complex as the template used in such a displacement could only be a long domain that is bound to a complementary strand (*i.e.,* not a domain bound to the template strand as the template domain it is bound to would need to form base pairs with the evading strand and it could therefore not displace). Next, we note that such a long domain must be one bound adjacent to the strand to be displaced; otherwise, a pseudoknot would occur as it must cross (cover) at least one domain of another strand bound to the template. For example, in the top of Figure 5.10 only the long domain for $+C$ bound to its complement could displace the long domain $-D$ without creating a pseudoknot. Furthermore, a pseudoknot must also form if the strand being displaced interacts with adjacent strands on both sides (one must cross its own long domain bound to the template strand). Therefore, consider the case of an adjacent long domain bound to a complement strand that could displace the evading strand without creating a pseudoknot. Next, consider that by the sequence design the edit distance with the domain to be displaced is at least $2K_{\text{assoc}}$, thus any displacement using only the adjacent long domain and its complemenatary strand would result in an energy barrier of at least $2K_{\text{assoc}}$. As argued above, no other strands from the same complex could cooperate without creating a pseudoknot. Suppose one other complex is used to cooperate in the displacement. In this case, it has at most L_{T} free bases that could be used to lower the energy (using already paired bases cannot lower the energy barrier). However, as the association cost is $K_{\text{assoc}} > L_{\text{T}}$, then a second complex cannot be used to overcome the energy difference. Similarly, and consistent with Lemma 19, more than one additional complex cannot be used to lower the energy barrier. □

Lemma 21. From a particular configuration having energy B, with all template strands saturated, a *mismatch* displacement cannot complete within barrier $K_{\text{assoc}} - 1$.

Proof. Suppose to the contrary that a mismatch displacement can complete within barrier $K_{\text{assoc}} - 1$. By definition of a mismatch displacement, one or more invading strands are used, but the long domains used for displacement all differ from the long domain of the evading strand. By Lemmas 19 and 20, we need only consider the case of a single invading complex without the cooperation of strands in the same complex as the evading strand. For the L_{L} base pairs broken for the evading strand, the invading strand can form at most $L' \leq L_{\text{L}} - 2K_{\text{assoc}}$ new base pairs due to the sequence design constraints. Therefore, just prior to the evading strand removing its toehold base pairs, and assuming the invading strand formed base pairs to all of its toehold complement domain, the energy will be $B + K_{\text{assoc}} - L_{\text{T}} + L_{\text{L}} - L' \geq B + K_{\text{assoc}} - L_{\text{T}} + L_{\text{L}} - (L_{\text{L}} - 2K_{\text{assoc}}) = B + 3K_{\text{assoc}} - L_{\text{T}}$. However, $B + 3K_{\text{assoc}} - L_{\text{T}} > B + K_{\text{assoc}} - 1$ as $K_{\text{assoc}} > L_{\text{T}}$. Contradiction. □

Lemma 22. From a particular configuration having energy B, with all template strands saturated, a *blunt-end* displacement cannot complete within barrier $K_{\text{assoc}} - 1$.

Proof. Suppose to the contrary that a blunt-end displacement can complete within barrier $K_{\text{assoc}}-1$. Note that by Lemma 19 and 20, increasing the number of invading strands or cooperatively using strands of the same complex as the evading strand cannot help. In a blunt-end displacement, by definition, branch migration of a long domain is not preceded by the formation of base pairs in an adjacent unbound toehold. Both the case of the invading strand binding to a toehold which is not adjacent the evading strand, or the case of it binding to a toehold that is adjacent but on the wrong side (as expected in a legal displacement), can be ruled out as a pseudoknot would form. Since the template strand is saturated, then it must be the case that an existing base pair, involving a base on the template strand, must first break prior to the invading stranding forming its first base pair, raising the energy to $B+1$. When the invading strand associates the number of complexes decreases by one, and the number of base pairs increases by one, thus raising the energy to $B + K_{\text{assoc}}$. Thus, the energy barrier is at least K_{assoc}. Contradiction. □

Lemma 23. From a particular configuration having energy B, with all template strands saturated, a *spontaneous* displacement cannot complete within barrier $K_{\text{assoc}} - 1$.

Proof. Suppose to the contrary that a spontaneous displacement can complete within barrier $K_{\text{assoc}} - 1$. Since the length of the long domain on the evading strand is L_{L}, then the energy after all but the last base pair is broken is $B + L_{\text{L}} - 1 > B + K_{\text{assoc}} - 1$ as $L_{\text{L}} > K_{\text{assoc}}$. Thus, the energy barrier is at least K_{assoc}. Contradiction. □

Theorem 21. The EB-IPFP-MULTI problem, namely the energy barrier for indirect pseudoknot-free folding pathway of multiple interacting strands problem, is PSPACE-complete.

Proof. Using the reduction described above, and by Theorem 16 and Theorem 12, given an arbitrary instance ψ of Q3SAT having n variables and m clauses, a DSD can be constructed with the discussed modifications, in time $poly(m + n)$, with a fully specified nucleotide sequence that has $poly(m + n)$ total bases overall, such that: (i) all template strands of the DSD are initially saturated, and (ii) the DSD will produce a special signal strand, s_{yes}, through a sequence of legal strand displacements if and only if ψ is satisfiable.

Let B be the initial energy of the resulting DSD. We now show that any sequence of legal displacements follows an indirect folding pathway within energy barrier $K_{\text{assoc}} - 1$. This follows immediately from the construction (all templates are saturated) and by Lemma 18 as it guarantees each legal displacement is within the energy barrier and returns to the initial energy B (and all templates are again saturated). In the other direction, Lemmas 19–23 ensure that any sequence with at least one non-legal displacement must exceed the energy barrier $K_{\text{assoc}} - 1$. Therefore, the folding pathway is within energy barrier $K_{\text{assoc}} - 1$ if and only if the DSD follows a sequence of legal displacements. Thus, ψ is satisfiable if and only if the strand s_{yes} can be displaced within energy barrier $K_{\text{assoc}} - 1$. □

5.8 Chapter summary

In this chapter, we asked the question: can space *and* energy efficient computation be realized by chemical reaction networks (CRN) and DNA strand displacement systems (DSD)? We have shown this can be achieved in general by giving a logically reversible space efficient CRN implementation capable of solving any problem in PSPACE—the class of all problems solvable in polynomial space. Furthermore, our CRN can be realized by a space and energy efficient DSD. We have also shown how these results can be extended to solve any problem in SPACE. Thus, any computation that halts can be solved by a space and energy efficient DSD. The only other DSD implementation capable of solving any problem in SPACE is the stack machine implementation of Qian et al. [98]. The result of this chapter improves upon the stack machine implementation in terms of space efficiency, as the stack machine uses space proportional to computation length. However, our result falls short in a number of other respects when compared with the stack machine. Our construction provides a non-uniform model of computation, and thus, as currently described, a new CRN and thus DSD, must be created for each different problem instance that must be solved. It is conceivable that the result can be generalized to solve any problem instance, up to a particular size. The stack machine implementation is Turing universal. Since our result is based on a non-uniform model of computation, it cannot simulate computations that do not halt, and is therefore not capable of Turing universal computation.

In addition to further characterizing the computational power of standard molecular programming systems, our result has a number of important consequences. For instance, we show that even determining if a certain state is reachable in a CRN, such as a desirable or undesirable configuration, is PSPACE-hard, effectively demonstrating the intrinsic complexity of model checking and formal verification of chemical reaction networks. We further show the problem is PSPACE-complete for restricted classes of CRNs, such as when the CRNs are proper or when the reaction volume is a closed system (*i.e.,* tagged CRNs). The results also hold at the DSD level. In this chapter we once again reason concretely at the sequence level to consider folding pathways. We show that beginning with our Q3SAT solver construction we can prove that finding minimum energy barrier indirect folding pathways for multiple interacting strands is PSPACE-complete.

Chapter 6

Conclusion

Our research began with a desire to better understand the combinatorial nature of nucleic acid folding pathways between two secondary structures of the same nucleic acid strand. As folding pathways tend to avoid high-energy structures, a primary motivation was to understand and computationally predict low energy barrier folding pathways exhibited in biological systems. As with early studies of RNA structure prediction, we decided to focus on the *simple energy model* that corresponds to the number of base pairs of the involved structures. The reasoning for this choice was two-fold. First, this model is significantly simpler and remains sufficient to understand the complexity of the underlying combinatorial problem. If the problem is hard in the simple energy model, it provides evidence that it is hard for more complex models. Second, if effective algorithms are developed in the simple energy model, then it is possible they could be adapted for more complex energy models. This was the case for the RNA structure prediction problem that was first studied with the simple energy model [89] and later improved to use the Turner energy model [79].

6.1 Predicting folding pathways

At the outset of this research, the computational complexity for this problem remained unknown. We began by studying *direct* folding pathways [83] where intermediate structures could only remove base pairs from the initial structure and only add base pairs from the final structure. In Chapter 2 we have shown that the energy barrier problem for direct pseudoknot-free folding pathways is NP-complete, via a reduction from the 3-PARTITION problem. Thus, unless NP = P, there is no polynomial-time algorithm for calculating the energy barrier of direct folding pathways.

The proof in Chapter 2 can help shed insight on energy landscapes. A property of the proof is that there are exponentially many *partial* folding pathways that are within the minimum energy barrier, however, by design, only one will lead to a full pathway with minimum energy barrier. Thus, if pathways are followed according to a random process, it could take exponential time for the random process to find the pathway with minimum energy barrier. In one view, this suggests that for certain instances, it would be much more informative to ask which is the most *likely* folding pathway. This would be appropriate when the relative barrier difference between many possibilities is small. In another view, this suggests folding pathways may be leveraged to perform non-trivial

computation, especially if a guarantee can be made that the correct pathway has a significantly lower overall barrier than incorrect pathways. Unfortunately, our proof is deficient in forcing this separation between the correct pathway and other incorrect pathways. Specifically, the difference between a minimum energy barrier in the contrived construction of our proof, and an alternate *incorrect* folding pathway, may be some small constant. Therefore, while it remains hard to find a minimum energy barrier pathway, in the worst case, it may not be hard to find a close approximation. Our current result does not preclude this possibility.

Shortly before this initial research had begun, interesting new directions were being explored in the field of DNA computing and molecular programming. In particular, DNA strand displacement systems (DSD) were designed and implemented to perform simple computations, among other tasks. These systems use multiple interacting strands and are designed such that a correct sequence of strand displacements follows a low energy barrier folding pathway, while incorrect displacement sequences must overcome a larger energy barrier. Furthermore, the underlying designed folding pathways for many of these initial systems were *direct*. Specifically, these early systems shared the common characteristic that any particular strand may be displaced once and may displace one other strand. Thus, there was a growing need to understand and predict folding pathways of multiple interacting strands, even for direct folding pathways. Such knowledge could be used in the design and debugging of new molecular programs that leverage folding pathways. From our result on the single strand case in Chapter 2, by restriction we were able to conclude that predicting direct folding pathways for multiple interacting strands is also an NP-complete problem.

However, these initial complexity results did not resolve the complexity of the general energy barrier problem, in which the pathway need not be direct. Two challenges in understanding the complexity of this problem which needed to be considered were repeat base pairs—base pairs added and removed multiple times in a pathway—and temporary base pairs—base pairs not specified in the initial nor final structure but form temporarily in order to improve the energy barrier.

Regardless of problem complexity, there was a need for an exact prediction algorithm that is efficient in practice. Prior to this research, all exact algorithms had time and space complexity that were exponential in the size of the input (length of nucleic acid strand(s)). In Chapter 3, we proposed an algorithm to exactly solve a generalized version of the direct energy barrier folding pathway problem that is defined in terms of bipartite graphs. The algorithm has an exponential worst case time complexity, but, importantly, it uses only polynomial space. The algorithm is practical for most instances tested in our empirical study, although it fails to solve some instances in a reasonable runtime. Moreover, the algorithm is inherently parallel, and this parallelism could be exploited to help solve hard instances. One important contribution of this work is a polynomial time algorithm that can split a problem instance into many smaller sub-instances. While we cannot avoid exponential worst case runtime

with our method, our splitting algorithm may be of independent interest and could be used in conjunction with heuristic methods. For instance, it could be used to first partition the solution space into sub-problems, with the aim of improving both the efficiency and accuracy of the overall heuristic method used to solve each sub-problem.

Our pathway prediction algorithm only considered single strands. For direct pathways, it seems that the algorithm could be generalized in a straightforward manner to consider multiple interacting strands. Such a generalization should also consider the entropic penalty for strand association. It seems plausible that additional nodes added to the corresponding bipartite graph instance could achieve this aim. Such an algorithm would be useful for verification of DNA strand displacement systems which follow direct folding pathways.

Unfortunately, the algorithm does not seem immediately applicable for indirect folding pathways. The design of the algorithm explicitly assumes that the graph modeling the conflicts, between the arcs (representing base pairs) of the initial and final structures of a problem instance, is bipartite. In an indirect folding pathway, where any non-crossing arc forming a Watson-Crick base-pair can be added at any point along a pathway, the conflict graph is not necessarily bipartite (and unlikely to be in general). Still, it is possible that a better understanding of the structure of conflict graphs for indirect pathways could lead to a similar result. The conflict graphs formed for indirect folding pathways can be characterized as circle graphs. The conflict graphs for direct pathways are 2-colourable circle graphs (see Figure 3.8 for an example). For direct pathways, we were able to exploit the following property: if one could identify an MFE structure C consisting of arcs from both the initial and final structures, A and B respectively, then there always exists an optimal pathway from A to B via C. Could a generalized version of the algorithm proposed here be adapted for indirect pathways? Most properties exploited in the proofs are argued in terms of independent sets. Removing assumptions regarding the colourability of the graph would be a necessary first step.

Interestingly, by proving that the algorithm of Chapter 3 is correct, we were also able to prove that repeat base pairs do not help in a direct folding pathway. This established that the direct-with-repeats energy barrier folding pathway problem is NP-complete for the single strand case and NP-hard for the multiple interacting strand case. However, these early results, even those that consider repeat base pairs, did nothing to resolve the complexity of predicting indirect folding pathways that permit temporary base pairs. As these prediction problems were computationally hard, what they did do was to motivate us to study the computational limits of DNA strand displacement systems (DSD) that leverage low energy barrier folding pathways. How appropriate then that it was a DSD construction we devise in Chapter 5 that served as the basis to show that predicting minimum energy barrier indirect folding pathways for multiple interacting strands is PSPACE-complete.

However, our new construction shares a common deficiency with our original construction for the direct folding pathway prediction problem. Specifically, the minimum energy barrier pathway is only guaranteed to be a small constant

improvement over incorrect pathways. While the proof is sufficient to show the hardness of the problem in the worst case, it leaves open the possibility that there may exist a polynomial time constant factor approximation algorithm. It also underscores a significant issue, already well known to the community, that must be overcome in the design of DSDs: blunt-end displacements. In our folding pathway construction which is based on a sequence design for a particular DSD construction, we identified three types of illegal displacements. By using a more sophisticated sequence design, for two of these types of illegal displacements, we could ensure that the difference in energy between the minimum barrier pathway and any other pathway grows polynomially in the combined length of the strands. However, as a blunt-end displacement occurs with the use of an identical domain, a clever sequence design cannot improve the desired energy barrier separation. In these cases, the energy barrier separation is dictated fully by the length of toehold domains. By design, toeholds are always of constant length to ensure displaced strands can easily disassociate from template strands. While this is a significant issue for the design of DSDs, this does not preclude the possibility that another construction, not based on DSDs, could be found that gives a polynomial, or even logarithmic, separation between the minimum barrier pathway and all other pathways. Not only would this be informative for the prediction problem, it would also be an interesting future direction in the design of folding pathways for computation and other molecular programming tasks.

Unfortunately, the complexity of predicting a minimum barrier indirect folding pathway of a single strand remains open. This was the original problem that motivated this entire line of research and is the most relevant problem for understanding folding pathways within a biological context. While other variants of the problem proved to be computationally hard, it remains possible that this problem is in P. It could be the case that the direct folding pathway problem is too constrained to be easy, while the increased complexity in the indirect case only arises when there are a polynomial number of strands. If a polynomial time algorithm for this problem emerges, a logical next step would be to extend the result to use a more sophisticated energy model [79].

6.2 Designing folding pathways

The complexity of predicting minimum energy barriers suggested to us that folding pathways may be a mechanism for performing non-trivial computation. From that perspective, we next aimed to understand the computational power of deterministic molecular programs that leverage folding pathways. In particular, we were interested in understanding the computational limits of DNA strand displacement systems (DSD) and more generally, the chemical reaction networks (CRN) that they implement.

There was a particular need to understand space complexity in these models. Towards that end, in Chapter 4 we introduced the concept of a *tagged* chemical reaction network in order to account for changes to auxiliary strands and

complexes, often called *fuel* or *transformers* in our work, when reactions are implemented in a DSD. Specifically, each reaction i is assigned a unique tag, $\mathcal{T}_i$. When the reaction occurs in the forward direction, $\mathcal{T}_i$ is consumed and, if the reaction is reversible, a new tag $\mathcal{T}_i^R$ is produced. Should the reaction need to occur again in the forward direction, then either another copy of the tag must be available or the reaction must first be reversed in order to consume the tag $\mathcal{T}_i^R$ and re-produce the tag $\mathcal{T}_i$. This simple mechanism allows us to reason concretely about the required quantity of molecules necessary to complete a computation. In the context of molecular programs operating in a closed reaction volume, space can be thought of as the necessary size of that volume to fit all molecules necessary to complete a computation, inclusive of tags which represent fuel/transformers.

Can a *biological soup* of nucleic acids having total size $poly(n)$ perform a computation, by means of a folding pathway, of $\Theta(2^n)$ steps? In Chapter 4 we demonstrated that yes, this is possible, by first giving a tagged CRN for an n-bit Gray code counter that deterministically advances through 2^n states while only requiring $\Theta(n)$ total molecules and then showing how it can be implemented as a DSD using $poly(n)$ space. This implementation introduced the concept of recycling, or molecule reuse, in strand displacement systems and chemical reaction networks. To our knowledge, in addition to being the first molecular program to significantly recycle molecules/strands, this is the first example of a designed indirect folding pathway that has length exponential in the number of nucleotides of the interacting strands.

In developing our result, we also introduced the use of a *mutex* strand to an existing construction of Qian et al. [98] to ensure that any chemical reaction can be realized by a DNA strand displacement cascade that appears to execute atomically. In the least, this contribution provides a direct correspondence from a tagged CRN to a DSD implementation and as a result, greatly simplified our correctness proofs. Furthermore, it motivated us to continue reasoning at the more abstract level of CRNs.

While the Gray counter demonstrated that space-efficient molecular programming is possible, we next asked if *all* space-bounded Turing machine computations could be realized by strand displacement systems whose space and expected time are within a (small) polynomial factor of the space and time of the Turing machine computation. In Chapter 5, we gave a logically reversible space efficient CRN implementation of a quantified Boolean formula solver capable of solving any problem in PSPACE—the class of all problems solvable in polynomial space. Furthermore, our CRN can be realized by a space efficient DSD. We have also shown how these results can be extended to solve any problem in SPACE. Thus, any computation that halts can be solved by a space efficient DSD.

The only other DSD implementation capable of solving all problems in SPACE is the stack machine implementation of Qian et al. [98]. The model of their result is a variant of the DSD model in which reactions not only produce and consume signal strands, but can also extend or reduce the number of base units at one end of a polymer. The result of Chapter 5 improves upon the

stack machine implementation in terms of space efficiency, as the stack machine uses space proportional to computation length. However, our result falls short in a number of other respects when compared with the stack machine. Our construction provides a non-uniform model of computation, and thus, as currently described, a new CRN and thus DSD, must be created for each different problem instance that must be solved. It is conceivable that the result can be generalized to solve any problem instance, up to a particular size. The stack machine implementation is Turing universal. Since our result is based on a non-uniform model of computation and uses a reduction from a Turing machine to a quantified Boolean formula, it cannot simulate computations that do not halt, and is therefore not capable of Turing universal computation.

In addition to further characterizing the computational power of standard molecular programming systems, we considered a number of related problems in Chapter 5. We showed that determining if a certain state is reachable in a CRN, such as a desirable or undesirable configuration, is PSPACE-hard. This demonstrates the intrinsic complexity of model checking and formal verification of chemical reaction networks. We further showed that the problem is PSPACE-complete for restricted classes of CRNs, such as when the CRNs are proper—each reaction produces the same number of molecules it consumes—or when the reaction volume is a closed system (*i.e.,* the CRN is *tagged*). The results also hold at the DSD level.

Aside from the potential biological and chemical applications, DSDs and CRNs are also of independent interest due to their promise for realizing energy efficient computation. Rolf Landauer proved that logically irreversible computation — computation as modeled by a standard Turing machine — dissipates an amount of energy proportional to the number of bits of information lost, such as previous state information, and therefore cannot be energy efficient [69]. Surprisingly, Charles Bennett showed that, in principle, energy efficient computation is possible, by proposing a logically reversible universal Turing machine and identified nucleic acids (RNA/DNA) as a potential medium for reversible computation [8]. However, this remained a theoretical result with no known physical implementation. It was Qian et al. [98] who first demonstrated that energy efficient computation could be realized, in principle, with a logically reversible DSD system that simulates a stack machine. Our quantified Boolean formula solver of Chapter 5 is also logically reversible. Thus, we have demonstrated that any space-bounded Turing machine computation can be realized, in principle, by a space *and* energy efficient CRN and DSD.

However, our CRN implementations throughout this thesis share a common assumption with the stack machine implementation: certain initial signal molecules must occur as a single copy. Initially, this assumption served to simplify the discussion. However, in Chapter 4 we have shown that this assumption is actually crucial to achieve space-efficient computation. We have shown that for any proper CRN, any signal molecule can be produced using just $O(n^2)$ reaction steps when $\Theta(n)$ copies of the initial signal molecules share the same volume. This result has since been improved by others to consider more general classes of CRNs [27]. We have also shown a much stronger result for determinis-

tic computations in a closed volume. Specifically, even having a second copy of the initial input signals ensures that no tagged CRN can be deterministic after a linear number of steps. The intuition as to why the single copy assumption is important is that it gives us a means to *erase* information. In a single copy setting, once a molecule of a particular type is consumed, it is no longer present. In a multi-copy setting, once a molecule of a particular type is consumed, there is no guarantee that the other copies are simultaneously consumed.

While the single copy restriction permitted us to study the very limits of computation for a *biological soup*, it imposes a significant engineering challenge. All DSD implementations to-date use concentrations of strands of each type. Producing and successfully executing a DSD with a single copy restriction is currently challenging, but feasible. For instance, the first published result on the measurement of a single enzyme molecule was by Boris Rotman, in 1961 [107]. The experimental techniques developed in that first paper are still influential and in use, and new advancements in single molecule studies continue to be made [62].

Our results also hold at the more general level of chemical reaction networks. Thus, any physical realization of a CRN could, in principle, make use of our constructions. Furthermore, the problems in a multi-copy system only arise when signals from one copy interfere, or *cross-talk*, with signals from another copy. If one copy could be compartmentalized from another, the challenge could be overcome. This may involve a move away from a *biological soup*, and back to a strictly surface based model [14], or some hybrid, possibly involving recent advances in DNA origami [105]. If all signals from each copy were tethered to a surface, and separated from other copies, then reactions could proceed as expected. This is not our idea. It has been suggested as a means to improve the speed of DSD reactions by co-locating related strands [19, 100].

Should a practical means be developed to address the single-copy issue, then a more rigorous study of logically reversible CRNs is appropriate. In the course of our research it became clear that certain techniques could be used when developing a logically reversible CRN. For instance, the 3SAT verification procedure could be reliably executed, much like a subroutine, by producing signal molecules necessary for either its initial or its final reaction, and by ensuring that signals produced and consumed by intermediate reactions involved only signals *local* to that procedure. Furthermore, a common technique we used was to add an additional reaction to effectively double the length of a computation by forcing the original chain of reactions to reverse. This permitted us to *actively* recycle molecules/strands and was evident in both our Gray code counter of Chapter 4 and also in our quantified Boolean formula solver of Chapter 5. These techniques could conceivably be extended into a formal grammar for programming logically reversible CRNs.

Finally, we find the current complexity classes for logically reversible computation too general to capture the realities of logically reversible molecular programming. The class ReversibleSPACE represents all problems that can be solved by a space-bounded logically reversible Turing machine. As with any Turing machine, the space bound is with respect to the length of tape necessary

to complete the computation. In CRNs and DSDs, bits of information are represented with the presence and absence of signal molecules. Thus, the length of tape required in the Turing machine computation corresponds well with the maximum quantity of signals required during the CRN computation. However, this does not account for *fuel* (transformers) that a CRN may require to complete its computation. The reaction is the fundamental operation in a CRN just as a state transition is the fundamental operation for a Turing machine. However, with current technology, a reaction in a CRN requires *fuel*, which in turn requires physical space, whereas a Turing machine state transition does not. In essence, a logically reversible Turing machine could perform all state transitions in only one direction, while still using significantly less space than the number of computation steps. This is not currently possible in molecular programming.

We have demonstrated that any space-bounded computation can be realized with a logically reversible *tagged* CRN that requires only one tag per reaction equation. In essence, our logically reversible CRN strictly alternates its reactions in the forward and reverse direction. It is conceivable that we could simulate our CRN with a logically reversible Turing machine. It is also conceivable that our simulation could be constructed to ensure that each state transition of the Turing machine either strictly alternates being applied in the forward and reverse direction, or adheres to a polynomial bound in the difference between forward and reverse transitions, at every step of the computation. Should such a construction be possible, we will have given a logically reversible Turing machine, capable of simulating any space-bounded Turing computation, that is semantically restricted to capture the notion of *fuel*. We let $\mathsf{ReversibleSPACE}^{\bigstar}$ denote the class of problems solvable by such a Turing machine. It has already been shown by Lange et al. [70] that $\mathsf{ReversibleSPACE} = \mathsf{SPACE}$. In future work, it is our goal to show that $\mathsf{ReversibleSPACE}^{\bigstar} = \mathsf{SPACE}$.

Part II

Space efficient text indexes motivated by biological sequence alignment problems

Chapter 7

Introduction

7.1 Text indexing

The study of strings, their properties, and associated algorithms has played a key role in advancing our understanding of problems in areas such as compression, text mining, information retrieval, and pattern matching, amongst numerous others. A most basic and widely studied question in stringology asks: given a string T (the text) how many occurrences and in what positions does it contain a string P (the pattern) as a substring? It is well known that this problem can be solved in time proportional to the lengths of both strings [63]. However, it is often the case that we wish to repeat this question for many different pattern strings and a fixed text T of length n over an alphabet of size σ. The idea is to create a full-text index for T so that repeated queries can be answered in time proportional to the length of P alone. It was first shown by Weiner [139] in 1973 that the suffix tree data structure could be built in linear time for exactly this purpose. The ensuing years have seen the versatility of the suffix tree as it has been demonstrated to solve numerous other related problems.

While suffix trees use $O(n)$ words of space in theory, this does not translate to a space efficient data structure in practice. For this reason, Manber and Myers [78] proposed the suffix array data structure. Though a great practical improvement over suffix trees, the $\Omega(n \log n)$[26] bit space requirement is often prohibitive for larger texts. Building in part on the pioneering work of Jacobson [56] on succinct data structures, two seminal papers helped usher in the study of so-called succinct full-text indexes. Grossi and Vitter [45] proposed a compressed suffix array that occupies $O(n \log \sigma)$ bits; the same space required to represent the original string T. Soon after, Ferragina and Manzini [33] proposed the FM-index, a type of compressed suffix array that can be inferred from the Burrows-Wheeler transform (BWT) of the text and some auxiliary structures, leading to a space occupancy proportional to $nH_k(T)$ bits, where $H_k(T)$ denotes the k^{th} order empirical entropy[27] of T. BWT is a reversible transformation that produces a permutation of the original text (T^{bwt}) which is more easily compressible[28] with the use of local compressors such as run length

[26] We use log to denote $\log_2$ throughout.

[27] For a text T of length n, the term $(H_k(T))n$ is a lower bound on the number of bits required to encode T with any algorithm that uses contexts of length at most k.

[28] The BWT works by (1) creating a conceptual matrix where each row is a different cyclic rotation of the original text (with an appended special character, lexicographically smaller than any character from the alphabet); (2) sorting the rows in lexicographical order and (3)

encoding (often preceded by the Move-to-front transform [10]). The Burrows-Wheeler transformed text T^{bwt} is related to the original text by the so-called LF mapping. Thus, as Ferragina and Manzini showed, any queries on the compressed representation of T^{bwt} can be interpreted as queries on T itself (and conversely). This is the basic idea behind most succinct self-indexes proposed thus far [88]. We note that the success of succinct data structures in general relies on efficient operations on bit vectors, and is due to the seminal work of Jacobson [56] and Munro *et al.* [86]. For an in-depth discussion, the reader is referred to the work of Mäkinen and Navarro [76]. For details on the LF mapping, BWT or compressed full text indexes in general, the reader is referred to the excellent review by Navarro and Mäkinen [88]. These and subsequent results have made it possible to answer efficiently the substring question on texts as large, or larger, than the Human genome.

7.2 Biological sequence alignment

The Human Genome Project has enabled a revolutionary step forward in understanding our genes and their function. A significant next challenge is to understand genome variation across individuals and its correlation with disease, as well as genomic mutations and rearrangements in cancerous cells. Since at least two reference human genome sequences are now available, *de novo* assembly of a genome of interest from short fragments — inferring a linear genome sequence from a collection of shorter DNA fragments called *reads* — is no longer required in most human genomic studies. Instead, current studies focus on resequencing, that is, inference of the genome of interest by alignment of the reads, produced by sequencing the genome, to the available reference genomes (see Figure 7.1). The actual information sought is not the canonical sequence of the genome of interest, but rather, how does it differ from a known reference?

For example, single nucleotide variations (SNVs) in an individual's genome (compared with the *wild type* or reference genome) have been identified as significant in many types of human cancer [13, 40, 48, 93, 103, 144] (see Figure 7.1). These discoveries are enabling the development of novel methods for disease diagnosis and therapy [22].

Fueling the discovery of genetic variation amongst populations and individuals has been the application of next generation sequencing technology (NGS). While the technologies underlying competing NGS platforms vary, all share significant differences from traditional, Sanger style sequencing [57]. The new technologies focus on massively parallel sequencing and are capable of producing millions of reads in a typical run [53, 57]. While the sheer quantity of reads and overall bases which can be sequenced in a given time frame are vastly greater

outputting the last column of the sorted matrix. The BWT is useful for compression as it often produces successive runs of the same character. For instance, consider a text over the English language containing many instances of the words *this, that, there, the, those, etc.* Cyclic rotations beginning with the letter *h* would have a high probability of ending with the letter *t*. Thus, after sorting, the transformed text, taken from the last column, would likely contain runs of the letter *t*.

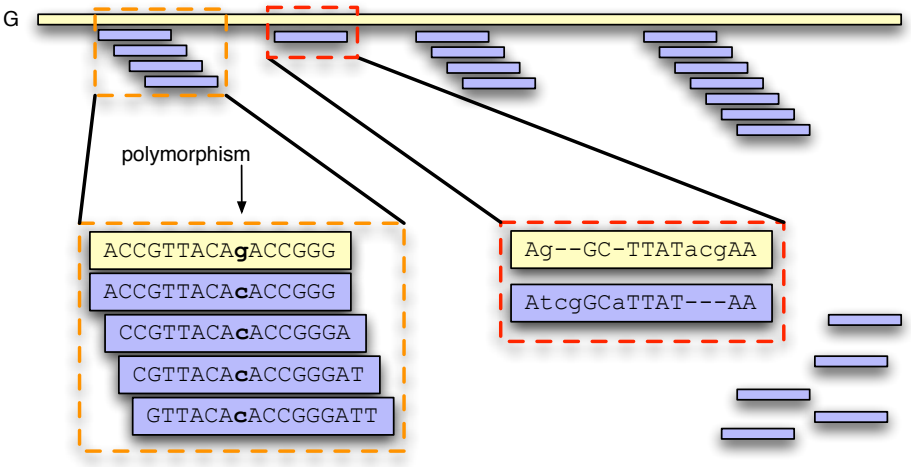


Figure 7.1: An example of short reads aligned to a reference genome G. Alignments may contain matches, mismatches, insertions and deletions. For instance, the alignment of the single read to the reference (red outline) contains a match in the first position of the alignment, a mismatch in the second, an insertion in the third position and a deletion in the twelfth position. Sequencing the genomes of individuals helps determine genetic mutations, such as single nucleotide polymorphisms/variations (SNPs/SNVs) of individuals compared to a reference genome.

than traditional Sanger style sequencing [118], there are at least two caveats. First, NGS reads are short, typically between 28 and 300 bases, dependent on the specific platform (compared to $\sim 1,000$ base reads of Sanger style sequencing) [118]. Second, NGS reads are more prone to sequencing errors, whereby the reported sequence of a read differs from the true sequence of the DNA molecule; these differences can be characterized by the common string edit operations of substitution, insertion and deletion. The rate and type of sequencing errors is also dependent on the platform employed. Both of these features of NGS reads, coupled with the scale of the produced data, can confound the task of efficiently aligning reads to a reference genome and have forged it as one of the most actively researched problems in contemporary bioinformatics.

NGS is also being utilized to capture data from the transcriptome; a process referred to as RNA-Seq [84]. Instead of sequencing genomic DNA, RNA-Seq aims to sequence the complementary DNA (cDNA) of RNA molecules in a cell. Transcriptome read alignment is providing valuable information to researchers, beyond genomic sequencing. In particular, this technology can be used to quantify the level of expression of various transcripts by sequencing messenger RNA, thus implicating the relative expression level of proteins. For instance, a highly expressed transcript should yield higher read coverage than a poorly expressed transcript. This technology has also been used to elucidate RNA editing events, exon boundaries and novel alternative splice junctions [96, 132, 138].

7.3 Objectives

The use of full-text indexing has played a crucial part in advancing our understanding of biological sequence data. The abundance and production rate of high-throughput sequencing data and the size of genome sequences all but made necessary the use of succinct text indexes—those using space proportional to the information theoretic lower bound. Indeed, the more recent aligners, including `Bowtie` [71] and `BWA` [73] amongst numerous others, make use of succinct self-indexes[29]. As memory resources on commodity hardware have steadily increased, succinct indexes are no longer necessary for many of these applications when large memory machines are available. However, even when sufficient memory is available to use non-succinct text indexes, use of succinct indexes often results in more efficient solutions due to better caching performance as one would expect in non-uniform memory architectures (NUMA).

Rather than add to the growing number of new tools that improve sequence alignment efficiency through the use of various heuristics and implementation techniques, my interest lies in adapting and improving the underlying full-text index data structures to better model biological reference sequences such as genomes. In particular, I have identified two specific types of biological sequence events, alluded to in the previous section, that are not naturally captured by a static, linear text. The first is the presence of known variations, amongst a

[29] A text index is a *self-index* if it does not require the original text to be stored but can provide efficient access to any substring of the original text.

population, in specific positions of a reference genome. The second is known splicing events between positions in a reference genome that form part of a common transcript sequence. More details of these events and how they are incorporated into a full-text index are given in subsequent chapters. The solutions proposed are theoretically rigorous in the sense that all time and space complexity claims are formally proved. My focus in proposing these indexes was to improve various time and space trade-offs compared with existing solutions found in the literature. While these contributions are made and studied from a theoretical perspective, the hope is that incorporating this information directly into the primary indexes will result in more biologically meaningful alignments and/or improve the overall efficiency of the alignment problem. As discussed in the conclusion of this part, additional steps are necessary for these results to have a practical impact, such as adapting query algorithms for approximate matching. While my motivation arises from these particular applications in biological sequence alignment, the results I give in this part are more general and may have applications in other problem domains.

7.4 Contributions

We now describe the contributions of this part of the thesis, in the order they are discussed:

1. We propose a compressed full-text dictionary that can index a set of text segments in space proportional to the compressed size of their concatenation while still supporting a number of efficient query operations. These include determining which text segments are contained within a query pattern and which contain or prefix a query pattern.

2. We propose new succinct indexes for text containing wildcards that improve the space complexity compared with the existing state-of-the-art at the time they were proposed without increasing query time complexity. Independently and in parallel with another group, we give the first compressed index for this problem. We also show how our results can be combined with the results proposed independently to improve the state-of-the-art for this problem.

3. We propose a new query algorithm, based on dynamic programming, for indexes of text with wildcards that significantly improves the query working space complexity over existing solutions. The algorithm is fairly general and easily adapted for use with other indexes that use the same general strategy of pattern matching in text with wildcards.

4. We show a correspondence between the wildcard and hypertext indexing problems by demonstrating that standard strategies and techniques for solving the former can be generalized to solve the latter.

5. We propose the first index for hypertext, a graphical generalization of text. We first propose a succinct index and later show how the index only requires space proportional to the length of the compressed text and topology of its graphical structure. We also study a number of interesting restrictions for hypertext.

7.5 Outline

Each chapter of this part builds on results from previous chapters. Chapter 8 introduces much of the notation and existing results from the literature that are leveraged throughout the part. Subsequent chapters develop notation and introduce other existing results as needed, in a cumulative manner. The first technical result presented in this part is the design of the full-text dictionary index and is presented in Chapter 8. This result is leveraged in the design of our other proposed text indexes. While it is unnecessary to understand the complete implementation details of the full-text index in order to understand the wildcard or hypertext index, it is critical to understand its supported query operations as stated in Theorem 22. In Chapter 9 we develop indexes for text containing wildcards and the associated query algorithms for exact matching. Chapter 10 details indexes for hypertext, a graphical generalization of linear text. In Chapter 11 we summarize our contributions of this part and highlight a few of the more important open problems arising from this work.

Chapter 8

A compressed full-text dictionary

8.1 Introduction

A *full-text index*[30] is a data structure that can efficiently determine the positions, in a fixed string T (the text), where an arbitrary query string P (the pattern) appears as a substring. Such an index is useful, for instance, to search an electronic book repeatedly for different words, or a genome sequence for biological sequence signatures. A *dictionary* is a data structure for a fixed and ordered collection $D = (T_1, T_2, \ldots, T_d)$ of strings called *text segments*, that can efficiently determine all occurrences of all text segments that appear as a substring in an arbitrary query string P; these are called *dictionary matches*. Such an index is useful, for instance, to search many email messages for all occurrences of a fixed collection of keywords associated with spam[31]. The entirety of this chapter is devoted to describing a new data structure called a compressed *full-text dictionary* that combines the features of a full-text index with those of a dictionary, and uses space roughly equal to that of just a compressed full-text index. Specifically, we are interested in the benefits of a full-text index for a string T and also the ability to perform dictionary matches when T is composed of a number of text segments delimited by a special character. Such a data structure can be used to improve the space complexity of current approaches for indexing text containing wildcards (see Chapter 9). Note that it is always possible to create a compressed full-text dictionary of any ordered collection of text segments, by first concatenating them using a special character as a delimiter. In Chapter 10, we show this approach is useful for creating a compressed index for hypertext, a generalization of linear text.

Content from this chapter appears in the proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011) [127] and the journal Theoretical Computer Science [128].

[30] The term *full-text index* is used in the string community to denote a data structure that supports efficient substring query operations over a text. Constructing such a data structure is referred to as *indexing* the text and the data structure itself is referred to simply as an *index* for the text.

[31] The term *spam* is used to describe unsolicited and indiscriminate messages sent in bulk as electronic mail.

8.1.1 Related work

A succinct dictionary was used as a subsidiary data structure in the approach of Tam et al. [124] who proposed the first succinct index for text containing wildcards. Very recently, Belazzougui [6] proposed a compressed dictionary based on the Aho-Corasick automaton having optimal query time. The compressed space occupancy was further improved by a modification given by Hon et al. [50]. While these results are impressive and interesting in their own right, the wildcard matching problem and hypertext matching problem, discussed in subsequent chapters, benefit from a full-text dictionary that can report the text segments contained in P (dictionary matches), as well as the text segments which are prefixed by P and also fully contain P (substring matches).

8.2 Preliminaries

We first develop notation that will be used throughout this chapter and the subsequent chapters of this part. We also state useful lemmas for fundamental and well-known succinct data structures that our results employ as subsidiary data structures. In addition to the space complexity of these data structures, we give the time complexity for the relevant operations we perform on them. It is not necessary to understand the details of how these subsidiary data structures support the listed operations in order to understand our results. However, we point the reader to relevant literature should the details of such data structures be of interest. Unless otherwise stated, equivalent or improved versions of these subsidiary data structures can be substituted in the development of our data structures.

Let $T[1, n]$ be a string over a finite alphabet Σ of size σ. We denote its j^{th} character by $T[j]$ and a substring from the i^{th} to the j^{th} position by $T[i..j]$. We assume that an end-of-text sentinel character $\$ \notin \Sigma$ has been appended to T ($T[n] = \$$) and $\$$ is lexicographically smaller than any character in Σ. For any substring X we use $|X|$ to denote its length and $\overline{X}$ to denote its reverse sequence. The suffix array SA of T is a permutation of the integers $[1, n]$ giving the increasing lexicographical order of the suffixes of T where $\mathsf{SA}[i] = j$ means that the i^{th} lexicographically smallest suffix of T begins at position j. Conceptually, SA can be thought of as a list of all suffixes of T in lexicographic order. For example, Figure 8.1(c) gives the sorted list of the twelve suffixes of the string `mississippi$`.

A string X has a suffix array (SA) range $[a, b]$ with respect to SA if $a - 1$ suffixes of T are lexicographically smaller than X and $b - a + 1$ suffixes of T contain X as a prefix. If $a > b$ the range is said to be an empty SA range and X does not exist as a substring of T. Consider the sorted suffixes for the string `mississippi$` shown in Figure 8.1(c). The query string `iss` has the SA range $[4, 5]$ as there are three lexicographically smaller suffixes, and exactly two suffixes are prefixed by `iss`.

In our full-text dictionary, we will not construct the suffix array for T.

Rather, we will use a compressed suffix array CSA of T. A compressed suffix array is a space efficient representation of both the string T and also the suffix array for T. Many compressed suffix array implementations store a representation of T^{BWT}, the Burrows-Wheeler transform of T. Determining T^{BWT} for a string T can be thought of in this way: (i) create a conceptual matrix of all cyclic rotations of T, (ii) sort all rows of the matrix into lexicographic order, and (iii) output the last column of the conceptual matrix. This process is illustrated in Figure 8.1.

(a) List cyclic rotations (b) Sort cyclic rotations (c) Output last column

Figure 8.1: The Burrows-Wheeler transform of a string $T =$ `mississippi$` is $T^{\text{BWT}} =$ `ipssm$pissii`.

Compressed suffix array implementations that rely on the Burrows-Wheeler transform make use of the so-called LF-mapping, that relates characters in the *last* column of the conceptual transform matrix, to characters in the *first* column. Specifically, the i^{th} occurrence of a character $c \in \Sigma$ in the last column corresponds to the i^{th} smallest suffix that begins with character c. For example, Figure 8.2(b) shows how the third and fourth occurrence of the character `'i'` in T^{BWT} corresponds to the third and fourth smallest suffix that begins with character `'i'`.

These implementations search for a match of a pattern $P[1 \ldots m]$ by first finding the suffix array range $[sp, ep]$ for the string $P[m-1 \ldots m]$. If $[sp, ep]$ is not an empty range, then a new range is determined for the string $P[m-2 \ldots m]$, and so on. In this way, patterns are searched backwards and the search algorithm is appropriately called *backward search* [34]. An example of extending a match from the pattern `'s'` to the pattern `'is'` is shown in Figure 8.2. The idea is to find, in the current SA range of the string T^{BWT}, the first and the last occurrence

of the character that will extend the pattern (in this case `'i'`). The LF-mapping can then be used to update the current SA range to point to all suffixes prefixed by the extended pattern (`'is'`). Knowledge of how this algorithm works is not necessary to understand our result; however, for details of the algorithm and related topics we refer the reader to the excellent review by Navarro and Mäkinen [88].

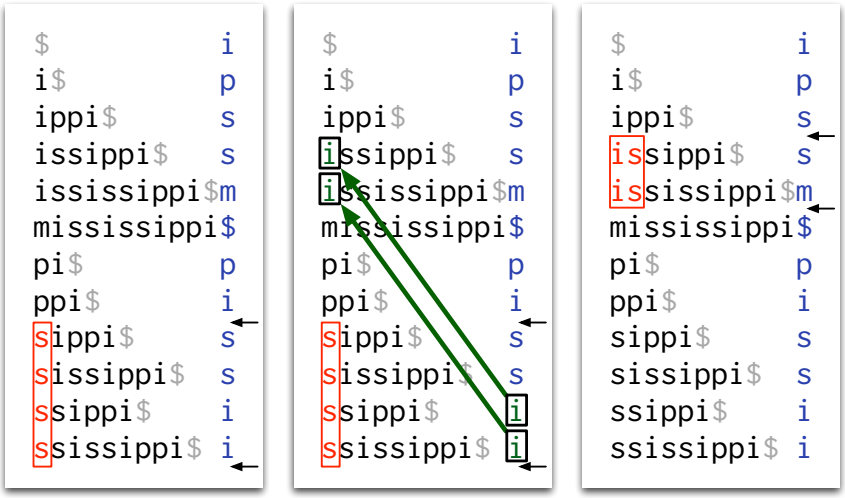


Figure 8.2: Performing backward search to find the SA range of the string `'is'` from the SA range of the string `'s'`, using $T^{\mathtt{BWT}}$, the Burrows-Wheeler transform of text T. (a) The current match and SA range for `'s'`. (b) All occurrences of character `i` in $T^{\mathtt{BWT}}$ within the current SA range are identified. (c) The LF-mapping is used to update the SA range to the new match `'is'`.

Our full-text dictionary can be made to utilize any compressed suffix array that supports the LF-mapping, and thus backward search. However, we restrict our attention to an implementation based on the wavelet tree representation [46] of $T^{\mathtt{BWT}}$ whose properties are exploited in Section 9.6 to further reduce the required space of the overall index we propose for text containing wildcards.

Lemma 24 (Mäkinen & Navarro [76])**.** A compressed suffix array CSA, based on the wavelet tree of $T^{\mathtt{BWT}}$, can be represented in $nH_k(T) + o(n \log \sigma)$ bits of space, for any $k \leq \alpha \log_\sigma n - 1$ and $0 < \alpha \leq 1$, such that the operation $\mathtt{rank}_c(T^{\mathtt{BWT}}, i)$ which counts the occurrences of character c up to position i in $T^{\mathtt{BWT}}$ and also the LF operation are supported in time $O(\log \sigma)$, the suffix array range of every suffix of a string X can be computed in $O(|X| \log \sigma)$ time, and each match of X in T can be reported in an additional $O(\log^{1+\epsilon} n)$ time, for any $\epsilon > 0$, where T is a text of length n over an alphabet of size σ.

In our full-text dictionary construction, we also make use of the following well known data structures.

Lemma 25 (Raman et al. [101])**.** A bit vector B of length n containing d 1 bits can be represented in $d \log \frac{n}{d} + O(d + n\frac{\log\log n}{\log n})$ bits to support the operations $\texttt{rank}_1(\mathsf{B}, i)$ giving the number of 1 bits appearing in $\mathsf{B}[1..i]$ and $\texttt{select}_1(\mathsf{B}, i)$ giving the position of the i^{th} 1 in B in $O(1)$ time.

Lemma 26 (Grossi & Vitter [45])**.** An array L of d integers where $\sum_{i=1}^{d} \mathsf{L}[i] = n$ can be represented in $d(\lceil \lg(n/d) \rceil + 2 + o(1))$ bits to support $O(1)$ time access to any element.

Lemma 27 (Munro & Raman [85])**.** A sequence BP of d balanced parentheses can be represented in $(2+o(1))d$ bits of space to support the following operations in $O(1)$ time: $\texttt{rank}_(\mathsf{BP}, i)$, $\texttt{select}_(\mathsf{BP}, i)$, and similarly for right parentheses, as well as:

- $\texttt{findclose}(\mathsf{BP}, l)$ ($\texttt{findopen}(\mathsf{BP}, r)$): index of matching right (left) parenthesis for left (right) parenthesis at position l (r)
- $\texttt{enclose}(\mathsf{BP}, i)$: indices (l, r) of closest matching pair to enclose $(i, \texttt{findclose}(\mathsf{BP}, i))$ if such a pair exists and is undefined otherwise

The matching statistics for a string X with respect to T is an array ms of tuples such that $ms[i] = (q_i, [a_i, b_i])$ states that the longest prefix of $X[i..|X|]$ that matches anywhere in T has length q_i and suffix array range $[a_i, b_i]$. Very recently Ohlebusch et al. [91] showed matching statistics can be efficiently computed with backward search if CSA is *enhanced* with auxiliary data structures using $O(n)$ bits to represent so-called longest common prefix (lcp) intervals (*cf.* [91]). We leverage this result in the design of our compressed full-text dictionary and its search algorithm.

Lemma 28 (Ohlebusch et al. [91])**.** The matching statistics of a pattern X with respect to text T over an alphabet of size σ can be computed in $O(|X| \log \sigma)$ time given a compressed enhanced suffix array of T.

8.3 Overview of the full-text dictionary

The data structure we propose in this chapter can be built based on an already existing string that contains text segments as substrings, or based on an ordered list of d text segments. Suppose we are given a string T that contains d text segments. Specifically, let $T = \phi^{k_1} T_1 \phi^{k_2} T_2 \phi^{k_3} T_3 \ldots \phi^{k_d} T_d \phi^{k_{d+1}}\$$ be a string over an alphabet $\Sigma \cup \{\phi\}$, followed by the traditional end-of-text sentinel \$, having total length n. We define ϕ to be lexicographically smaller than any $c \in \Sigma$ and \$ to be lexicographically smaller than ϕ. We call the character ϕ a *delimiter*, and let ϕ^{k_i} denote the i^{th} group, or run, of delimiter characters having length $k_i \geq 0$, for $1 \leq i \leq d+1$. The string T is defined to contain exactly d text segments—maximal substrings that do not contain a delimiter character. By definition,

text segments must be separated by a run of one or more delimiter characters. Therefore, ϕ^{k_i}, the delimiter group separating text segments T_{i-1} and T_i, must have length $k_i > 0$, for $1 < i \leq d$. In this case, the underlying ordered list of text segments is $D = (T_1, T_2, \ldots, T_d)$. Suppose we are given an ordered list $D = (T_1, T_2, \ldots, T_d)$ of d text segments. Then we can construct a string $T = T_1\phi T_2\phi \ldots T_d\$$, that is a serialization of all text segments in D delimited by the character ϕ. In either case, we will begin our construction with a string T of length n containing the d text segments of $D = (T_1, T_2, \ldots, T_d)$, delimited by at least one ϕ character.

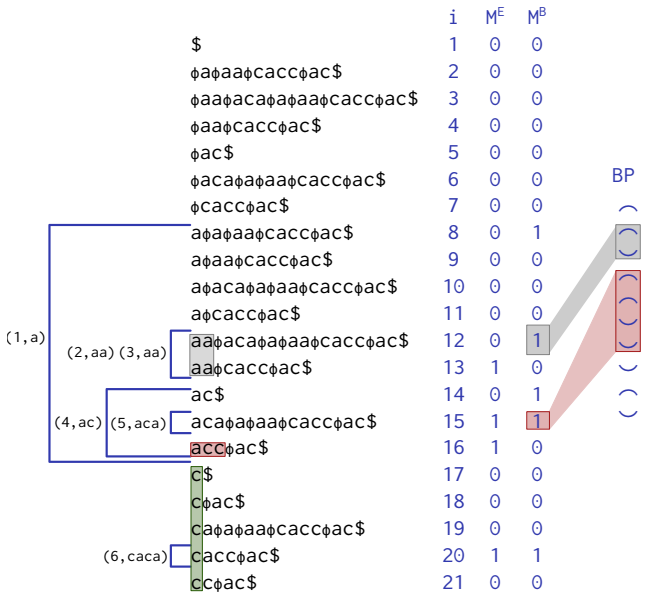


Figure 8.3: A compressed full-text dictionary for the ordered list of text segments (aa, aca, a, aa, cacc, ac). The first three columns give a conceptual representation of the full-text dictionary. The second column shows the sorted suffixes of the serialized string $T = \phi$aaϕacaϕaϕaaϕcaccϕac\$ representing the text segments. The third column contains the array i indicating the sorted lexicographic rank of each suffix of T. The first column shows the SA ranges of the text segments and their containment relationship. Each text segment SA range is labeled by (lex_id, segment) pairs. Shown in the last three columns are actual data structures used in the full-text dictionary representation: the M^{E} array which marks the end of one or more text segment SA ranges, the M^{B} array which marks the beginning of each text segment SA range, and the BP array that represents the containment of text segment SA ranges (their tree topology). Three different queries (shaded intervals) are shown with their corresponding smallest enclosing text segment SA range (if any) marked in the BP array.

8.3.1 The `lex_id` of text segments

Let $pos(T_i)$ denote the starting position in T of the i^{th} text segment. In our supported operations, we find it convenient to refer uniquely to each of the d text segments as a number in the range $[1, d]$, called a `lex_id`, which is based on the relative lexicographic order of the d suffixes of T that are prefixed by a text segment. Informally, if the suffix $T[pos(T_i)..n]$ is the j^{th} lexicographically smallest suffix, of all d suffixes of T that are prefixed by a text segment, then T_i will have `lex_id` j. We can formalize this notion as follows. Let $[a_i, b_i]$ be the SA range of $T[pos(T_i)..n]$, the suffix of T beginning with the i^{th} text segment. Since all suffixes of T are unique, then $a_i = b_i$, and $a_i \neq a_j$, unless $i = j$. The `lex_id` of text segment T_i is $j = |\{a_k \,|\, a_k \leq a_i, 1 \leq k \leq d\}|$.

The `lex_id` is defined in such a way that a range of matching `lex_id`s can be returned for the `prefix` operation that our full-text dictionary will support. We will see the benefits of this when the full-text dictionary is used within the other data structures we develop in Chapters 9 and 10. An example full-text dictionary is given in Figure 8.3 and shows the `lex_id`s and SA ranges for six different text segments. Note that in this example, there are two text segments that share a common string (`'aa'`) and therefore there are only five unique SA ranges.

Now that the form of the data that we are creating a data structure for is clear, and the concept of `lex_id`s of text segments has been formally defined, we can state the main result we show in this chapter. Throughout the chapter, we will assume[32] that a query pattern P is over the alphabet Σ.

Theorem 22. A string T over alphabet $\Sigma \cup \{\phi\}$ of length n, that contains d, not necessarily distinct, text segments over alphabet Σ, can be represented by a compressed full-text dictionary F in $|\mathsf{CSA}| + O(n) + O(d \log n)$ bits, to support the following operations given any query pattern P:

- `dict_prefix` (F, P): returns the (possibly empty) list of the occ_1 `lex_id`s for text segments that prefix P in $O(|P| \log \sigma + occ_1)$ time,
- `dict_match` (F, P): returns the (possibly empty) list of the occ_2 `lex_id`s for text segments that are contained as substrings in P in $O(|P| \log \sigma + occ_2)$ time,
- `dict_count` (F, P): returns the count of text segments that are contained as substrings in P in $O(|P| \log \sigma)$ time,
- `prefix(F,` P): returns the (possibly empty) range $[lexid1, lexid2]$ of `lex_id`s for text segments that are prefixed by P in $O(|P| \log \sigma)$ time,
- `locate(F,` P): returns the (possibly empty) list of the occ_3 positions in T that are prefixed by P in $O(|P| \log \sigma + occ_3 \log^{1+\epsilon} n)$ time, for any $\epsilon > 0$,

[32] This simplifies the discussion, though it is worth noting that any invalid character in a pattern P can be identified in $O(|P|)$ time.

- `match_stats` (F, P): returns the matching statistics $\{(q_i, [a_i, b_i]) \parallel 1 \leq i \leq |P|\}$ for P with respect to T in $O(|P| \log \sigma)$ time,

where $\sigma = |\Sigma \cup \{\phi\}|$ and $|\mathsf{CSA}|$ denotes the size of any compressed suffix array of T supporting the LF operation in time $O(\log \sigma)$.

8.4 Components of the full-text dictionary

Before discussing how to perform any operations on a full-text dictionary F, we first describe its subsidiary data structures. We focus first on describing these data structures, but leave details on how they are constructed to Section 8.6.

8.4.1 CSA: compressed *enhanced* suffix array

We first build CSA, the compressed suffix array for T, using $|\mathsf{CSA}|$ bits, and then *enhance* it to represent longest common prefix (lcp) intervals (*cf.* [91]) using an additional $O(n)$ bits. Analogous to the notation $\mathsf{SA}[i]$, we let $\mathsf{CSA}[i]$ denote the starting position of the i^{th} lexicographically smallest suffix of a text T.

8.4.2 The `sa_id` identifier and RSA: conceptual tools

In Section 8.3.1 we formally defined the `lex_id` as a unique identifier for every text segment in T. By definition, text segments are not necessarily distinct strings. Text segments that are the same string will have the same SA range in T and also the same length. To simplify the discussion of the components of our data structure, and also the operations it supports, we will introduce an internal identifier for text segments called the `sa_id`. Intuitively, the `sa_id` specifies a total ordering of all the unique SA ranges that are associated with text segments. We will give a formal definition, but first we prove the properties we need to group text segments. The following lemma formalizes two notions: (i) text segments representing different strings must have SA ranges that *begin* at different positions; and (ii) those representing the same string must have the same SA range and the same length.

Lemma 29. Let $[a, b]$ and $[c, d]$ be the non-empty suffix array ranges in CSA for a text segment T_i and a text segment T_j. Then $a = c$ if and only if $T_i = T_j$ and $b = d$.

Proof. Suppose $a = c$. Then T_i and T_j share a common prefix of length $\min(|T_i|, |T_j|)$. Since each text segment in T is followed by a character in $\{\phi, \$\}$ and no text segment contains a character in $\{\phi, \$\}$, then it must be the case that $|T_i| = |T_j|$. Therefore $T_i = T_j$, and $c = d$ since identical strings must have the same SA range in T. Suppose $T_i = T_j$ (and $b = d$). Identical strings must have the same SA range in T and therefore $a = c$. □

Let $\mathsf{RSA} = ([a_1, b_1], [a_2, b_2], \ldots, [a_{d'}, b_{d'}])$ be the list of the d' unique SA ranges for the d text segments, where $1 \leq d' \leq d$, ordered by the start position

of the range (*i.e.,* $a_1 < a_2 < \cdots < a_{d'}$). (Note that by Lemma 29 each start position, a_i, is distinct, for $1 \leq i \leq d'$.) If a text segment has SA range $[a_i, b_i]$ we say it has an `sa_id` of i, *i.e.,* the `sa_id` specifies the relative order of text segment SA ranges when they are sorted by their start position. We will use the `sa_id` of text segments as an index into some of our subsidiary data structures comprising F. Note that the value of any particular `sa_id` will be in the range $[1, d']$ as there are exactly d' distinct strings among the set of all d text segments. The list RSA is only conceptual. It simplifies our discussion of the components of F and also the operations performed on F; however, it does not need to be stored to perform queries. We will make use of it when constructing the data structure and give details in Section 8.6.

8.4.3 L: text segment lengths

Let $\mathsf{L}[i]$ be the length of text segments with `sa_id` i. Since L stores at most $d' \leq d$ lengths that sum to some value $n' \leq n$, then it can be stored as a compressed integer array using $O(d \log n)$ bits by Lemma 26.

Lemma 30 summarizes how we can use the length and SA range of any text segment to determine if it is a prefix of a given text X (and vice versa).

Lemma 30. Let $[c, d]$ be the non-empty SA range for a string X. Then (i) text segments with SA range $[a_j, b_j]$ (the j^{th} range in RSA) are a prefix of X if and only if $a_j \leq c \leq d \leq b_j$ and $|X| \geq \mathsf{L}[j]$. Similarly, (ii) X is a prefix of all text segments with SA range $[a_j, b_j]$ if and only if $c \leq a_j \leq b_j \leq d$.

Proof. Consider proposition (i). We let T_j be any text segment with SA range $[a_j, b_j]$; T_j therefore has length $|T_j| = \mathsf{L}[j]$. Suppose that T_j is prefix of X. Then it must be the case that $|T_j| \leq |X|$. By definition $T[\mathsf{CSA}[a_j]..|T|]$ ($T[\mathsf{CSA}[b_j]..|T|]$) is lexicographically smaller (greater) than any other suffix of T prefixed by the string T_j; thus, $[a_j, b_j]$ must enclose $[c, d]$ and we have $a_j \leq c \leq d \leq b_j$. Next suppose $a_j \leq c \leq d \leq b_j$ and $|T_j| \leq |X|$. Since $[a_j, b_j]$ encloses $[c, d]$ they must share a common prefix of length $\min(|X|, |T_j|)$. If $[a_j, b_j] = [c, d]$ it could be the case that X is a proper prefix of T_j; however, since $|X| \geq |T_j|$ by supposition then X and T_j must share a common prefix of length at least $|T_j|$. Thus, T_j is a prefix of X.

Consider proposition (ii). Suppose that X is a prefix of T_j. By definition $T[\mathsf{CSA}[c]..|T|]$ ($T[\mathsf{CSA}[d]..|T|]$) is lexicographically smaller (greater) than any other suffix of T prefixed by the string X; thus, $[c, d]$ must enclose $[a_j, b_j]$ and we have $c \leq a_j \leq b_j \leq d$. Next, suppose that $c \leq a_j \leq b_j \leq d$. Since $[c, d]$ encloses $[a_j, b_j]$ they must share a common prefix of length $\min(|X|, |T_j|)$. Since the character following T_j is a special character in the set $\{\phi, \$\}$, we know that $|T_j|$ cannot be a proper prefix of X and therefore $|T_j| \geq |X|$. Thus, T_j and X share a prefix of length $|X|$ making X a prefix of T_j. □

8.4.4 LEX, $\mathsf{M^B}$, $\mathsf{M^E}$, E: text segment SA range representation

The following lemma formalizes the notion that text segments with the same `sa_id` can be identified by a contiguous range of `lex_id`s.

Lemma 31. If an SA range is common to $k > 0$ distinct text segments, then those text segments form a contiguous range of k `lex_id`s.

Proof. If $k = 1$ then the condition is trivially met. Suppose $k > 1$. Proof by contradiction. Suppose only k text segments share the common SA range and they do not form a contiguous range of k `lex_id`s. Recall that `lex_id`s are assigned according to the lexicographic rank of all suffixes of T that are prefixed by a text segment. Let a and b be the minimum and maximum `lex_id` of the k text segments sharing the common SA range, respectively. By assumption that the k `lex_id`s do not form a contiguous range, then $b - a + 1 > k$. Therefore, there must exist a text segment with `lex_id` c, $a \leq c \leq b$, that is not one of the k text segments sharing the common SA range. By definition of `lex_id`, the text segment with `lex_id` c is lexicographically equal to or larger than the text segment with `lex_id` a. Similarly, it is smaller or equal to the text segment with `lex_id` b. Since the text segments with `lex_id` a and b share a common SA range, and since SA ranges cannot cross, then the text segment with `lex_id` c must also share the same SA range. Contradiction. □

We construct a simple array LEX to store, as a 2-tuple, the range of `lex_id`s associated with each `sa_id`. Formally, set $\mathsf{LEX}[i] = \langle j, k \rangle$ if the text segments with `lex_id`s $j, j+1, \ldots, k$ have the common `sa_id` i. We can represent the $d' \leq d$ entries in $O(d \log d)$ bits.

We next construct two bit vectors, $\mathsf{M^B}$ and $\mathsf{M^E}$, each of length n, to mark the beginning and end, respectively, of SA ranges in RSA. Formally, set $\mathsf{M^B}[a_i] = 1$, and set $\mathsf{M^E}[b_i] = 1$, for $1 \leq i \leq d'$. All other entries in the bit vectors have value 0. By Lemma 25, both can be represented in $O(d \log n) + o(n)$ bits. Since each SA range in RSA must begin at a distinct position, then $\mathsf{M^B}$ contains d' bits set to 1 and can therefore be used to count the number of ranges in RSA that begin prior to some position p (*i.e.,* $cnt = \mathtt{rank}_1(\mathsf{M^B}, p-1)$). However, ranges in RSA can end at the same position. Therefore, the number of 1 bits in $\mathsf{M^E}$ is some value d'', $1 \leq d'' \leq d'$, and we cannot use $\mathsf{M^E}$ to directly count the number of ranges that end prior to some position p. For this reason, we make use of an additional array E and set $\mathsf{E}[i] = |\{b_j \mid b_j \leq b_i, 1 \leq j \leq i\}|$, for $1 \leq i \leq d''$. The array E keeps a cumulative count of the closed ranges up to and including the i^{th} bit marked 1 in $\mathsf{M^E}$. It can be stored in $O(d \log d)$ bits. This is sufficient information to complete our count query (*i.e.,* $cnt = \mathsf{E}[\,\mathtt{rank}_1(\mathsf{M^E}, p-1)\,]$).

8.4.5 BP: containment of text segment SA ranges

If a pattern P is prefixed by one or more text segments with SA range $r_i = [a_i, b_i]$, then it is also prefixed by text segments with SA ranges that enclose

r_i. We seek a means to efficiently identify these SA ranges. We now show how to create a forest to represent the *containment* relationship between text segment SA ranges. This forest is defined formally, in addition to its balanced parenthesis representation, however a straightforward algorithm for constructing the balanced parenthesis representation directly is presented in Section 8.6.

Consider a range $r_i = [a_i, b_i]$ from RSA with `sa_id` i. We define the *parent range* of r_i to be the smallest range r_j from RSA, $r_i \neq r_j$, that encloses it. We will call `sa_id` j the *parent* of `sa_id` i. We say that r_i is a *child range* of r_j. We will call `sa_id` i a *child* of `sa_id` j. If $r_j = [a_j, b_j]$ encloses $r_i = [a_i, b_i]$, then it must be the case that $a_j < a_i$ and therefore $j < i$. When no range from RSA encloses r_i, other than itself, then it has no parent range. Since SA ranges cannot cross, then each range in RSA has at most one parent range. We can formally describe this relationship as a forest. We create a node with label i representing `sa_id` i, for $1 \leq i \leq d'$. We add an edge between nodes j and i, $j < i$, if and only if j is a parent of i. Since the label of a parent node is strictly less than the label for any of its children, then the direction of the relationship is always clear.

Any forest can be represented by a sequence of balanced parentheses. In particular begin with the lowest node label not yet visited and perform a pre-order traversal of the tree rooted at that node, outputting a left parenthesis when visiting a node for the first time, and outputting a right parenthesis when returning to a node from processing its children. (Children are visited in ascending order of their label value.) This process can be repeated for all unprocessed trees until the entire forest is represented as a sequence bp of balanced parentheses. Importantly, the i^{th} left parenthesis represents the `sa_id` i, for $1 \leq i \leq d'$. The sequence bp can be thought of intuitively in the following way. It has d' left parentheses that denote the start position of each SA range from RSA. The i^{th} left parenthesis denotes the start position of the i^{th} SA range, $[a_i, b_i]$, in RSA. Its corresponding right parenthesis denotes the end position. Any other left parenthesis between this pair denote other SA ranges, from RSA, that are enclosed by $[a_i, b_i]$. Thus the sequence bp uses at most $2d' \leq 2d$ parentheses to fully capture the containment relationship among all SA ranges from RSA.

We create BP, an indexed representation of bp, in order to support a number of useful operations. By Lemma 27, BP can be represented in $O(d)$ bits.

8.4.6 CNT: count of text segment prefixes

As stated in the previous section, if a pattern P is prefixed by one or more text segments with SA range $r_i = [a_i, b_i]$, then it is also prefixed by text segments with SA ranges that enclose r_i. To permit efficient counting queries, we create an array CNT of length d', such that CNT$[i]$ is the count of all text segments that enclose $r_i = [a_i, b_i]$ (inclusive of those with `sa_id` i). The count for each entry can be determined by an in-order traversal of the forest represented by BP and by using the LEX array to determine ranges of `lex_id`s (which can be used to determine counts for each `sa_id`). Specifically, the count of a child is the count of text segments with the same `sa_id` of the child in addition to the

count of the parent. As each entry sums to at most d, then the array CNT of $d' \leq d$ entries can be stored in $O(d \log d)$ bits.

8.4.7 Summary of full-text dictionary components

To aid in the discussion of supported operations, we list all subsidiary data structures that comprise the full-text dictionary F in Table 8.1.

Symbol	Description	Space (bits)
CSA	compressed enhanced suffix array of T	$\|CSA\| + O(n)$
L	array storing length of each text segment	$O(d \log n)$
LEX	array of `lex_id` ranges for each text segment SA range	$O(d \log d)$
M^{B}	bit vector marking beginning of text segment SA ranges	$O(d \log n) + o(n)$
M^{E}	bit vector marking end of text segment SA ranges	$O(d \log n) + o(n)$
E	array of cumulative count of closed text segment SA ranges	$O(d \log d)$
BP	balanced parentheses representation of text segment SA range containment	$O(d)$
CNT	count of text segments that prefix each text segment SA range	$O(d \log d)$

Table 8.1: Inventory of space usage for data structures comprising a full-text dictionary for a string T of length n containing d text segments.

Combining the space for the subsidiary data structures, we have the following.

Lemma 32. A string T over alphabet $\Sigma \cup \{\phi\}$ of length n, that contains d, not necessarily distinct, text segments over alphabet Σ, can be represented by a compressed full-text dictionary F in $|\mathsf{CSA}| + O(n) + O(d \log n)$ bits.

8.5 Using the full-text dictionary

We now describe how to support various operations on a full-text dictionary F using its subsidiary data structures as described in Section 8.4. Throughout this section, we assume that F is built for a string T of length n containing d text segments and that queries are with respect to a query pattern P. To simplify the description of the operations, we again define the ordered list of the unique SA ranges of text segments as we did when describing the components of F. Let $\mathsf{RSA} = ([a_1, b_1], [a_2, b_2], \ldots, [a_{d'}, b_{d'}])$ be the list of d' unique SA ranges for the d text segments, where $1 \leq d' \leq d$, ordered by the start position of the range. Note that list RSA is only conceptual and need not be stored to perform queries.

8.5.1 Pre-processing the pattern

When performing any of the query operations, we first calculate the *matching statistics* of P. Recall that the matching statistics for P with respect to T is an array ms of tuples such that $ms[i] = (q_i, [c_i, d_i])$ states that the longest prefix of $P[i..|P|]$ that matches anywhere in T has length q_i and SA range $[c_i, d_i]$. By Lemma 28 we can compute the matching statistics for P in time $O(|P| \log \sigma)$ using CSA.

8.5.2 Finding parent ranges and longest matches

We first develop some useful lemmas to simplify the description of our operations.

Lemma 33. Given any `sa_id` i the `sa_id` of its parent can be determined in $O(1)$ time. If `sa_id` i has no parent then 0 is returned in $O(1)$ time.

Proof. Let $l_i = \texttt{select}_(\mathsf{BP}, i)$ which gives the position, in BP, of the left parenthesis for `sa_id` i. Let $(l_j, r_j) = \texttt{enclose}(\mathsf{BP}, l_i)$. If the `enclose` operation returns undefined, then i has no parent and we return 0.

Otherwise, l_j is the position of the left parenthesis representing the parent of `sa_id` i. We can determine the actual `sa_id` value $j = \texttt{rank}_(\mathsf{BP}, l_j)$. We return j.

We perform a constant number of operations, all supported in $O(1)$ time. □

Lemma 34. Given a string X and its SA range $[c, d]$, the value i, such that text segments with `sa_id` i form the *longest* prefix match of X (of any text segment strings), can be returned in $O(1)$ time. If there is no text segment that prefixes X, then $i = 0$ is returned in $O(1)$ time.

Proof. We want to find the *longest* text segment string that prefixes X, if one exists. As a first candidate, we will determine the maximum i, such that $[a_i, b_i]$ encloses $[c, d]$. Let $b = \texttt{rank}_1(\mathsf{M}^{\mathrm{B}}, c)$ and $e = \mathsf{E}[\texttt{rank}_1(\mathsf{M}^{\mathrm{E}}, d_1 - 1)]$. These are, respectively, the number of SA ranges in RSA that begin up to and including position c, and the number that close prior to position d. Let $i = b - e$. Intuitively, i is the `sa_id` (the position in RSA) of the last SA range that began up to position c that does not close prior to position d. If $i = 0$ then X is not prefixed by a text segment and we return 0.

Otherwise, when $i > 0$, $[a_i, b_i]$ is the smallest range in RSA to enclose $[c, d]$. By Lemma 30 we must ensure that $|X| \geq L[i]$. If the condition is satisfied, we return i.

Otherwise, when $|X| < L[i]$, X is a proper prefix of text segments with `sa_id` i so they cannot be prefixes of X. However, if `sa_id` i has a *parent* with `sa_id` j we know that its SA range $[a_j, b_j]$ must enclose $[a_i, b_i]$ and $a_j < a_i$ (by Lemma 29). The range $[a_j, b_j]$ must therefore enclose $[c, d]$. We also know that $c > a_j$, since $c \geq a_i$. Therefore, if $[a_j, b_j]$ exists, then it represents text segments that are a proper prefix of X and therefore $L[j] < |X|$. By Lemma 30, text segments with `sa_id` j would be a prefix of X. We can find the parent for `sa_id` i, if it exists, in $O(1)$ time by Lemma 33. If it exists, we return its `sa_id`. Otherwise, we return 0.

Overall, we performed a constant number of operations all supported in $O(1)$ time. □

8.5.3 `dict_prefix`: report text segments that prefix P

With these results, we can now show how to implement the `dict_prefix` operation which reports all text segments that prefix a query pattern.

Lemma 35. Given a pattern P and its matching statistics, ms, the `dict_prefix` operation can return the `lex_id`s of all occ_1 text segments that prefix P in $O(1 + occ_1)$ time.

Proof. Using $ms[1] = (q_1, [c_1, d_1])$, by Lemma 34, we can find i, the `sa_id` for the text segments with the *longest* prefix match to $P[1..q_1]$ in $O(1)$ time. If $i = 0$ then there are no matches and we are done. Otherwise, we report the `lex_id`s in the range $\mathsf{LEX}[i]$. By Lemma 33, we can find j, the `sa_id` for the parent of i in $O(1)$ time, if it exists. If i does not have a parent ($j = 0$), then we are done. Otherwise, we report the `lex_id`s in the range $\mathsf{LEX}[j]$. We repeat this procedure for subsequent parents, until we no longer find a parent. Note that in this case, we will have done at most $occ_1 + 1$ operations to find a parent if there are occ_1 overall prefix matches to report. Therefore, the overall time[33] is $O(1 + occ_1)$. □

8.5.4 `dict_match`: report text segments contained in P

Reporting all text segments that match in P can be achieved by reporting all matches for each prefix of P using the `dict_prefix` operation.

Lemma 36. Given a pattern P and its matching statistics, ms, the `dict_match` operation can return the `lex_id`s of all occ_2 text segments that are substrings of P in $O(|P| + occ_2)$ time.

Proof. We use the `dict_prefix` operation $|P|$ times to report matches for each prefix, resulting in occ_2 overall matches. The overall time required is $O(|P| + occ_2)$. □

8.5.5 `dict_count`: counting text segments contained in P

Counting all text segments that match in P can be achieved by counting all matches for each suffix of P. This can be achieved by first identifying the `sa_id` for text segments that form the longest prefix match to a particular suffix of P and then looking up the count in the array CNT.

Lemma 37. Given a pattern P and its matching statistics, ms, the `dict_count` operation can count all occurrences of text segments that are substrings of P in $O(|P|)$ time.

Proof. Using $ms[i] = (q_i, [c_i, d_i])$, by Lemma 34, we can find j, the `sa_id` for the text segments with the *longest* prefix match to $P[i..q_i]$ in $O(1)$ time. If $i = 0$ then there are no matches. Otherwise, the count of matches is $\mathsf{CNT}[j]$. We can sum the counts for all i, $1 \leq i \leq |P|$ in $O(|P|)$ time. □

[33] If instead a succinct representation of the output giving ranges of `lex_id`s is acceptable, then the overall time to report can be bounded as $O(\min(|P|, occ_1))$ as we perform at most $O(\min(|P|, occ_1))$ operations to find a parent, since each parent is a proper prefix of its children.

8.5.6 `prefix`: report range of `lex_id`s that prefix P

In contrast to the `dict_prefix` operation which reports matches of text segments that prefix a pattern P, we now show how the range of `lex_id`s for text segments that contain P as a prefix can be determined. Note that the technique used to implement this operation can be performed using any CSA based on the Burrows Wheeler transform to determine `lex_id`s for the text segments. No additional data structures other than the CSA are required. We will use this technique in subsequent chapters.

Lemma 38. Given a pattern P and its matching statistics, ms, the `prefix` operation can return the range of `lex_id`s for text segments that contain P as a prefix in $O(1)$ time.

Proof. We will use $ms[1] = (q_1, [c_1, d_1])$. For every suffix i of T that begins a text segment, we have that $T^{\texttt{BWT}}[i]$ is either a ϕ or \$ character. We can use this property to determine the relative rank of every text segment contained in the SA range $[c_1, d_1]$ by inspecting $T^{\texttt{BWT}}[c_1, d_1]$. If there are k total ϕ characters and exactly one \$ character in T, which are lexicographically smaller than any other character in Σ, then the first $k+1$ rows of the suffix array for T are for suffixes that begin with a special character in $\{\phi, \$\}$. Let $t = \texttt{rank}_\phi(T, k+1) + \texttt{rank}_\$(T, k+1)$. Then, the pair $(lexid1+1, lexid2)$, where $lexid1 = \texttt{rank}_\phi(T, a-1) + \texttt{rank}_\$(T, a-1) - t$ and $lexid2 = \texttt{rank}_\phi(T, b) + \texttt{rank}_\$(T, b) - t$, denotes the range of `lex_id`s of text segments contained in the SA range $[c_1, d_1]$. □

8.5.7 `locate`: report positions in T containing P

Finding matches of P in the string T is already supported by the CSA data structure, giving us the following result.

Lemma 39. Given a pattern P and its matching statistics, ms, the `locate` operation can return the occ_3 positions in T that contain P as a prefix in $O(occ_3 \log^{1+\epsilon} |T|)$ time.

8.5.8 `match_stats`: finding the matching statistics of P

Finding the matching statistics of P with respect to F is already supported by the enhanced CSA data structure, giving us the following result.

Lemma 40. Given a pattern P the `match_stats` operation can determine the matching statistics of P with respect to F in $O(|P| \log \sigma)$ time.

The proof of Theorem 22 follows from Lemmas 28,32,35,36,3738,39 and 40.

8.6 Constructing the full-text dictionary

Construction of the overall full-text dictionary is straightforward and consists of the construction of subsidiary data structures, such as the compressed suffix

Algorithm 3: Constructing the *bp* sequence

Input: RSA specifies the SA ranges of text segments in order of their beginning position

Output: The balanced parentheses array *bp* representing the containment relationship of text segment SA ranges

```
1:  initialize an empty stack S
2:  for i = 1 ... d do
3:      (sp, ep) ← R[i]
4:      while S is not empty and sp > Top(S) do
5:          print ')'
6:          Pop(S)
7:      print '('
8:      Push(S, ep)
9:  while S is not empty do
10:     print ')'
11:     Pop(S)
```

array and marking of SA ranges of text segments. However, construction of the BP index cannot occur until *bp*, the sequence of balanced parentheses representing the containment relationship of SA ranges, is known. Below, we elaborate on how one may construct this sequence, prior to creating its index.

We first construct the array RSA from Section 29 which is the sorted list of unique SA ranges of all text segments. For each text segment $T_i \in D$, $1 \leq i \leq d$, we find its SA range and then append it into a temporary array t of length d. Finding the SA range for all text segments takes $O(n \log \sigma)$ time as their combined length is $O(n)$. The array t can then be sorted in $O(d \log n)$ time using the beginning position of each SA range as the sort key. The array RSA can then be determined from t. Moreover, by keeping the duplicate SA ranges in t, the `lex_id`s can be determined and can be used to create other auxiliary data structures.

Constructing the *bp* sequence which represents the containment relationship of text segment SA ranges is relatively straightforward and a procedure is given in Algorithm 3. The text segment SA ranges are processed in increasing lexicographical order. The algorithm ensures that right parentheses of intervals are appended to the *bp* sequence only after any contained intervals have been closed with right parentheses. This is accomplished with the use of a stack and by comparison of previously computed SA ranges stored in RSA. The stack stores at most d integers from $[1, n]$, thus the algorithm requires $O(d \log n)$ bits of working space. It is straightforward to see that the algorithm and construction of RSA can be accomplished in $O(n \log \sigma + d \log n)$ overall time and uses $O(d \log n)$ overall bits of temporary work space.

Chapter 9

Indexing text with wildcards

9.1 Introduction

We are interested in designing a compressed full-text index to answer a generalized version of the problem of aligning a pattern P of length m to a text T of length n, where T contains k wildcard positions that can match any character of P. Our motivation arises in the context of aligning short-read data, produced by high throughput sequencing technologies. Typically, short-reads are aligned against a so-called reference genome; however, the quantity of positions known to differ between individuals due to single nucleotide polymorphisms (SNPs) numbers in the millions [39]. Therefore, one canonical reference sequence is not representative of an entire population of individuals. Modeling SNPs as wildcards would yield more informed, and by extension, more accurate alignment of short-reads. While our motivation is grounded in biological sequence alignment, the solutions we propose in this chapter are more generally applicable to any problem benefiting the indexing of text containing wildcard characters.

Cole, Gottlieb & Lewenstein [24] were among the first to study the problem of indexing text sequences containing wildcards and proposed an index using $O(n \log^k n)$ words of space capable of answering queries in $O(m+\log^k n \log\log n+ occ)$ time, where occ denotes the number of matching positions. This result was later improved by Lam et al. [66] resulting in space usage of only $O(n)$ words and a query time no longer exponential in k. A key idea in their work was to build a type of dictionary of the text segments of $T = T_1\phi^{k_1}T_2\phi^{k_2}\ldots\phi^{k_d}T_{d+1}$ where each text segment T_i contains no wildcards and ϕ^{k_i} denotes the i^{th} *wildcard group* of size $k_i \geq 1$, for $1 \leq i \leq d \leq k$. In their result, the query time includes the term $\gamma = \sum_{i,j} \mathsf{prefix}(P[i..|P|], T_j)$ where $\mathsf{prefix}(P[i..|P|], T_j) = 1$ if T_j is a prefix of $P[i..|P|]$ and 0 otherwise. Intuitively, γ is the number of occurrences of text segments within P. Despite this improvement in query time complexity, $O(n)$ words of space can be prohibitive for texts as large as the Human genome. The use of dictionary matching of text segments within a pattern was also crucial in the approach of Tam et al. [124] who proposed the first succinct index that uses $(3+o(1))n\log\sigma$ bits. Using a compressed suffix array CSA, their space

Content from this chapter appears in the proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011) [127] and the journal Theoretical Computer Science [128].

complexity can be reduced to $3|\mathsf{CSA}| + O(d \log n)$ bits.

In our first contribution of this chapter, we show how to build on the full-text dictionary proposed in Chapter 8, to attain a succinct index using only $2|\mathsf{CSA}| + O(n) + O(d \log n)$ bits while maintaining the same query time complexity as the index of Tam et al. [124]. However, in our view, the main challenge that must be overcome for successful wildcard matching is a reduction of the query working space. The fastest solution of Tam et al. [124], matches our query time, if modified to use the same subsidiary data structures we use, but requires a query working space of $O(n \log d + m \log n)$ bits. Acknowledging that the first term is impractical for large texts, they give a slower solution that reduces the working space to be $O(n \log \sigma + m \log n)$. This makes the solution feasible, but constraining considering the fact that p parallel queries necessarily increase the working space by a factor of p. A main contribution of this chapter is an algorithm that reduces the query working space complexity significantly to $O(\min(dm, \gamma \log d) + m \log n)$ bits. For our motivating problem, alignment of short-reads to the Human genome (3 billion bases with 1-2 million SNPs), this reduces the working space by two orders of magnitude from gigabytes to tens of megabytes.

Finally, we show that by permitting an increase in worst case query time the space of the index can be reduced to only $nH_k(T) + o(n \log \sigma) + 2n + O(d \log n)$ bits. Existing solutions store a compressed suffix array for both T and another for its reverse, $\overline{T}$. The key to the space reduction is the elimination of the reverse index by exploiting a method used for bidirectional search [113]. Independently and in parallel with this work, Hon et al. showed an alternate approach to eliminate the reverse index [49]. This decreases the overall text index space term to $nH_k(T) + o(n \log \sigma) + O(d \log n)$ bits with an increase in query time compared to the fastest solution presented here (which uses more space). While the construction and use of their index to achieve these bounds is quite technical, their query time is faster than that of our smallest index. However, the ideas presented here and in the work of Hon et al. are complementary and can be combined to improve the overall state-of-the-art for this problem.

Our results for indexing text with wildcards are summarized and compared with existing results in Table 9.1. Results are also summarized when ideas from this work are combined with those of Hon et al. [49]. Details for combining the approaches are given in Section 9.6. For a fair comparison, the results of Tam et al. [124] have been adjusted to use the same subsidiary data structures used by our index.

Index Space		**Query Time**	**Query Working Space**		
$O(n \log^k n)$	words	$O(m + \log^k n \log\log n + occ)$	$O(1)$	words	[24]
$O(n)$	words	$O(m \log n + \gamma + occ)$	$O(n)$	words	[66]
$3\lvert\mathsf{CSA}\rvert + O(d \log n)$	bits	$O\left(\begin{array}{l} m\left(t_{\mathrm{LF}} + \min\left(m, \hat{d}\right)\log d\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \log d + \gamma \end{array}\right)$	$O(n \log d + m \log n)$	bits	[124]
		same as above with working space reduced by increasing query time			
$3\lvert\mathsf{CSA}\rvert + O(d \log n)$	bits	$O\left(\begin{array}{l} m\left(t_{\mathrm{LF}} + \min\left(m, \hat{d}\right)\log d\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \log d + \gamma \log_\sigma d \end{array}\right)$	$O(n \log \sigma + m \log n)$	bits	[124]
$2\lvert\mathsf{CSA}\rvert + O(n) + O(d \log n)$	bits	$O\left(\begin{array}{l} m\left(t_{\mathrm{LF}} + \min\left(m, \hat{d}\right)\frac{\log d}{\log\log d}\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \frac{\log d}{\log\log d} + \gamma \log \gamma \end{array}\right)$	$O(\gamma \log d + m \log n)$	bits	†
$2\lvert\mathsf{CSA}\rvert + O(n) + O(d \log n)$	bits	$O\left(\begin{array}{l} m\left(t_{\mathrm{LF}} + \min\left(m, \hat{d}\right)\frac{\log d}{\log\log d}\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \frac{\log d}{\log\log d} + \gamma \end{array}\right)$	$O(dm + m \log n)$	bits	†
$nH_k(T) + o(n \log \sigma) + 2n$ $+O(d \log n)$	bits	$O\left(\begin{array}{l} m^2 \log \sigma + m\left(\log n + \min\left(m, \hat{d}\right)\frac{\log d}{\log\log d}\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \frac{\log d}{\log\log d} + \gamma \end{array}\right)$	$O(dm + m \log n)$	bits	†
$nH_k(T) + o(n \log \sigma) + 2n$ $+O(d \log n)$	bits	$O\left(\begin{array}{l} m^2 \log \sigma + m\left(\log n + \min\left(m, \hat{d}\right)\frac{\log d}{\log\log d}\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \frac{\log d}{\log\log d} + \gamma \log \gamma \end{array}\right)$	$O(\gamma \log d + m \log n)$	bits	†
$nH_k(T) + o(n \log \sigma)$ $+O(d \log n)$	bits	$O\left(\begin{array}{l} m \log^{1+\epsilon} n + m\left(\min\left(m, \hat{d}\right)\log d\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \log d + \gamma \log \gamma \end{array}\right)$	$O((\gamma + m) \log n)$	bits	[49]
$nH_k(T) + o(n \log \sigma)$ $+O(d \log n)$	bits	$O\left(\begin{array}{l} m \log^{1+\epsilon} n + m\left(\min\left(m, \hat{d}\right)\frac{\log d}{\log\log d}\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \frac{\log d}{\log\log d} + \gamma \end{array}\right)$	$O(dm + m \log n)$	bits	‡
$nH_k(T) + o(n \log \sigma)$ $+O(d \log n)$	bits	$O\left(\begin{array}{l} m \log^{1+\epsilon} n + m\left(\min\left(m, \hat{d}\right)\frac{\log d}{\log\log d}\right) \\ +occ_1 \log^{1+\epsilon} n + occ_2 \frac{\log d}{\log\log d} + \gamma \log \gamma \end{array}\right)$	$O(\gamma \log d + m \log n)$	bits	‡

Table 9.1: A comparison of text indexes supporting wildcard characters in a text T over an alphabet of size σ containing d distinct groups of wildcards. $\lvert\mathsf{CSA}\rvert$ is the size of a subsidiary compressed suffix array implementation supporting `rank` queries in $O(t_{\mathrm{LF}})$ time. $\hat{d}$ is the # of distinct wildcard group lengths, occ_1, occ_2, occ are the # of occurrences containing no wildcard group, 1 wildcard group, and overall, respectively; $\gamma = \sum_{i,j} \mathsf{prefix}(P[i..|P|], T_j)$, † = our result, ‡ = our result combined with Hon et al. [49]

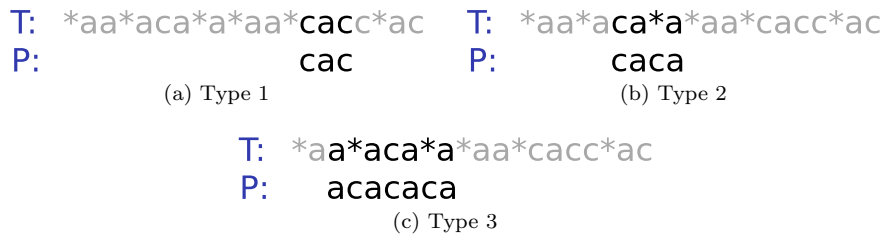
9.2 Preliminaries

Our wildcard matching algorithm makes use of an orthogonal range query data structure; specifically, it is an index for a set of two-dimensional points that can count and report the set of points contained inside a query rectangle.

Lemma 41 (Bose et al. [11])**.** A set d of points from universe $M = [1..d] \times [1..d]$ can be represented in $(1+o(1))d \log d$ bits to support orthogonal range reporting in $O((1 + occ)\frac{\log d}{\log \log d})$ time, where occ is the size of the output.

9.3 Overview of indexing text containing wildcards

Let T be a string over an alphabet $\Sigma \cup \{\phi\}$ of size σ where $\phi \notin \Sigma$ and $T[i] = \phi$ if and only if position i is a wildcard position in T. In particular, we denote the structure of the input string as $T = T_1\phi^{k_1}T_2\phi^{k_2} \dots \phi^{k_d}T_{d+1}$ where each text segment T_i contains no wildcards and ϕ^{k_i} denotes the i^{th} *wildcard group* of size $k_i \geq 1$, for $1 \leq i \leq d$. Our goal is to create an index for the purpose of identifying all the locations in T that exactly match any query pattern P, modulo wildcard positions. Similar to previous approaches [66, 124], we classify the match into one of three cases: the match of P contains no wildcard group (Type 1), the match of P contains exactly (some portion of) one wildcard group (Type 2), and the match of P contains more than one wildcard group (Type 3). See Figure 9.1 for examples of each of the three types of matches. Our solution for Type 2 matching is largely inspired by previous approaches [66, 124], so we give an overview of the approach but omit the details. Our algorithm for Type 3 matching is novel and can result in significantly reduced working space.



(a) Type 1 (b) Type 2

(c) Type 3

Figure 9.1: The three cases to consider when matching a pattern to a text with wildcards. Here and throughout this chapter, we will illustrate the wildcard character as '∗'.

9.4 Components of the text with wildcards index

Before detailing how we implement the three types of matching algorithms, we first give an overview of the subsidiary data structures that comprise our overall full-text index for text containing wildcards. The space complexity given for each component is assuming an input text T of length n having d groups of k overall wildcards. We will use $|\mathsf{CSA}|$ to denote the size (in bits) of a compressed suffix array for T.

9.4.1 F, R: indexing the text

We first build F, a compressed full-text dictionary of Chapter 8, for T. Among others, we will make use of the `dict_prefix` operation that reports all text segments from T that are a prefix of a query pattern P. From Theorem 22, F requires $|\mathsf{CSA}| + O(n) + O(d \log n)$ bits of space. We also construct a compressed suffix array R for $\overline{T}$, the reverse of T. The space required for R is $|\mathsf{CSA}|$ bits. Note that R does not need to support location reporting as our full-text dictionary F already does.

9.4.2 lex_ids, rlex_ids, and Π: text segment identifiers

By design, many of the operations supported by the full-text dictionary F report the match of a text segment T_j using its `lex_id` which is T_j's lexicographic rank among all text segments in T. We will also need to know the reverse lexicographic rank of each text segment (*i.e.,* when the reverse of each text segment are sorted lexicographically). The reverse lexicographic rank, or `rlex_id`, of a text segment T_j is equivalent to the lexicographic rank of $\overline{T_j}$ in R, the compressed suffix array for $\overline{T}$. The `rlex_id`s for each text segment with respect to R can be determined in the equivalent manner they are determined for F by the `prefix` operation as described in Section 8.5.6.

In addition, we also need to know the relative position of a text segment in T among all text segments in T. Therefore, we store a permutation $\Pi_{F \to P}$ mapping the `lex_id`s of text segments with respect to the forward index to their relative position order in T. For instance, if T_j has `lex_id` k, then $\Pi_{F \to P}[k] = j$. We do the same for mapping the `rlex_id` of text segments with respect to the reverse index to their relative position in T by creating the mapping $\Pi_{R \to P}$. The space required to store $\Pi_{F \to P}$ and $\Pi_{R \to P}$ is $O(d \log n)$ bits. We will use the relative position order as the main index into the other subsidiary data structures of our index.

9.4.3 RSA, $\overline{\mathsf{RSA}}$: storing SA ranges

We construct an array RSA of length $d + 1$ to store the SA ranges of each text segment with respect to F. For instance, $\mathsf{RSA}[j]$ specifies the SA range for text segment T_j. Similarly, we construct an array $\overline{\mathsf{RSA}}$ of length $d + 1$ to store the

SA ranges of the reverse of each text segment with respect to R. For instance, $\overline{\mathsf{RSA}}[j]$ specifies the SA range for $\overline{T_j}$, the reverse of text segment T_j. Both arrays can be stored in $O(d \log n)$ bits.

9.4.4 LEN, POS, WCS: auxiliary arrays

We find it convenient to store additional information for each text segment, in auxiliary arrays, indexed by the relative position order. We store the length of each text segment in an array LEN. Note that array LEN of the dictionary construction can be adapted to store lengths in this relative position order with the use of Π. We store the beginning position of a text segment in T (*i.e.,* its offset from the beginning of the string T) using the array POS. We store the size of the preceding wildcard group in the array WCS. Note that all arrays have length $d+1$ and overall require $O(d \log n)$ bits of space to support constant time access.

9.4.5 RQ: supporting range queries

We approach Type 2 matching no different than previous approaches [66, 124], by employing a number of 2D orthogonal range query data structures. Specifically, we will create a data structure RQ_i, for each $1 \leq i \leq \hat{d}$, where $\hat{d}$ is the number of unique lengths of wildcard groups that separate adjacent text segments. We will add a 2D point (i, j) into the data structure RQ_k if and only if the text segment with `lex_id` j is adjacent to the text segment with `rlex_id` i and they are separated by exactly k wildcard characters in T. For example, in Figure 9.1b, the two text segments that form the match with the pattern P would be represented by a point in the data structure RQ_1 as they are separated by a wildcard group having length 1. We use the data structure of Lemma 41 for each RQ_i. As there are exactly d total points added among all RQ_i then the total space required is $O(d \log d)$ bits.

9.4.6 Summary of the components

To aid in the discussion of supported operations, we list all subsidiary data structures that comprise our index for text containing wildcards in Table 9.2.

Combining the space of subsidiary data structures, we have the following.

Lemma 42. Given a text T of length n containing d groups of wildcards the combined space required of the above indexes is $2|\mathsf{CSA}| + O(n) + O(d \log n)$ bits.

9.5 Matching in text with wildcards

We now outline the various operations supported by our index. In particular, we give details on how each of the three types of matches can be determined.

Symbol	Description	Space (bits)
F	compressed full-text dictionary of T	$\lvert\mathsf{CSA}\rvert + O(n) + O(d \log n)$
R	compressed suffix array of $\overline{T}$	$\lvert\mathsf{CSA}\rvert$
$\Pi_{F \to P}$	a mapping from `lex_id` to relative position order of text segments	$O(d \log n)$
$\Pi_{R \to P}$	a mapping from `rlex_id` to relative position order of text segments	$O(d \log n)$
RSA	SA ranges for each T_j w.r.t. F	$O(d \log n)$
$\overline{\mathsf{RSA}}$	SA ranges for each $\overline{T_j}$ w.r.t. R	$O(d \log n)$
LEN	length of each text segment	$O(d \log n)$
POS	beginning position in T of each text segment	$O(d \log n)$
WCS	size of preceding wildcard group of each text segment	$O(d \log n)$
RQ_i	2D point data structure relating the `rlex_id` of a text segment to the `lex_id` of the text segment that follows it in T, when the two are separated by a wildcard group of length i, for $1 \leq i \leq \hat{d}$	$O(d \log d)$

Table 9.2: Inventory of space usage for data structures comprising an index for a text T of length n containing d groups of wildcards and $\hat{d}$ denotes the number of unique lengths of wildcard groups separating text segments.

9.5.1 Pre-processing the pattern

All three matching types make use of the matching statistics of P with respect to F. Types 2 and 3 matching also make use of the SA ranges for each suffix of $\overline{P}$ with respect to R. Both can be computed in $O(m \log \sigma)$ time (by Lemmas 24 and 28) and require $O(m \log n)$ bits to store. We incorporate these time and working space complexities into the results for each type.

9.5.2 `type1_match`: finding all type 1 matches of P

Type 1 matching corresponds exactly to the traditional pattern matching problem, where we must locate positions in T that contain P as a substring. Therefore, as our full-text dictionary F supports the `locate` operation, then type 1 matches can be identified in the time bounds specified in Theorem 22.

Lemma 43. All occ_1 Type 1 matches can be reported using $O(m \log n)$ bits of working space in $O(m \log \sigma + occ_1 \log^{1+\epsilon} n)$ time, for $\epsilon > 0$.

9.5.3 `type2_match`: finding all type 2 matches of P

A Type 2 match occurs when the alignment of P to T contains exactly (a portion of) one wildcard group. Our solution for Type 2 matching is the same as previous approaches [66, 124]. Therefore, we outline the high level idea of the approach for completeness, but omit details. First suppose that a match of P aligns with two text segments (and thus properly contains one wildcard group). Then, we seek a pair of neighbouring text segments T_j and T_{j+1}, separated by a wildcard group of size k_j, where $P[i..|P|]$ aligns to the first $|P| - i + 1$ characters of T_{j+1}—referred to as the *suffix match* (of P)—and $P[1..i-1-k_j]$ aligns to the last $i - 1 - k_j$ characters of T_j—referred to as the *prefix match*.

$$\cdots \text{——} T_j \text{——}\dashv \phi \ \cdots \ \phi \vdash\text{——} T_{j+1} \text{——} \cdots$$

By construction, the data structure RQ_{k_j} will contain a point (p, q) if and only if the text segment with `rlex_id` p is followed in T by the text segment with `lex_id` q. For a fixed suffix $P[i..|P|]$ and wildcard group length k_j our strategy will be to (i) find all potential suffix matches and record their `lex_id`s, (ii) find all potential prefix matches and record their `rlex_id`s, (iii) determine which candidate prefix matches are adjacent to a candidate suffix match in T, and (iv) report the matching text segments forming a match with P. Using F, $\texttt{prefix}(\mathsf{F}, P[i..|P|])$ will return the range of `lex_id`s, $[s_1, s_2]$, for the candidate suffix matches completing step (i). By using the same technique described in Section 8.5.6, we can determine the range of `rlex_id`s, $[r_1, r_2]$, for the candidate prefix matches of $P[1..i-1-k_j]$ with respect to R completing step (ii). Next, we can determine all pairs of prefix and suffix candidates that are adjacent in T, and are separated by a length k_j wildcard group, by determining all occ_2 points (x_l, y_l), for $1 \leq l \leq occ_2$, that are in the query rectangle $[r_1, r_2] \times [s_1, s_2]$, completing step (iii). Since, x_l, for $1 \leq l \leq occ_2$, gives the `rlex_id` for the text segment that forms a match with a prefix of P, then using $\Pi_{R \to P}$ we can determine the relative position of the matching text segments in T. Using POS the actual position within T for each match can be reported, completing step (iv).

In general, we can repeat the above procedure for every combination of suffix length and wildcard group length bound by m. However, as pointed out by Tam et al. [124] the number of distinct wildcard group sizes $\hat{d}$ is often a small constant, particularly in genomic sequences. We therefore only consider at most $\hat{d}$ lengths, provided they are not larger than m. Handling Type 2 matches that end or begin a match within a wildcard group is similar to the procedure outlined above. However, when we determine matches where the first k characters of P match within a wildcard group, we can query all RQ_l, for each $k \leq l \leq m$ (or for each of the $\hat{d}$ unique lengths if they are between length k and m), using $[1, d]$ as the range of `rlex_id`s, in conjunction with the actual range of `lex_id`s for the remaining suffix of P. This essentially performs a two-sided orthogonal range query. The analogous action can be taken when determining matches that end within a wildcard group.

Lemma 44. All Type 2 matches can be reported using $O(m \log n)$ bits of working space in $O(m(\log \sigma + \min(m, \hat{d}) \frac{\log d}{\log \log d}) + occ_2 \frac{\log d}{\log \log d})$ time.

9.5.4 `type3_match`: finding all type 3 matches of P

Type 3 matches contain at least (portions of) two wildcard groups and therefore must fully contain at least one text segment. The general idea in previous approaches and in this chapter is to consider this case as an extension of the dictionary matching problem: text segments contained within P are candidate positions, but we must verify if they can be extended to a full match of P.

Algorithm 4: Report Type 3 matches

Input: a string P of length m, SA ranges for all suffixes of P w.r.t. F, SA ranges for all suffixes of $\overline{P}$ w.r.t. R

Output: positions in T forming a Type 3 match with P, modulo wildcard positions

```
1: for i = m to 1 do
2:   for each lex_id returned by dict_prefix(F, P[i..m]) do
3:     j ← Π_{F→P}[lex_id]
4:     [a_p, b_p] ← SA range of P̄[1..i−1] w.r.t R
5:     [a_s, b_s] ← SA range of P[i + LEN[j] + WCS[j+1]..m] w.r.t F
6:     [c_p, d_p], [c_s, d_s] ← RSA̅[j−1], RSA[j+1]
7:     if j = d + 1 or LEN[j] + WCS[j+1] ≥ m − i + 1 then
       // Case 1: P does not contain T_{j+1}
8:       if LEN[j] + WCS[j+1] ≥ m − i + 1 or [a_s, b_s] encloses [c_s, d_s] then
         // Case 1: suffix condition satisfied
9:         if j = 1 or LEN[j−1] + WCS[j] > i − 1 then
           // Case 1a: P does not contain T_{j−1}
10:          if WCS[j] ≥ i − 1 or [a_p, b_p] encloses [c_p, d_p] then
             // Case 1a: prefix condition satisfied
11:            print match at position POS[j] − i + 1
12:        else
           // Case 1b: P must contain T_{j−1}
13:          set bit j of W[i] to 1
14:    else
       // Case 2: P must contain T_{j+1}
15:      if bit j + 1 of W[i + LEN[j] + WCS[j+1]] is set to 1 then
         // Case 2: suffix condition is satisfied
16:        if j = 1 or LEN[j−1] + WCS[j] > i − 1 then
           // Case 2a: P does not contain T_{j−1}
17:          if WCS[j] ≥ i − 1 or [a_p, b_p] encloses [c_p, d_p] then
             // Case 2a: prefix condition satisfied
18:            print match at position POS[j] − i + 1
19:        else
           // Case 2b: P must contain T_{j−1}
20:          set bit j of W[i] to 1
```

However, we execute this idea in an altogether novel manner by proposing a dynamic programming algorithm that can greatly reduce the working space over existing approaches.

Before giving details, we note that there is a trade-off in working space and query time of our dynamic programming algorithm depending on whether the dynamic programming table W (employed by the algorithm) uses bit vectors or sorted lists. W requires $O(dm)$ bits in the former case, and $O(\gamma \log d + m \log \gamma)$ bits in the latter, where γ is number of dictionary matches contained in the query pattern. If sorted lists are used, there is a $\log \gamma$ factor slowdown to the query time compared to using bit vectors. While in practice we expect γ to be small, in theory it can be as large as $O(dm)$ and therefore using bit vectors in these cases would result in a faster query algorithm and smaller working space. Fortunately, our approach allows us to perform a counting query to first determine the size of γ. Since we have available F, the full-text dictionary of T as described in Chapter 8, we can determine $\gamma = \texttt{dict\_count}(\mathsf{F}, P)$ in $O(|P| \log \sigma)$ time. Thus, we can always ensure the working space is $O(\min(\gamma \log d + m \log \gamma, dm))$. The complete details of our approach are given in Algorithm 4. We now highlight the main idea and give the intuition behind the correctness.

First, suppose that text segment T_j matches in P starting at position i. Consider the conditions that must be satisfied to confirm that this match can be extended to a complete match of P in T. We must verify that (i) $P[i + \mathsf{LEN}[j]..|P|]$ can be matched to the text following T_j in T — referred to as the *suffix condition* — and (ii) $P[1..i-1]$ can be matched to the text preceding T_j in T — referred to as the *prefix condition*. If both conditions are verified, we can report that P matches T beginning at position $\mathsf{POS}[j] - i + 1$.

$$\cdots \phi \vdash\!\!\!- T_{j-1} \dashv\!\!\!- \phi \cdots \phi \vdash\!\!\!- T_j \dashv\!\!\!- \phi \cdots \phi \vdash\!\!\!- T_{j+1} \dashv\!\!\!- \phi \cdots$$

For working space, we make use of an array W containing m entries, one for each suffix of P. We describe the algorithm assuming the use of bit vectors and comment in the proof on the effect of using sorted lists. Each of the m entries of W contain a bit vector of $d + 1$ bits (one for each text segment), with all entries set to zero using the constant time initialization technique [12]. During the course of the algorithm the j^{th} bit of $\mathsf{W}[i]$ is set to 1 if and only if the suffix condition is true for $P[i..m]$ with respect to T_j. Essentially, this entry would mean that the string $P[i..m]$ matches $T[\mathsf{POS}[j]..n]$ as a prefix. There are exactly m stages of the algorithm corresponding to the m suffixes of P. Each stage i considers a longer suffix of P ($i = m, m-1, \ldots, 1$). In a given stage i we consider each text segment T_j found to be a prefix of the i^{th} suffix of P. This can be accomplished using the `dict_prefix` operation of F.

To verify the prefix and suffix conditions for T_j we first consider (line 7 of Algorithm 4): will $P[i + \mathsf{LEN}[j]..m]$ need to fully contain the next text segment T_{j+1} in order to match in T? This breaks our algorithm into the two main cases.

If the match will not fully contain T_{j+1} (Case 1), we verify the suffix condition by checking whether $P[i + \mathsf{LEN}[j]..m]$ is compatible with the wildcard group to

its right and the prefix of T_{j+1} to which it must align (line 8). If the suffix condition is satisfied, we consider (line 9): will $P[1..i-1]$ need to fully contain the previous text segment T_{j-1} in order to match in T? If it does not need to fully contain the previous text segment T_{j-1} (Case 1a), we verify the prefix condition is satisfied by checking that $P[1..i-1]$ is compatible with the wildcard group to its left and the suffix of T_{j-1} to which it must align (line 10). If indeed the prefix condition is satisfied, we output a match (line 11). If it does need to fully contain the previous text segment T_{j-1} (Case 1b), we set the j^{th} bit of entry $\mathsf{W}[i]$ to 1, to indicate that a suffix condition holds for $P[i..m]$ with respect to T_j (line 13). The key idea here is that we only attempt to verify the prefix condition when T_j would be the last text segment to occur in P (*i.e.,* Case 1a) and if not (Case 1b), we record information in W stating that we currently have a partial match, but for it to remain viable, T_{j-1} should be a suffix of $P[1..i-\mathsf{WCS}[j]-1]$ which can be verified in a future stage of the algorithm.

Case 2 occurs when P must contain the next text segment T_{j+1} to satisfy the suffix condition (lines 14–20). Since stages of the algorithm proceed for longer suffixes of P, and thus decreasing values of i, then the suffix condition would have been previously checked and, if satisfied, the bit $j+1$ of $\mathsf{W}[i+\mathsf{LEN}[j]+\mathsf{WCS}[j+1]]$ would be set to 1. The remaining questions are answered as before: the prefix condition is verified if possible, and otherwise successful partial matches are again recorded in W.

Lemma 45. All Type 3 matches can be reported in $O(m \log \sigma + \gamma)$ time using $O(dm + m \log n)$ bits of working space, or reported in $O(m \log \sigma + \gamma \log \gamma)$ time using $O(\gamma \log d + m \log n)$ bits of working space.

Proof. Recall that the algorithm proceeds in m stages for decreasing $i = m, \ldots, 1$ for each suffix of P. It is clear in the algorithm description that verification of a match of T_j proceeds by first ensuring the suffix condition can be satisfied (Case 1: if P does not contain T_{j+1}) or ensuring it was previously satisfied (Case 2: P must contain T_{j+1}), and then verifying the prefix condition in the cases where P does not contain T_{j-1} (Cases 1a, 2a) (and reporting a match when verified), or by instead marking W to signify a partial match, expecting the match to be continued by a match of T_{j-1} in a future stage (Cases 1b, 2b). The correctness relies on showing that W is set correctly to confirm the satisfaction of the suffix condition for the next text segment (T_{j-1}) for a future time step. We show correctness by induction on i. Consider the base case when we are in stage $i = m$. All candidate text segments T_j fall into Case 1 which (importantly) does not rely on the correctness of previous stages of the algorithm. The suffix condition is trivially true. The prefix condition is split into two cases. The first case (Case 1a) is when a successful match of P will not contain T_{j-1}. This can be verified by checking if the appropriate prefix of P is a suffix of T_{j-1}, unless the prefix of P is fully matched by the preceding wildcard group. If the prefix of P matches, both conditions have been satisfied and we have an overall match that can be reported. If P must fully contain T_{j-1} for a successful match, then bit j of $\mathsf{W}[i]$ is set to denote that the suffix condition of $P[i..m]$ is satisfied with respect to T_j. Now assume we are in some stage i and the algorithm is correct

for all shorter suffixes (*i.e.,* stages $i+1,\ldots,m$). Case 1 is handled as before and does not rely on the correctness of previous stages, so assume we are in Case 2 (P must contain T_{j+1}). Then, if the suffix condition is satisfied bit $j+1$ of $\mathsf{W}[i+\mathsf{LEN}[j]+\mathsf{WCS}[j+1]]$ should be set to 1. This bit would have been set in an earlier stage $t>i$, and we have assumed the algorithm is correct for earlier stages (*i.e.,* $i+1,\ldots,m$). Therefore, it must be the case that the suffix condition for T_j is satisfied if and only if $\mathsf{W}[i+\mathsf{LEN}[j]+\mathsf{WCS}[j+1]]$ has bit $j+1$ set to 1. Similarly to before, if the suffix condition is satisfied, we can attempt to verify the prefix condition when P does not contain T_{j-1} or by recording the partial match in W when P must contain T_{j-1}. This completes the correctness proof.

We now consider the additional runtime and work space incurred for Type 3 matching. There are γ candidate positions overall that can be reported in $O(m\log\sigma+\gamma)$ time by Theorem 22. Each candidate is processed once, in $O(1)$ time when using bit vectors. The array W occupies $O(dm)$ bits as working space. Thus, the overall time complexity is $O(m\log\sigma+\gamma)$ and working space is $O(dm+m\log n)$ when using bit vectors. Clearly, one could also maintain a sorted list of text segment position ids for the m entries of W instead of a bit vector. Also, m pointers can be used to mark the head of each list. Since there are γ total entries of matching text segment position ids in all lists, and a text segment position id can be uniquely identified with $\lceil\log d+1\rceil$ bits, then the total space is $O(\gamma\log d)$ bits to store the sorted ids. Inserting an entry into the list or querying an entry takes at most $O(\log\gamma)$ time, compared with $O(1)$ time when using bit vectors. We note that the space to store the m pointers to the head of each sorted list use no more than $O(m\log\gamma)$ bits and is therefore absorbed into the $O(m\log n)$ term denoting the space to store the suffix array ranges. □

Combining the results for the 3 types of matching we arrive at our main result.

Theorem 23. A text T of length n containing d groups of wildcards, can be represented in $2|\mathsf{CSA}|+O(n)+O(d\log n)$ bits, to support the following operations given any query pattern P:

- `type1_match`(P): returns all occ_1 positions in T that match P, using no wildcard groups, in $O(m\log\sigma+occ_1\log^{1+\epsilon}n)$ time and $O(m\log n)$ bits of working space,
- `type2_match`(P): returns all occ_2 positions in T that match P, using (some portion of) one wildcard group, in $O(m(\log\sigma+\min(m,\hat{d})\frac{\log d}{\log\log d})+occ_2\frac{\log d}{\log\log d})$ time and $O(m\log n)$ bits of working space,
- `type3_match`(P): returns all occ_3 positions in T that match P, using (some portion of) two or more wildcard groups, in $O(m\log\sigma+\gamma)$ time and $O(m\log n+dm)$ bits of working space, or $O(m\log\sigma+\gamma\log\gamma)$ time and $O(m\log n+\gamma\log d)$ bits of working space,

where $\sigma = |\Sigma \cup \{\phi\}|$, $|\mathsf{CSA}|$ denotes the size of any compressed suffix array of T supporting the LF operation in time $O(\log \sigma)$, $\hat{d}$ is the number of unique wildcard group lengths, and γ is the number of text segments that match in P, and $\epsilon > 0$.

9.6 Less haste, less waste: reducing the space further

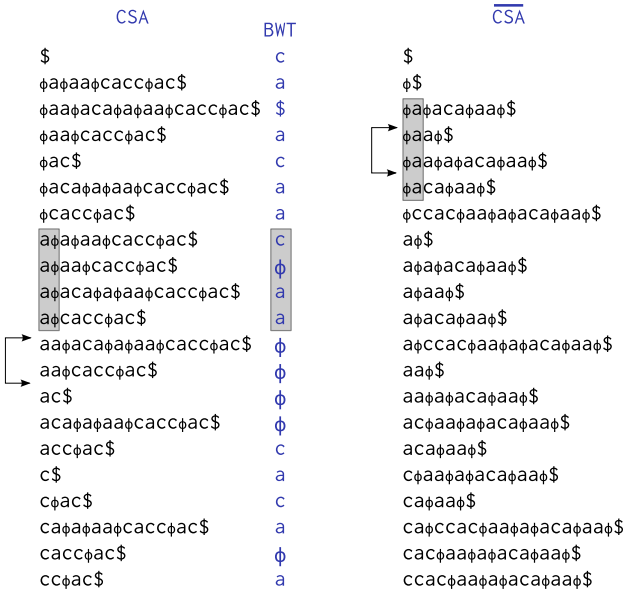


Figure 9.2: Shown is a compressed suffix array for a text $T = \phi aa\ \phi aca\ \phi a\ \phi aa\ \phi cacc\ \phi ac$ and a compressed suffix array for the reverse of T. The shaded intervals denote the SA range of a query $a\phi$ in the forward index and corresponding SA range of ϕa in the reverse index. Using backward search the SA range in the forward index can be updated for the pattern $aa\phi$, and by leveraging information in $T^{\mathtt{BWT}}$ the corresponding SA range for ϕaa can be updated in the reverse index. Both new SA ranges are shown demarcated with arrows. See the text for details.

Letting $|\mathsf{CSA}|$ denote the size of a subsidiary compressed suffix array, our index requires $2|\mathsf{CSA}| + O(n) + O(d \log n)$ bits in comparison to that of Tam et al. [124]—the first succinct index for this problem—which requires $3|\mathsf{CSA}| + O(d \log n)$ bits. For alphabets such as proteins ($\sigma = 20$) or larger this can result in a substantially smaller index. However, for small alphabets such as DNA ($\sigma = 4$), the $O(n)$ term becomes quite significant. This term arises from the need to store auxiliary data structures for determining lcp parent intervals

when computing matching statistics of a query string. Ohlebusch and Gog [90] proposed a solution that computes parent intervals in constant time (for $\sigma = O(1)$) and has been demonstrated to use between $3n$–$5n$ bits in practice [91]. This approach would ensure no slowdown in query time at the expense of a larger index compared to that of Tam et al. for the DNA alphabet. Using a solution by Fischer et al. [35] we can store the necessary lcp information using at most $2n + o(n)$ bits. This would yield an index of roughly the same size as that of Tam et al. when $\sigma = 4$; however, it incurs a logarithmic slowdown (in n) when computing parent intervals. Specifically, the time to pre-process the pattern becomes $O(m \log n)$ as at most m parent intervals must be computed for the m suffixes of P.

In either case, both our index and that of Tam et al. store a compressed suffix array for both the text and its reverse. An interesting question is whether we can eliminate the suffix array of the reverse text. Doing so would lead to a substantial space reduction, regardless of alphabet size. We now show that this question can be answered in the affirmative. First, consider how the reverse index is used. In order to determine if some prefix $P[1 \ldots i]$ of a pattern P is a suffix of a text segment, a compressed suffix array R of the reverse text is searched using $\overline{P[1 \ldots i]}$ as the query (*cf.* Section 9.4.1). The resulting matches would form a contiguous interval in the reverse index. This property allows for easy verification of a partial match given a suffix array range of a query (in the reverse index) and is the basis for the orthogonal range query data structure relating the forward and reverse lexicographic order of text segments.

Note that if $\overline{P[1 \ldots i]}$ has a non-empty SA range $[a, b]$ in R, then there is a non-empty SA range $[c, d]$ in F for the query $P[1 \ldots i]$ and $d-c = b-a$. Recently, Schnattinger et al. [113] demonstrated that with the use of a compressed suffix array based on a wavelet tree, one can perform *bidirectional search*. Specific to our example, by performing an incremental backward search of the query $P[1 \ldots i]$ in F, the SA range for $\overline{P[1 \ldots i]}$ in R can also be updated incrementally, *without performing any queries on* R.

Since the idea is central to our space reduction, we now give the intuition of the method; however, the reader is referred to Schnattinger et al. [113] for the details. Shown in Figure 9.2 are two compressed suffix arrays: one for the text $T = \phi aa\phi aca\phi a\phi aa\phi cacc\phi ac$ (CSA) and one for $\overline{T}$ ($\overline{\mathsf{CSA}}$). Suppose we wish to locate the SA range of a query string $X = aa\phi$ in CSA and the corresponding SA range of $\overline{X}$ in $\overline{\mathsf{CSA}}$. The shaded regions represent the SA ranges matching the suffix $X[2 \ldots |X|]$ and the corresponding match of $\overline{X[2 \ldots |X|]}$ in $\overline{CSA}$. Given the SA range of $X[2 \ldots |X|]$, the SA range of X in CSA, shown demarcated by arrows, can be determined by backward search in the usual manner as we are prefixing the currently matched pattern by one character. However, finding the SA range of $\overline{X}$ requires suffixing the currently matched pattern by one character. Therefore, the SA range of $\overline{X[2 \ldots |X|]}$ must contain the SA range of X in $\overline{\mathsf{CSA}}$.

Since all suffixes of $\overline{T}$ are in sorted order in $\overline{\mathsf{CSA}}$, it follows that if we knew how many suffixes of $\overline{T}$, prefixed by $\overline{X[2 \ldots |X|]}$, were (i) lexicographically less than $\overline{X}$, and (ii) how many were lexicographically greater, then we could exactly determine the correct SA range of $\overline{X}$ in $\overline{\mathsf{CSA}}$. Schnattinger et al. [113]

demonstrated that we can answer both questions by exploiting the relationship of the $T^{\texttt{BWT}}$ string to $\overline{CSA}$. Continuing with our example, for any character α in the shaded region of $T^{\texttt{BWT}}$, there must exist a suffix of T prefixed by $\alpha X[2 \ldots m]$. Furthermore, $\alpha X[2 \ldots m]$ must prefix some suffix of $\overline{T}$. Therefore, to determine the number of suffixes of $\overline{T}$, prefixed by ϕa, that are lexicographically less than ϕaa, we can simply count the number of characters less than a in the shaded region of $T^{\texttt{BWT}}$. In this case, one character (ϕ) is less than a. Similarly, we discover one character (c) is lexicographically larger than a. Given these values, we can update the SA range in $\overline{\mathsf{CSA}}$ to the interval demarcated by the arrows. Importantly, it was not necessary to perform any query on $\overline{\mathsf{CSA}}$ to determine the correct SA range in $\overline{\mathsf{CSA}}$. Thus, $\overline{\mathsf{CSA}}$ does not need to be constructed in the first place.

By using the technique of Schnattinger et al. [113], and without any modification to the data structures, we can determine the SA range of the reverse of every text segment with respect to the reverse index R, *without having constructed* R, by performing queries only on F since it is backed by a compressed suffix array. This can be computed for all text segments in $O(n \log \sigma)$ time. Furthermore, for a pattern P of length m, we can compute the SA range of $\overline{P}$ in R in $O(m \log \sigma)$ time. It follows that we can determine the corresponding SA ranges for all m prefixes of P in $O(m^2 \log \sigma)$ time.

By modifying the result of Theorem 23 to use the lcp representation of Fischer et al. [35], using the compressed suffix array of Lemma 24, and by employing bidirectional search as described above, we have the following result.

Theorem 24. A text T of length n containing d groups of wildcards, can be represented in $nH_k(T)+o(n \log \sigma)+2n+O(d \log n)$ bits, to support the following operations given any query pattern P:

- `type1_match`(P): returns all occ_1 positions in T that match P, using no wildcard groups, in $O(m \log \sigma + occ_1 \log^{1+\epsilon} n)$ time and $O(m \log n)$ bits of working space,
- `type2_match`(P): returns all occ_2 positions in T that match P, using (some portion of) one wildcard group, in $O(m(\log \sigma + \min(m, \hat{d}) \frac{\log d}{\log \log d}) + occ_2 \frac{\log d}{\log \log d})$ time and $O(m \log n)$ bits of working space,
- `type3_match`(P): returns all occ_3 positions in T that match P, using (some portion of) two or more wildcard groups, in $O(m \log n + m^2 \log \sigma + \gamma)$ time and $O(m \log n + dm)$ bits of working space, or $O(m \log n + m^2 \log \sigma + \gamma \log \gamma)$ time and $O(m \log n + \gamma \log d)$ bits of working space,

where $\sigma = |\Sigma \cup \{\phi\}|$, $H_k(T)$ denotes the k^{th} order empirical entropy of T (for any $k \geq 0$), $\hat{d}$ is the number of unique wildcard group lengths, γ is the number of text segments that match in P, and $\epsilon > 0$.

Independently and in parallel with this work, Hon et al. showed that two sparse suffix trees can be used in conjunction with an FM-index for the forward

index in order to eliminate the reverse index [49]. This decreases the overall index space to $nH_k(T) + o(n \log \sigma) + O(d \log n)$ bits with an increase in query time compared to the fastest solution presented here (which uses more space). In particular, their approach has the following time and space bounds.

Theorem 25 (Hon et al. [49])**.** Given a text T of length n containing d groups of wildcards, an index of $nH_k(T) + o(n \log \sigma) + O(d \log n)$ bits of space can be built to report all matches of a pattern P of length m using $O((m + \gamma) \log n)$ bits of working space in $O(m(\log^{1+\epsilon_1} n + \min(m, \hat{d}) \log d) + occ_1 \log^{1+\epsilon_2} n + occ_2 \log d + \gamma \log \gamma)$ time, where $\epsilon_1 > 0$, $\epsilon_2 > 0$, $\hat{d}$ is the number of unique wildcard group lengths, and γ is the number of matching text segments in P.

While the construction and use of the sparse suffix trees for the claimed space and query time is quite technical—we refer the reader to their paper [49] for the details—their query algorithm does not suffer from the m^2 term that is necessary in the worst case for the smallest index described above in this work and is therefore faster except for degenerate cases. However, ideas from both the Hon et al. approach and this work are complementary and can be combined to improve indexing text with wildcards. For instance, the dynamic programming algorithm for Type 3 matching in this work results in reduced working space and a query time that can be faster, and is the same in the worst case, when compared with the Hon et al. approach. Combining ideas from both indexes, we can achieve the following result.

Theorem 26. A text T of length n containing d groups of wildcards, can be represented in $nH_k(T) + o(n \log \sigma) + O(d \log n)$ bits, to support the following operations given any query pattern P:

- `type1_match`(P): returns all occ_1 positions in T that match P, using no wildcard groups, in $O(m \log \sigma + occ_1 \log^{1+\epsilon_1} n)$ time and $O(m \log n)$ bits of working space,
- `type2_match`(P): returns all occ_2 positions in T that match P, using (some portion of) one wildcard group, in $O(m(\log^{1+\epsilon_2} n + \min(m, \hat{d}) \frac{\log d}{\log \log d}) + occ_2 \frac{\log d}{\log \log d})$ time and $O(m \log n)$ bits of working space,
- `type3_match`(P): returns all occ_3 positions in T that match P, using (some portion of) two or more wildcard groups, in $O(m \log^{1+\epsilon_2} n + \gamma)$ time and $O(m \log n + dm)$ bits of working space, or $O(m \log^{1+\epsilon_2} n + \gamma \log \gamma)$ time and $O(m \log n + \gamma \log d)$ bits of working space,

where $\sigma = |\Sigma \cup \{\phi\}|$, $H_k(T)$ denotes the k^{th} order empirical entropy of T (for any $k \geq 0$), $\hat{d}$ is the number of unique wildcard group lengths, γ is the number of text segments that match in P, $\epsilon_1 > 0$, and $\epsilon_2 > 0$.

Finally, we note that different time and space trade-offs can be achieved simply by using different subsidiary text indexes and range reporting data structures. Recently Belazzougui and Navarro showed, for the first time, that a

compressed text index can be constructed that supports query times independent of alphabet size [7]. In particular, their result can be used to improve the $m \log \sigma$ term to m for query time by introducing an $O(n)$ term of additional index space. Any other self-index, that supports the LF operation, with different space and time trade-offs could also be used in the approaches discussed here. Furthermore, the $O(d \log n)$ term may be significantly smaller than the size of the compresses text, if for instance, the text does not contain many wildcards groups. In these cases, the overall query time can be improved by using non-succinct range reporting structures [17].

Chapter 10

Indexing hypertext

10.1 Introduction

Much more progress has been made in mapping reads from genome data to reference genomes than on aligning reads derived from transcriptomes. The latter problem is harder by the very nature of the events it is capable of capturing compared to genomic sequencing. Since introns are spliced from genes in the process of transcription (see Figure 10.2), *spliced reads* may map to two regions of the genome that are separated by many hundreds or thousands of bases. The difficulty of aligning NGS reads that span intron boundaries is exacerbated by their short length, and often is not attempted, resulting in a significant loss of information. When compared with aligning reads to a reference text, the transcriptome read alignment problem is modeled more accurately by the problem of aligning patterns to a *hypertext*.

Informally, hypertext is a generalization of text from a linear structure to a directed graph, $G = (V, E)$, with each node being a fragment of text and edges implying which fragments of text can be appended; thus, any path in the graph is a substring of the hypertext. An example of a hypertext is given in Figure 10.1. The example transcriptome in Figure 10.2 consists of five overall exons between two genes. The splicing events, and valid transcripts are also shown. The resulting hypertext model of this transcriptome has a node for each exon, and an edge between exons joined by a splicing event, resulting in two components (one for each gene).

The seminal work on pattern matching in hypertext is due to Manber and Wu [77] who proposed a $O(|V| + m|E| + occ \log\log m)$ time algorithm, where m is the length of the pattern and occ are the number of matches. Akutsu [1] proposed an $O(n)$ algorithm for matching in hypertext forming a tree structure, where n is the total length of text in all nodes. Park and Kim [95] considered the case where the hypertext forms a directed acyclic graph by proposing a $O(n + m|E|)$ time algorithm, under the assumption that no node in G matches to more than one position in the pattern. Amir et al. [2] proposed an algorithm with the same runtime complexity; however, theirs was the first algorithm for the case of hypertext forming a general graph. Amir et al. [2] and Navarro [87] also considered the problem of approximate matching in hypertext.

In all cases, the runtimes of the previously proposed pattern matching algo-

Content from this chapter appears in the proceedings of the 18th Annual International Conference on String Processing and Information REtrieval (SPIRE 2011) [130].

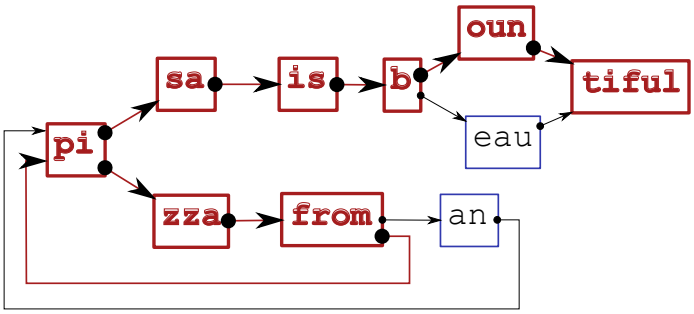


Figure 10.1: A example of a hypertext. A query matches within a hypertext if and only if it can be aligned as a path through the graph. A path shown in bold matches the query pattern *pizzafrompisaisbountiful*.

rithms in hypertext are impractical for alignment of millions of transcriptome reads as they are at least linear in the size of the hypertext. Surprisingly, no index for hypertext, succinct or otherwise, has been previously proposed. In this work, we propose a succinct index to model hypertext. Our index can model any hypertext forming a general graph and makes no restriction to the topology. We also propose a new exact pattern matching algorithm, capable of aligning a pattern to any path in the hypertext, that is especially efficient for hypertexts where few nodes share common prefixes or when all nodes are of constant degree. In particular, our new algorithm can report all patterns crossing at most one edge—a valid assumption for current transcriptome read datasets—in $O(m \log \sigma + m \frac{\log |V|}{\log \log |V|} + occ_1 \log^{1+\epsilon} n + occ_2 \frac{\log |V|}{\log \log |V|})$ time, where occ_1 (occ_2) is the number of matches that cross no (one) edge. We also consider a restricted version of the problem, where only certain paths in the hypertext are considered valid and also prove the worst case query time complexity is improved for other restrictions including graph topology. A main contribution of this chapter is to show the correspondence between the hypertext matching problem and the problem of matching text containing wildcards. As we will show, the former can be viewed as a generalization of the latter. In particular, recent strategies for indexing text with wildcards are applicable for indexing hypertext. Improvement to one problem may immediately lead to improvements of the other.

While our results in this chapter are general and relevant to applications that are appropriately modeled by a hypertext, our original motivation was to better model the transcriptome read alignment problem. We view the results in this chapter as a theoretical contribution towards that end. However, the reads produced by current sequencing technologies contain *sequencing errors*—errors introduced during the sequencing process—in addition to the genetic variation expected between the experimental sequence and a reference sequence. A significant challenge that must be overcome, before these approaches could yield practical tools for transcriptome read alignment, is to efficiently support

approximate pattern matching queries.

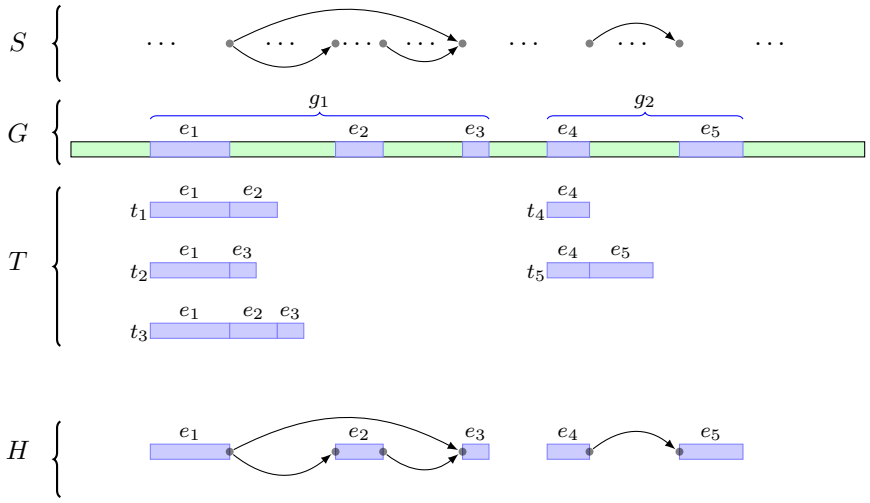


Figure 10.2: A simple genome, G, is shown having five exons contained in two genes. Exons are strings over the four letter alphabet of DNA. Below is the corresponding transcriptome, T, which consists of five transcripts. Transcripts are formed by the concatenation of certain exons from G. Above is the splicing graph, S, where each of the five nodes correspond to one of the five exons from G, and each directed edge denotes splicing events (concatenation of exons) that are found in T. A hypertext model H for the transcriptome is also shown.

10.2 Preliminaries

In developing our hypertext index, we will use the same notation and leverage many existing results from the literature listed in Section 8.2 and Section 9.2. The remainder of this section details an existing result and definitions specific to developing our hypertext index.

10.2.1 Succinct graph representation

The succinct graph representation of Farzan & Munro supports a number of graph topology query operations in $O(1)$ time using the best space achievable [31]. In this application, we only require the use of efficient adjacency queries. Their result is stated in terms of Boolean matrices supporting access queries. In terms of graphs, this is equivalent to determining adjacency of nodes using an adjacency matrix.

Lemma 46 (Farzan & Munro [31])**.** A Boolean matrix of size $n \times n$ with m ones can be represented in $(1+\epsilon)\lg\binom{n^2}{m}$ bits for any constant $\epsilon > 0$, while supporting access (and successor) queries in $O(1)$ time.

	F^{BWT}
$	t
ϕacaϕgϕgaϕcgϕct$	$
ϕcgϕct$	a
ϕct$	g
ϕgϕgaϕcgϕct$	a
ϕgaϕcgϕct$	g
aϕcgϕct$	g
aϕgϕgaϕcgϕct$	c
acaϕgϕgaϕcgϕct$	ϕ
caϕgϕgaϕcgϕct$	a
cgϕct$	ϕ
ct$	ϕ
gϕct$	c
gϕgaϕcgϕct$	ϕ
gaϕcgϕct$	ϕ
t$	c

	R^{BWT}
$	c
ϕacaϕgϕagϕgcϕtc$	$
ϕagϕgcϕtc$	g
ϕgϕagϕgcϕtc$	a
ϕgcϕtc$	g
ϕtc$	c
aϕgϕagϕgcϕtc$	c
acaϕgϕagϕgcϕtc$	ϕ
agϕgcϕtc$	ϕ
c$	t
cϕtc$	g
caϕgϕagϕgcϕtc$	a
gϕagϕgcϕtc$	ϕ
gϕgcϕtc$	a
gcϕtc$	ϕ
tc$	ϕ

Figure 10.3: (left) An example of the underlying suffix array and BWT string for the forward index F of the text $T = \phi aca\phi g\phi ga\phi cg\phi ct\$$, representing the serialization of possible text in exons $e_1, \ldots, e_5$, supposing those five exons consist of the five sequences $\{aca, g, ga, cg, ct\}$ respectively, from Figure 10.2. (right) The underlying suffix array and BWT string for the reverse index R of the text $\overline{T} = \phi aca\phi g\phi ag\phi gc\phi tc\$$.

10.2.2 Hypertext

A hypertext generalizes the notion of text to be a directed graph $G = (V, E)$ such that each node $v \in V$ contains text over an alphabet Σ and the outgoing edges of v are incident to nodes containing text that can follow v's. A match of a pattern P to the hypertext G is a path $p = v_1, \ldots, v_k$ through G, and an offset l into the first node v_1, such that P matches the concatenation of the text in nodes $v_1, \ldots, v_k$, beginning at position l in v_1, and ending at some prefix of v_k.

Problem 11. (Pattern matching in hypertext)
Instance: A hypertext G and a pattern P
Question: Which paths in G match P?

Previous algorithms for matching in hypertext focused on reporting only the initial node, and offset within that node, of paths in G matching P [1, 2, 95]. For our motivating problem of aligning patterns to a transcriptome, the actual path is required to be known, and that is our focus in the remainder of the paper. However, our matching algorithm can be simplified if only the initial node of a match (and the offset within the node) is desired.

10.3 Construction of the hypertext index

The succinct hypertext index is a collection of three sets of data structures: those indexing the node text, those indexing the graph topology, and useful auxiliary structures. In our pattern matching algorithms we find it useful to identify nodes of the graph by two different identifiers: the `lex_id`, and the `rlex_id`. This is reflected in our descriptions of the data structures below. The `lex_id` gives the prefix lexicographic rank[34] of the text contained within the node as compared with all other nodes in V. Similarly, the `rlex_id` gives the rank with respect to the suffix lexicographic rank. We show how these ids can be determined in Section 10.3.3; their meaning is exactly as in the previous two chapters. However, for many hypertext applications there is a *canonical id* associated with each node, giving an absolute ordering of nodes, that should be used for reporting matches. This is the case, for instance, when modeling a transcriptome where each node represents an exon that can be identified by their order in a reference genome.

Our description below for indexing a hypertext will focus on minimizing the query time of the general case. The representation is redundant as both the text and graph topology is represented twice. In section 10.5, we show how space can be reduced by increasing the worst case query time.

10.3.1 Indexing node text

For a given hypertext $G = (V, E)$ over an alphabet Σ of size σ, we construct a text $T = \phi v_1 \phi v_2 \phi \dots \phi v_{|V|}\$$, of length n, that is a serialization of the combined text of the nodes of V, each prefixed by a character ϕ, such that $\$ < \phi < c, \forall c \in \Sigma$. We assume the nodes are concatenated in order of their canonical id. We will construct and store the *full-text dictionary* index F of T that was proposed in Chapter 8. We also construct and store a compressed suffix array R for $\overline{T}$, the serialization of the reverse of all node text. We let $\mathsf{F}^{\mathtt{BWT}}$ ($\mathsf{R}^{\mathtt{BWT}}$) denote the BWT string for F (R). See Figure 10.3 for an example. Note that it is not necessary to store T or $\overline{T}$. The full-text dictionary provides functionality for determining all nodes contained within a given pattern (dictionary matching problem), all nodes that contain a given pattern (pattern matching problem), and also which nodes contain a suffix of a pattern as a prefix. The reverse index provides the functionality to determine which nodes contain a prefix of a pattern as a suffix. In order to determine the canonical id of a node that properly contains a match of a query pattern we utilize a fully indexable dictionary (FID) similar to applications for document listing [110]. Specifically, we build D, the FID of Lemma 25 to represent a bit vector having the same length as T with 1 bits demarcating the ending of patterns in T. The full-text dictionary can report the absolute position of a match in T. Determining the number of 1 bits, prior to this position in the FID, using a `rank` query, will yield the canonical id of the matched node. If the match is within the node with canonical id 0, then the

[34] Prefix lexicographic rank compares characters left to right, whereas suffix lexicographic rank compares characters right to left.

offset is simply the position within T. Otherwise, it is the distance to a previous position marked with a 1 bit.

10.3.2 Storing graph topology

We store the graph topology twice. First, we construct a 2D range query index, P, that is heavily utilized for matching patterns crossing a single edge. Conceptually, the x-axis corresponds to `lex_id`s and the y-axis corresponds to `rlex_id`s. A point (a, b) is added to the index if and only if in E there is an edge from the node with `rlex_id` b to the node with `lex_id` a. Unfortunately, the 2D range query structure does not permit us to determine if two specific nodes are adjacent in $O(1)$ time; therefore, we construct Q using the succinct graph representation of Farzan & Munro [31] for this purpose. Specifically, all nodes in Q are referenced by their `lex_id` and an entry in row a and column b is set to 1 if and only if in E the node with `lex_id` a has an outgoing edge to the node with `lex_id` b.

10.3.3 Auxiliary data structures

Each node can be ranked according to its prefix lexicographic order in the forward index F. For instance, we can determine the prefix lexicographic order of all $|V|$ nodes by performing backward search on the serialized text T. After the text '$v_{|V|}$\$' has been matched in F, three facts are known: (i) the matching suffix array range $[a, b]$ will be a size one interval (*i.e.,* $a = b$) since \$ occurs only in one position of T, (ii) $\mathsf{F}^{\mathtt{BWT}}[a] = \phi$ since each node is prefixed by a ϕ character, and (iii) the rank of the ϕ character at position a in $\mathsf{F}^{\mathtt{BWT}}$ corresponds to the prefix lexicographic rank (*i.e.,* its `lex_id`) of node $v_{|V|}$, with respect to all other nodes in V, due to the properties of the LF mapping. We can continue to determine the prefix lexicographic order of all nodes in V in a single traversal of T. Similarly, we can determine the suffix lexicographic order (the `rlex_id`s) of all nodes using the reverse index. We find it convenient to store a permutation $\Pi_{R \to F}$ that maps the `rlex_id` of a node to its `lex_id`. Furthermore, since we report all matches with respect to the canonical id label, we also store a permutation $\Pi_{F \to C}$ that maps `lex_id`s to canonical ids. Our overall space usage for a general hypertext is summarized in Table 10.1 and Lemma 47. We explore how some of these data structures can be removed in Sections 10.5 and 10.6.

Symbol	Description	Space (bits)
F	full-text dictionary of forward text	$nH_h(T) + o(n \log \sigma) + n(2 + o(1)) + \lvert V\rvert(\lceil \log \frac{n}{\lvert V\rvert} \rceil + 2 + o(1))$
R	compressed suffix array of reverse text	$nH_h(T) + o(n \log \sigma)$
D	FID used to identify canonical id from position in T	$\lvert V\rvert \log \frac{n}{\lvert V\rvert} + O(\lvert V\rvert + n \frac{\log \log n}{\log n})$
P	2D point structure containing graph topology	$(1 + o(1))\lvert E\rvert \log \lvert E\rvert$
Q	succinct graph representation	$(1 + \epsilon) \log \binom{n^2}{m}$
$\Pi_{R \to F}$	mapping of `rlex_id` to `lex_id`	$\lvert V\rvert \lceil \log \lvert V\rvert \rceil$
$\Pi_{F \to C}$	mapping of `lex_id` to canonical id	$\lvert V\rvert \lceil \log \lvert V\rvert \rceil$

Table 10.1: Inventory of space usage for succinct index of a general hypertext. Sections 10.5 and 10.6 explore the removal of various components of the overall index.

Lemma 47. A hypertext $G = (V, E)$ can be represented in $2nH_k(T)+o(n \log \sigma)+n(2+o(1))+O(|E| \log |E|)$ bits by the above scheme, where the text of the nodes in V are over an alphabet of size σ and have a combined length of $n - |V|$.

10.4 Pattern matching in the hypertext index

We now demonstrate that extensions of techniques used to solve the problem of matching a pattern against a text containing wildcards from Chapter 9 are applicable and effective for matching a pattern in a hypertext. In a solution to the wildcard matching problem, Lam et al. [66] classified a match of a pattern P to the text T into three cases: P matches a position in T containing no wildcard group, P matches a position in T containing one wildcard group, and P matches a position in T containing more than one wildcard group, where a wildcard group is a consecutive sequence of wildcard characters. We will solve the problem of matching a pattern P in a hypertext $G = (V, E)$ by considering three analogous cases: (i) P does not span any edge from E (Type 1), (ii) P spans exactly one edge from E (Type 2), and (iii) P spans more than one edge from E (Type 3). As we will see, case (i) is identical, case (ii) is a restriction, and case (iii) is a generalization of the respective wildcard cases. An example of each case is given in Figure 10.4.

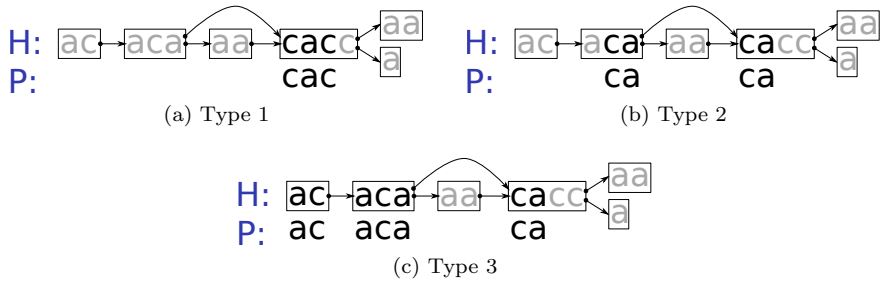


Figure 10.4: The three cases to consider when matching a pattern to a hypertext.

10.4.1 Preprocessing the pattern

Considering that a match of P is a path through G, we need an efficient means to determine which nodes contain P as a substring, which are contained within P, which nodes contain a prefix of P as a suffix, and which contain a suffix of P as a prefix. Consider for a moment how we may determine which nodes contain the suffix $P[i \ldots m]$ as a prefix. Suppose $P[i \ldots m]$ has a non-empty SA range $[a, b]$ in the forward index F. If $P[i \ldots m]$ is a prefix of some node v_i then two things must be true: (i) $[c, d]$, the SA range of v_i in F, must be a sub-interval of $[a, b]$, and (ii) $\mathsf{F}^{\mathtt{BWT}}[c \ldots d]$ must contain a ϕ character corresponding to v_i, since all node texts are prefixed by the ϕ character in the construction of F. Therefore, by

determining the rank of the first and the last ϕ characters in $\mathsf{F}^{\mathtt{BWT}}[a \dots b]$, we will determine a range of `lex_id`s corresponding to nodes that contain $P[i \dots m]$ as a prefix. Using backward search we can determine the range of matching `lex_id`s, if any, for each suffix of P in $O(m \log \sigma)$ time (by Lemma 24). To determine which nodes contain a prefix of P as a suffix, we can instead determine which nodes, when their text is reversed, contain a suffix of $\overline{P}$, the reverse of P, as a prefix. Therefore, by performing a backward search of $\overline{P}$ in the reverse index R, a range of `rlex_id`s can be determined in $O(m \log \sigma)$ time (by Lemma 24). Finally, we also determine the matching statistics of $P[1 \dots m]$ and $P[2 \dots m-1]$ with respect to F in $O(m \log \sigma)$ time (by Theorem 22). The matching statistics for $P[1 \dots m]$ are used to determine matches of the pattern within nodes, while the matching statistics for $P[2 \dots m-1]$ are used to determine matches of nodes contained within $P[2 \dots m-1]$. The `lex_id` and `rlex_id` ranges for every prefix and suffix of P as well as the matching statistics for $P[1 \dots m]$ and $P[2 \dots m-1]$ can be stored in $O(m \log n)$ bits.

10.4.2 Matching within a node

If a match of P in G does not span an edge, then P must match as a substring of some node. Let $(q, [a, b])$ be the matching statistics of $P[1 \dots m]$ calculated in the preprocessing step. If $q = m$ then there exists at least one match of P as a substring of a node of G. Furthermore, the SA range $[a, b]$ is non-empty and P is contained as a substring of one or more nodes exactly $b - a + 1$ times. In each instance, using the full-text dictionary F, we can determine the location of the match, in T, in $O(\log^{1+\epsilon} n)$ time. Using the fully-indexable dictionary D, we can determine the canonical id of the matching node, as well as the offset within that node.

Lemma 48. For a pattern P of length m, the occ number of matches of P within a node of G can be counted in $O(m \log \sigma)$ time. Their locations can be reported in an additional $O(occ \log^{1+\epsilon} n)$ time. The working space is $O(m \log n)$ bits.

10.4.3 Matching across a single edge

If a match of P in G spans a single edge, then there must exist some i, $1 < i \leq m$, such that $P[1 \dots i-1]$ is a suffix of some node $v_j \in V$, $P[i \dots m]$ is a prefix of some node $v_k \in V$, and the edge $(j, k) \in E$. In the preprocessing step, the range $[c, d]$ of `lex_id`s corresponding to nodes that contain $P[i \dots m]$ as a prefix, as well as the range $[a, b]$ of `rlex_id`s corresponding to nodes that contain $P[1 \dots i-1]$ as a suffix, were stored. Similar to other applications in stringology [23, 59, 124, 127], we make use of the range query data structure to relate the two ranges. If both ranges are non-empty, we can determine exactly which pairs of ids are connected by a directed edge by reporting all points in P contained in the range $[a, b] \times [c, d]$. See Figure 10.5 for an example. The auxiliary structures $\Pi_{R \to F}$ and $\Pi_{F \to C}$ can be used to report the canonical ids of matches.

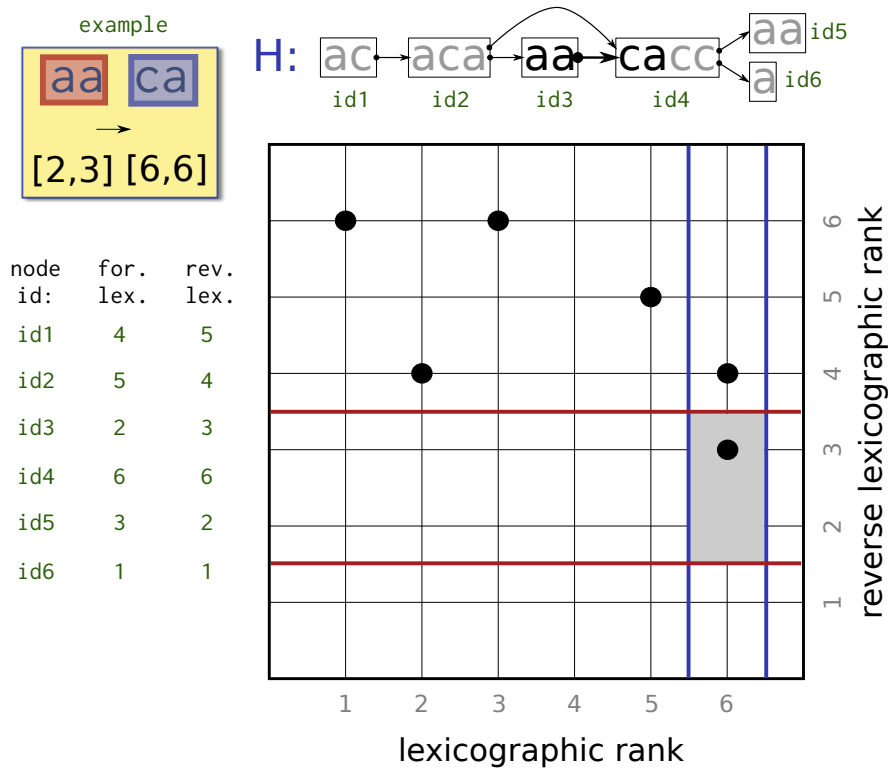


Figure 10.5: An example of the pattern `aaca` matching across a single edge in a hypertext. The pattern suffix `ca` prefixes nodes with lexicographic rank (`lex_id`) in the range $[6, 6]$ while the pattern prefix `aa` suffixes nodes with reverse lexicographic rank (`rlex_id`) in the range $[2, 3]$. Points in the query rectangle $[6, 6] \times [2, 3]$ are type 2 matches. A point (a, b) appears in the grid if and only if a node with lexicographic rank a has an incoming edge from a node with reverse lexicographic rank b.

Lemma 49. For a pattern P of length m the *occ* number of matches of P that cross a single edge of G can be counted in $O(m \log \sigma + m \frac{\log |V|}{\log \log |V|})$ time. Their path descriptions can be reported in an additional $O(occ \frac{\log |V|}{\log \log |V|})$ time. The working space is $O(m \log n)$ bits.

10.4.4 Matching across multiple edges

If a match of P in G spans more than one edge, then $P[2 \ldots m-1]$ must contain at least one node of G as a substring. The strategy here, as in previous solutions to the text with wildcard problem [66, 124, 127], is an extension of the dictionary matching problem: first identify all γ nodes contained within $P[2 \ldots m-1]$, and second, for each of those γ candidate matches, determine if it can be extended into a full match of P in G. Consider a candidate match of a node v_j, with `lex_id` j, to the substring $P[i \ldots i+k-1]$, where $i > 1$ and $i+k-1 < m$. This candidate can be extended into a full match if both the *suffix condition*—there exists a path leaving v_j matching $P[i+k \ldots m]$ —and the *prefix condition*—there exists a path ending at v_j matching $P[1 \ldots i-1]$—are satisfied.

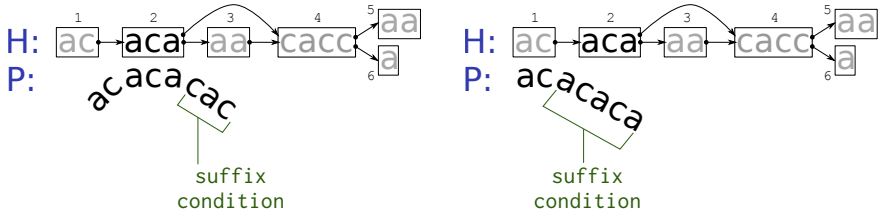
Recall the dynamic programming algorithm to solve the corresponding text with wildcards version of the problem from Chapter 9. The algorithm works in m stages, by considering successively longer suffixes of P, and in the process determines if the suffix and prefix conditions of candidate matches are satisfied. We will adapt this algorithm for our purposes here. However, we note that verifying the suffix (prefix) condition in the hypertext problem is more challenging as we must consider any path beginning (ending) at a node representing a candidate match. In the text with wildcards problem, one need only verify the text immediately preceding (succeeding) a candidate match position. This can be viewed as verifying in a hypertext that forms a single path. For this reason, the algorithm we propose below is a generalization of the original algorithm.

Overview of the algorithm

Conceptually, the algorithm will consider successively longer suffixes of $P[2 \ldots m-1]$. Specifically, for each suffix $P[i \ldots m-1]$, for $i = m-1, m-2, \ldots, 2$, the algorithm will consider all γ_i nodes of G that prefix $P[i \ldots m-1]$ using the full-text dictionary F. Each of these γ_i nodes is a *candidate* requiring the suffix condition to be first verified, and if successful, the prefix condition is tested. The algorithm will maintain a compact list of all sub-paths of G that are matched by $P[i \ldots m]$. The head of the list for suffix i will be stored at $\mathsf{W}[i]$, a working space array maintained during the search. In later stages of the algorithm, we will determine if these sub-paths can be extended to match longer suffixes of P. Overall, there are γ candidate positions that will be evaluated. Note that the exact count of candidates, γ, can be determined using F, prior to the first stage of the algorithm, in $O(m \log \sigma)$ time using a the `dict_count` operation. This permits us to allocate sufficient working space. Our algorithm attempts to track all matching sub-paths in as little working space as possible. We describe

the information tracked during the course of the algorithm, and comment at the end on the overall working space complexity. In what follows, we describe how the suffix and prefix conditions are verified for a candidate node v_j that matches a k length prefix of $P[i \ldots m-1]$. This same procedure will be used to verify all γ candidates, across all suffixes of $P[2 \ldots m-1]$.

Verifying the suffix condition



(a) Sub-path initiation event

(b) Sub-path extension event

Figure 10.6: The two cases to consider when verifying the suffix condition in type 3 matches. (a) The suffix match can form a sub-path initiation event with node 4. (b) The suffix match can form a sub-path extension event with node 2.

We must verify that there exists a sub-path in G matching $P[i+k \ldots m]$ that begins at some node v_t such that $(j,t) \in E$. There are two cases to consider: such a sub-path is a prefix of v_t (and thus ends within v_t), or it properly contains v_t. We will refer to the former as a *sub-path initiation event*, and the latter as a *sub-path extension event*. For each candidate node v_j we must consider both types.

To verify an initiation event, we first determine the range $[a,b]$ of `lex_id`s corresponding to nodes that contain $P[i+k \ldots m]$ as a prefix. This range was stored in the preprocessing step. Then, we must determine if any of those nodes have an incoming edge from node v_j. Suppose that node v_j has `rlex_id` j. A counting query in the range $[a,b] \times [j,j]$ of P determines cnt_{init}, the number of matching nodes. Specifically, cnt_{init} is the number of sub-paths that originate at node v_j, match $P[i \ldots m]$, and end within a node connected to v_j.

An extension event implies that $P[i+k \ldots m]$ must match a sub-path that contains, but does not end within, a node connected to v_j. Therefore, to verify an extension event, we must determine if any putative sub-paths stored in the list at $\mathsf{W}[i+k]$ begin with a node v_t such that $(j,t) \in E$. (Note that if one or more of these sub-paths do exist, they would have been stored in $\mathsf{W}[i+k]$ at an earlier stage of the algorithm.) For each of the at most γ entries in $\mathsf{W}[i+k]$, we use the `lex_id` of its initial node, and the `lex_id` of v_j, to perform an adjacency query using Q. Note that since we only need to establish adjacency between nodes, then no assumption on the graph topology is made and therefore any directed graph (possibly cyclic) is handled correctly. Let cnt_{ext} be the number of entries in $\mathsf{W}[i+k]$ that are connected by an edge from v_j.

If the suffix condition is satisfied we append a new putative sub-path entry to the list at $\mathsf{W}[i]$ and associate with it v_j as the *initial node*. If $cnt_{\text{ext}} > 0$, then we associate with that entry a list of the cnt_{ext} offsets that denote the entries in $\mathsf{W}[i+k]$ which are connected to v_j and form new putative sub-path matches for $P[i \ldots m]$. We also associate with each entry the count of sub-paths it begins. Consider that each of the cnt_{ext} entries in $\mathsf{W}[i+k]$ connected to v_j may represent many sub-paths. The number of sub-paths each represents is stored in the entry. Therefore, the number of sub-paths represented by our new entry is the sum of the counts for these previous cnt_{ext} entries, plus cnt_{init}.

Verifying the prefix condition

If the suffix condition is satisfied for a candidate node v_j, we can test the prefix condition. We need to determine if there exists one or more nodes that contain $P[1 \ldots i-1]$ as a suffix and have an outgoing edge to node v_j. In the preprocessing step, we stored the range $[a, b]$ of `rlex_id`s corresponding to nodes that contain $P[1 \ldots i-1]$ as a suffix. The cnt_p number of matching nodes can be found by querying the range $[j, j] \times [a, b]$ in P. If $cnt_p > 0$, and the current count of sub-paths beginning at v_j that match $P[i \ldots m]$ is cnt_s, then G contains $cnt_p \times cnt_s$ paths matching P.

Reporting all matching paths

Whenever a prefix condition is verified, all matching paths can be enumerated by a simple backtracking procedure using the information previously stored in W and the new prefix matches. The point data structure P is once again queried to determine the `lex_id`s of nodes that contain the end of a matching path. These can be translated into canonical ids for reporting.

Lemma 50. For a pattern P of length m, the *occ* number of matches of P that cross more than one edge in G, can be determined in $O(m \log \sigma + \gamma^2 + \gamma \frac{\log |V|}{\log \log |V|})$ time. Their path descriptions can be reported in an additional $O(h + occ \frac{\log |V|}{\log \log |V|})$ time, where h is the total number of nodes in all *occ* sub-paths matched by P. The working space is $O(m \log n + \gamma^2 \log \gamma + \gamma \log |V|)$ bits.

Proof. For each candidate node v_j, the suffix condition must be verified by checking for both initiation events and for extension events. When verifying initiation events, a range query on P is performed in $O(\frac{\log |V|}{\log \log |V|})$ time and there are at most $O(\gamma)$ initiation events. When verifying extension events for v_j, $O(\gamma)$ previous entries representing putative sub-paths are considered for extension by v_j. For each putative sub-path, an adjacency query is performed in $O(1)$ time using the graph representation of Q. Thus, verifying extension events for v_j takes $O(\gamma^2)$ time. The suffix condition can be verified for all γ candidate nodes in $O(\gamma^2 + \gamma \frac{\log |V|}{\log \log |V|})$ time. Verifying the prefix condition is analogous to verifying suffix initiation events and it takes $O(\gamma \frac{\log |V|}{\log \log |V|})$ time to

verify the at most γ candidates that satisfy the suffix condition. This yields an overall worst case time $O(m \log \sigma + \gamma^2 + \gamma \frac{\log |V|}{\log \log |V|})$.

The working space includes the $O(m \log n)$ bits from the preprocessing step, and $O(\gamma \log |V|)$ bits to store counts of putative sub-paths and initial nodes of those sub-paths. However, the working space is dominated by storing back pointers (offsets) in the putative sub-path entries of the working array W to each previous entry that forms a valid sub-path match for a suffix of P. There are $O(\gamma)$ entries in W. Thus, each can be uniquely identified with $\lceil \log \gamma \rceil$ bits. In the worst case, each entry stores $O(\gamma)$ back pointers giving an overall working space of $O(m \log n + \gamma \log |V| + \gamma^2 \log \gamma)$ bits. Importantly, we note that the total number of matches γ can be determined by F in a preprocessing step in $O(m \log \sigma)$ time using a counting query in order to allocate sufficient working space. □

Combining Lemmas 47 through 50 we have our main result of the chapter. We note that our query time is dependent on γ, the number of occurrences of nodes as substrings of the query pattern P. The query algorithm is designed under the assumption that, in practice, γ is expected to be proportional to the length of the pattern, P. As we discuss in Section 10.6, this can be shown in the worst case when no node is the prefix of another. However, when many nodes share a common text, and all match in numerous positions of the query pattern, then the query time becomes dominated by this parameter. The approach proposed here is inefficient in these cases and is not faster than pattern matching without an index.

Theorem 27. A hypertext $G = (V, E)$ can be represented in $2nH_k(T) + o(n \log \sigma) + n(2 + o(1)) + O(|E| \log |E|)$ bits, where the text of the nodes in V are over an alphabet of size σ and have a combined length of $n - |V|$, such that all matches of a pattern P of length m can be counted in $O(m \log \sigma + m \frac{\log |V|}{\log \log |V|} + \gamma^2 + \gamma \frac{\log |V|}{\log \log |V|})$ time, and reported in an additional $O(occ_1 \log^{1+\epsilon} n + (occ_2 + occ_3) \frac{\log |V|}{\log \log |V|} + h)$ time, where occ_1 is the number of matches within a node of G, occ_2 is the number of matches crossing a single edge, h is the total number of nodes in all occ_3 sub-paths matching P that cross more than one edge, and γ is the number of occurrences of a node as a substring of the pattern P. The working space is $O(m \log n + \gamma^2 \log \gamma + \gamma \log |V|)$ bits.

10.5 Reducing the index space

The index described in Section 10.3 is redundant in order to decrease worst case query time for a general hypertext, or support more general match reporting. If no predetermined canonical id needs to be reported for matching nodes, then the `lex_id`s can be used and the permutation $\Pi_{F \to C}$ can be eliminated. In this case, the topology stored in Q will be in terms of `lex_id`s as well. The topology of the graph need not be stored twice. The Q representation was used to enable $O(1)$ time adjacency queries; however, the point data structure P can

perform adjacency queries in $O(\frac{\log |V|}{\log \log |V|})$ time. This increases the γ^2 term in the time complexity of the pattern matching algorithm to $\gamma^2 \frac{\log |V|}{\log \log |V|}$. Finally, in addition to indexing the node text, the reverse of each node text is also indexed in order to determine which nodes contain a prefix of a query pattern as a suffix. Very recently, Hon et al. showed that a sparse suffix tree can be used for this purpose to calculate the appropriate suffix array ranges [49]. This decreases the text index space term from $2nH_k(T)$ to $nH_k(T) + O(|V| \log n)$ bits; however, the time increases from $O(m \log \sigma)$ to $O(m \log n)$ time.

10.6 Considering restricted hypertext

In this section, we consider a number of interesting yet relevant restrictions to general hypertext. All restrictions apply to our motivating example of modeling transcriptomes as hypertext.

10.6.1 Path constraints

While our motivating problem of aligning transcripts to transcriptomes is better modeled using a hypertext index than a linear text index, the model does not completely capture all necessary information. Specifically, the hypertext models the splicing graph; however, the set of valid transcripts in the transcriptome is a set $\mathcal{P}$ of paths through the splicing graph. Not every path in the splicing graph is necessarily a valid transcript. We now show how we can easily extend our index to only report matches of a pattern P if they are a sub-path of at least one path in $\mathcal{P}$.

For illustrative purposes, assume we have constructed a hypertext index for the example in Figure 10.2 and all transcripts are valid, expect for t_3. Further suppose the node labels in the hypertext correspond to the exon labels in the figure. Then the set of valid paths are $\mathcal{P} = \{[e_1, e_2], [e_1, e_3], [e_4], [e_4, e_5]\}$. We construct a serialization of these paths as a string $S = \phi e_1 e_2 \phi e_1 e_3 \phi e_4 \phi e_4 e_5$, over the integer alphabet $[1 \dots |V|] \cup \{\phi\}$, where $\phi = 0$. Next, we create a compressed suffix array S for S. We use the same matching algorithm as before, and for each candidate path $p = p_1, \dots, p_k$ through G reported, we perform an additional verification step. Specifically, we see if the string '$p_1 \dots p_k$' exists in S as a substring by performing a backward search query in S. Clearly, if the path description of p is a substring in S, then p is a sub-path of some valid path in $\mathcal{P}$.

Theorem 28. A set of valid paths $\mathcal{P}$ can be represented in $(1 + o(1))h \log |V|$ bits, where h is the total number of nodes in all paths in $\mathcal{P}$, such that a candidate path p, crossing k nodes, can be verified in $O(k \log |V|)$ time.

We note that the space required to store all valid paths (by canonical ids) of a transcriptome is still dominated by and nearly negligible compared to the space to index the node text.

10.6.2 Topology constraints

The worst case complexity of the proposed pattern matching algorithm can be improved with a small modification when the graph topology of the hypertext is know to be sparse; more specifically, if $\Delta(G)$, the maximum degree of any vertex in V, is $O(1)$. This is the case for our motivating problem of modeling transcriptomes.

For hypertexts with this property, we can adapt the algorithm when considering patterns crossing more than one edge. When a candidate is under consideration, sub-path extension and sub-path initiation events can be simplified. Since there are only a constant number of neighbours of any candidate, we can use Q and matching statistics stored during preprocessing to determine if any neighbour of the candidate satisfies a sub-path initiation event. (The same can be done for prefix verification.) This simplifies the $\gamma \frac{\log |V|}{\log \log |V|}$ term of the algorithm to γ.

To improve the time complexity for validating a sub-path extension event, we change how we store matching sub-paths in the working space W. Specifically, instead of storing a list of heads of sub-paths, for each suffix of P, we keep the matches in a balanced tree, such as a red-black tree [5], to support querying for a particular `lex_id` in $O(\log \gamma)$ time, as there are at most γ entries in any bucket. Overall, it will take $O(\gamma \log \gamma)$ time to insert all γ entries and $O(\gamma \log \gamma)$ time to check for extension events for all candidates. This simplifies the γ^2 term to $\gamma \log \gamma$, yielding the following result.

Theorem 29. For a hypertext $G = (V, E)$, with node text having total combined length n, if $\Delta(G)$, the maximum degree of any vertex in V, is $O(1)$, then the pattern matching algorithm can be adapted as described above to improve the counting time complexity to $O(m \log \sigma + m \frac{\log |V|}{\log \log |V|} + \gamma \log \gamma)$, and the working space to $O(m \log n + \gamma \log \gamma + \gamma \log |V|)$ bits, where m is the length of the pattern and γ is the number of occurrences of a node as a substring of the pattern.

10.6.3 Text constraints

A set of nodes in a hypertext form a *prefix-free code* if no node is the prefix of another. They are said to form a *quasi-prefix-free code* if none is a prefix of more than $O(1)$ other nodes.

Lemma 51. If the nodes of a hypertext $G = (V, E)$ form a *quasi-prefix-free code* then for a pattern P of length m, the number of candidate matches of any suffix of P is $O(1)$ and therefore $O(m)$ overall.

Proof. Suppose the nodes do form a quasi-prefix-free code but $\gamma = \omega(m)$. Then, by the pigeon hole principle, there must be at least one suffix, $P[i \dots m]$, that has $t = \omega(1)$ candidates as a prefix. Contradiction. □

With this restriction, each suffix of P contains $O(1)$ candidates as a prefix, and therefore, the working space W will track $O(1)$ head nodes for each position

of P. Therefore, each potential sub-path extension event only needs to check $O(1)$ candidates in $O(1)$ time using Q to perform adjacency queries. Since the number of overall candidates $\gamma = O(m)$, we have the following result.

Theorem 30. For a hypertext $G = (V, E)$, with node text having total combined length n, if the text in the nodes of V form a quasi-prefix-free code, then the pattern matching algorithm has a counting time complexity of $O(m \log \sigma + m \frac{\log |V|}{\log \log |V|})$ and working space $O(m \log n)$ bits.

Chapter 11

Conclusion

We have proposed a number of succinct and compressed text indexes motivated by biological sequence alignment problems. In Chapter 8 we proposed a new compressed full-text dictionary. The dictionary is competitive in terms of space with existing dictionary indexes. While its query time for identifying text segments contained within a query pattern is slower than current state-of-the-art dictionary indexes it was designed to have the additional functionality of identifying text segments which fully contain a query pattern. We demonstrated the utility of such an index in two applications: indexing text with wildcards and indexing hypertext.

We have presented a new compressed index for texts containing wildcard characters motivated by the problem of indexing genomes and accounting for known single nucleotide polymorphisms. We proposed a new query algorithm that can have a substantially reduced working space for short query patterns when compared to existing solutions. Our compressed index matches the query time of existing solutions while reducing the required space to $2nH_k(T) + o(n \log \sigma) + O(n) + O(d \log n)$ bits. By exploiting a technique used in bidirectional search, we were able to completely eliminate a portion of our index, without any additional modifications, that reduces space complexity of the compressed index to $nH_k(T) + o(n \log \sigma) + 2n + O(d \log n)$ bits, but increases the worst case query time. We also showed how ideas from a recent index proposed independently and in parallel by Hon et al. [49] can be combined with the ideas presented here to improve the overall approach to indexing text with wildcards.

We proposed a succinct index to model hypertext. The index can model any hypertext and places no restriction on the graph topology. We proposed a new pattern matching algorithm, capable of aligning a pattern to any path in the hypertext. We showed how the index can occupy space proportional to the compressed text and graph topology representation. We also studied a number of interesting restrictions of hypertext, including when the graph consists of nodes with constant degree, when only certain paths within the hypertext are considered valid, and when few nodes share common prefixes. In these cases, which are motivated by the problem of aligning patterns to transcriptomes, we gave variants of the algorithm with improved query time complexity that is dependent only on the pattern length and logarithmically in the size of the hypertext. Previous algorithms for exact matching were at least linear in the size of the hypertext. We demonstrated the correspondence between indexing and matching within hypertext to the problem of indexing and matching text containing wildcards. Improved solutions to one problem may immediately lead

to an improvement for the other.

Similar to our wildcard application, the query algorithm for the hypertext index makes use of a succinct 2D orthogonal range structure to model the graph topology and perform orthogonal range queries. To our knowledge, no succinct structure for this purpose also supports point (non-) existence queries in constant time. Such a structure would eliminate the need to store the graph topology twice without increasing the hypertext query time.

The query algorithm for both the wildcard and the hypertext indexes proposed here suffer from a common deficiency: they are not output sensitive. In particular both rely on a parameter γ. For the wildcard (hypertext) problem, this corresponds to the number of text segments (nodes) that are a substring of the query pattern. In practical scenarios, this quantity can be assumed to be proportional to the length of the query pattern; however, in theory it can be as large as $O(nm)$ for a pattern of length m and index text having a combined length of n. This is by design as the query algorithm is driven by dictionary matches. This motivates a number of open questions. Can a query algorithm be designed that is dependent only logarithmically in n, for an index of comparable size to this work? Can the index itself be redesigned to achieve an output sensitive query algorithm?

Finally, while our indexes were motivated by problems in biological sequence alignment, there are factors which must be considered to make them practical for this application. As illustrated by the short read alignment problem, it is generally the case in computational biology that one needs to solve the approximate string matching (ASM) problem, finding all *occ* occurrences of a pattern P of length m in a text T of length n, including positions containing up to k errors. The online version[35] of this problem is easily solved using the dynamic programming formulation of Wagner and Fischer [137]. However, as with the applications considered here, the text is known *a priori*. Offline ASM algorithms attempt to leverage this fact to improve upon the $O(kn)$ runtime required to solve the online version [67].

Early solutions to the offline ASM problem had worst case running times polynomial in n [58]. It was Ukkonen who proposed the first algorithm with a worst case runtime independent of n; the algorithm performs dynamic programming over a suffix tree structure. Other solutions that used an index originally intended for exact matching used neighbourhood generation. In neighbourhood generation, all strings within edit distance k of P are generated and subsequently checked for exact matches against T; thus the runtime is exponential in m and k, but not n.

Recent attention has been placed on providing ASM capabilities on succinct indexes. This has been achieved by Huynh et al. [54] using compressed suffix arrays, with a runtime of $O(|A|^k m^k \max(k, \lg n) + occ)$ using $O(n \lg n)$ bits of space, where A is the alphabet. More recently, Russo et al. [109] proposed a scheme based on hierarchical verification[36] and bidirectional string alignment

[35] An online string matching algorithm does not pre-process the pattern or text prior to search.

[36] Hierarchical verification is scheme where matches containing $i-1$ errors are first checked

[68] which can be used with nearly all existing succinct indexes. This approach uses significantly less space than previous ASM index approaches and does not require *a priori* knowledge of the maximum number of mismatches k. As our indexes use standard compressed suffix arrays as subsidiary structures, query algorithms such as this are readily adaptable for the applications considered here. It is unclear however that approximate query algorithms on compressed suffix arrays without an exponential term are possible. Should a breakthrough in approximate string matching occur, a reexamination of the indexing problems considered here would be warranted to incorporate new techniques.

before matches containing i errors, for $1 < i \leq m$.

Bibliography

[1] T. Akutsu. A Linear Time Pattern Matching Algorithm Between a String and a Tree. In *Symposium on Combinatorial Pattern Matching*, pages 1–10, 1993.

[2] A. Amir, M. Lewenstein, and N. Lewenstein. Pattern matching in hypertext. *Journal of Algorithms*, 35(1):82–99, 2000.

[3] M Andronescu, V Bereg, H H Hoos, and A Condon. RNA STRAND: the rna secondary structure and statistical analysis database. *BMC Bioinformatics*, 9:340–340, 2008.

[4] T Baumstark, A R Schroder, and D Riesner. Viroid processing switch from cleavage to ligation is driven by a change from a tetraloop to a loop e configuration. *The EMBO Journal*, 16:599–610, 1997.

[5] Rudolf Bayer. Symmetric binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica*, 1:290–306, 1972.

[6] D. Belazzougui. Succinct dictionary matching with no slowdown. In *Symposium on Combinatorial Pattern Matching*, pages 88–100, 2010.

[7] D. Belazzougui and G. Navarro. Alphabet-independent compressed text indexing. In *Proc. 19th Annual European Symposium on Algorithms (ESA)*, LNCS 6942, pages 748–759, 2011.

[8] C.H. Bennett. Logical reversibility of computation. *IBM journal of Research and Development*, 17(6):525–532, 1973.

[9] C.H. Bennett. Time/space trade-offs for reversible computation. *SIAM Journal on Computing*, 18(4):766–776, 1989.

[10] J.L. Bentley, D.D. Sleator, R.E. Tarjan, and V.K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4), 1986.

[11] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. *Algorithms and Data Structures*, pages 98–109, 2009.

[12] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Lett. Program. Lang. Syst.*, 2:59–69, 1993. ISSN 1057-4514.

[13] M.S. Brose, P. Volpe, M. Feldman, M. Kumar, I. Rishi, R. Gerrero, E. Einhorn, M. Herlyn, J. Minna, A. Nicholson, et al. BRAF and RAS mutations in human lung cancer and melanoma. *Cancer research*, 62(23):6997, 2002.

[14] Weiping Cai, Anne E. Condon, Robert M. Corn, Elton Glaser, Zhengdong Fei, Tony Frutos, Zhen Guo, Max G. Lagally, Qinghua Liu, Lloyd M. Smith, and Andrew Thiel. The power of surface-based DNA computation (extended abstract). In *Proceedings of the first annual international conference on Computational molecular biology*, RECOMB '97, pages 67–74, 1997.

[15] L. Cardelli. Strand algebras for DNA computing. *Natural Computing*, 10 (1):407–428, 2011.

[16] E. Cardoza, R. Lipton, and A.R. Meyer. Exponential space complete problems for Petri nets and commutative semigroups. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 50–54. ACM, 1976.

[17] T.M. Chan, K.G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th annual ACM symposium on Computational geometry*, pages 1–10. ACM, 2011.

[18] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, January 1981.

[19] Harish Chandran, Nikhil Gopalkrishnan, Andrew Phillips, and John Reif. Localized hybridization circuits. In *DNA Computing and Molecular Programming*, pages 64–83. Springer, 2011.

[20] Shi-Jie Chen and Ken A. Dill. RNA folding energy landscapes. *Proceedings of the National Academy of Science*, 97(2):646–651, January 2000.

[21] E. Chiniforooshan, D. Doty, L. Kari, and S. Seki. Scalable, time-responsive, digital, energy-efficient molecular circuits using DNA strand displacement. In *Proceedings of the 16th international conference on DNA computing and molecular programming*, pages 25–36, 2010.

[22] Y.L. Choi, K. Takeuchi, M. Soda, K. Inamura, Y. Togashi, S. Hatano, M. Enomoto, T. Hamada, H. Haruta, H. Watanabe, et al. Identification of novel isoforms of the EML4-ALK transforming gene in non-small cell lung cancer. *Cancer research*, 68(13):4971, 2008.

[23] Francisco Claude and Gonzalo Navarro. Self-indexed text compression using straight-line programs. In *Mathematical Foundations of Computer Science*, pages 235–246, 2009.

[24] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *ACM Symposium on Theory of Computing*, pages 91–100, 2004.

[25] A. Condon, A. Hu, J. Maňuch, and C. Thachuk. Less haste, less waste: on recycling and its limits in strand displacement systems. In *Proceedings of the 17th Annual International Conference on DNA Computing and Molecular Programming*, pages 84–99, 2011.

[26] A. Condon, A.J. Hu, J. Maňuch, and C. Thachuk. Less haste, less waste: on recycling and its limits in strand displacement systems. *Journal of the Royal Society: Interface Focus*, 2(4):512–521, 2012.

[27] Anne Condon, Bonnie Kirkpatrick, and Jan Maňuch. Reachability bounds for chemical reaction networks and strand displacement systems. In *Proceedings of the 18th Annual International Conference on DNA computing and Molecular Programming*, 2012.

[28] S. Cook. Short propositional formulas represent nondeterministic computations. *Information Processing Letters*, 26(5):269–270, 1988.

[29] Ivan Dotu, William A. Lorenz, Pascal Van Hentenryck, and Peter Clote. Computing folding pathways between RNA secondary structures. *Nucl. Acids Res.*, page gkp1054, 2009.

[30] D. Doty, L. Kari, and B. Masson. Negative interactions in irreversible self-assembly. In *Proceedings of the Sixteenth Annual Conference on DNA Computing and Molecular Programming, Springer-Verlag Lecture Notes in Computer Science 6518*, pages 37–48, 2010.

[31] Arash Farzan and J. Munro. Succinct representations of arbitrary graphs. In *16th Annual European Symposium on Algorithms*, pages 393–404, 2008.

[32] W. Feller. *An Introduction to Probability Theory and Its Applications, Vol. 1.* Wiley, 1971.

[33] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Symposium on Foundations of Computer Science*, pages 390–398, 2002.

[34] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):20, 2007.

[35] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[36] C. Flamm, W. Fontana, I. L. Hofacker, and P. Schuster. RNA folding at elementary step resolution. *RNA*, pages 325–338, 2000.

[37] C Flamm, I L Hofacker, S Maurer-Stroh, P F Stadler, and M Zehl. Design of multi-stable RNA molecules. *RNA*, 7:254–265, 2001.

[38] C. Flamm, I. L. Hofacker, P. F. Stadler, and M. T. Wolfinger. Barrier trees of degenerate landscapes. *Zeitschrift für Physikalische Chemie*, 216: 155–174, 2002.

[39] K.A. Frazer, D.G. Ballinger, D.R. Cox, D.A. Hinds, L.L. Stuve, R.A. Gibbs, et al. A second generation human haplotype map of over 3.1 million SNPs. *Nature*, 449(7164):851–861, 2007.

[40] P.A. Futreal, L. Coin, M. Marshall, T. Down, T. Hubbard, R. Wooster, N. Rahman, and M.R. Stratton. A census of human cancer genes. *Nature reviews. Cancer*, 4(3):177, 2004.

[41] Michael R Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[42] M Geis, C Flamm, M T Wolfinger, A Tanzer, I L Hofacker, M Middendorf, C Mandl, P F Stadler, and C Thurner. Folding kinetics of large RNAs. *Journal of Molecular Biology*, 379:160–173, 2008.

[43] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

[44] Ronald Graham, Donald Knuth, and Oren Patashnik. *Concrete Mathematics: a foundation for computer science*. Addison-Wesley, Reading, MA, USA, 1989.

[45] R. Grossi and J.S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *ACM Symposium on Theory of Computing*, pages 397–406, 2000.

[46] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 841–850, 2003.

[47] N.J.A. Harvey. Algebraic Algorithms for Matching and Matroid Problems. *SIAM Journal on Computing*, 39:679, 2009.

[48] M. Hollstein, D. Sidransky, B. Vogelstein, and CC Harris. p53 mutations in human cancers. *Science*, 253(5015):49, 1991.

[49] Wing-Kai Hon, Tsung-Han Ku, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Compressed text indexing with wildcards. In *Proceedings of the 18th international conference on String processing and information retrieval*, SPIRE'11, pages 267–277, 2011.

[50] W.K. Hon, T.H. Ku, R. Shah, S. Thankachan, and J. Vitter. Faster Compressed Dictionary Matching. In *Symposium on String Processing and Information Retrieval*, pages 191–200, 2010.

[51] G. Hongzhou, J. Chao, S.-J. Xiao, and N.C. Seeman. A proximity-based programmable DNA nanoscale assembly line. *Nature*, 465:202–205, 2010.

[52] J E Hopcroft and R M Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.

[53] D.S. Horner, G. Pavesi, T. Castrignano, P.D.O. De Meo, S. Liuni, M. Sammeth, E. Picardi, and G. Pesole. Bioinformatics approaches for genomics and post genomics applications of next-generation sequencing. *Briefings in Bioinformatics*, 2009.

[54] T.N.D. Huynh, W.K. Hon, T.W. Lam, and W.K. Sung. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 352(1-3):240–249, 2006.

[55] Tucker B J and R R Breaker. Riboswitches as versatile gene control elements. *Current Opinion in Structural Biology*, 15(3):342, 2005.

[56] Guy Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1989.

[57] S. Jay and H. Ji. Next-generation DNA sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[58] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *16th international symposium on Mathematical foundations of computer science*, page 240, 1991.

[59] Juha Kärkkäinen. *Repetition-Based Text Indexes*. PhD thesis, University of Helsinki, 1999.

[60] A. Kharam, H. Jiang, M. Riedel, and K. Parhi. Binary counting with chemical reactions. In *Proceedings of the 2011 Pacific Symposium on Biocomputing*, pages 302–313. World Scientific Publishing, 2011.

[61] Dénes Kőnig. Gráfok és alkalmazásuk a determinánsok és a halmazok elméletére. *Matematikai és Természettudomnyi Értesítő*, 34:104–119, 1916.

[62] AlexE. Knight. Single enzyme studies: A historical perspective. In Gregory I. Mashanov and Christopher Batters, editors, *Single Molecule Enzymology*, volume 778 of *Methods in Molecular Biology*, pages 1–9. Humana Press, 2011.

[63] D.E. Knuth, J.H. Morris Jr, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323, 1977.

[64] M. Lakin and A. Phillips. Modelling, simulating and verifying Turing-powerful strand displacement systems. *DNA Computing and Molecular Programming*, pages 130–144, 2011.

[65] M.R. Lakin, D. Parker, L. Cardelli, M. Kwiatkowska, and A. Phillips. Design and analysis of DNA strand displacement devices using probabilistic model checking. *Journal of The Royal Society Interface*, 2012.

[66] Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Siu-Ming Yiu. Space efficient indexes for string matching with don't cares. In *Conference on Algorithms and Computation*, pages 846–857, 2007.

[67] G.M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *Journal of Algorithms*, 10(2):157–169, 1989.

[68] G.M. Landau, E.W. Myers, and J.P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27:557–582, 1998.

[69] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of research and development*, 5(3):183–191, 1961.

[70] K.J. Lange, P. McKenzie, and A. Tapp. Reversible space equals deterministic space. *Journal of Computer Systems Science*, 60(2):354–367, 2000.

[71] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[72] L.Cardelli. Two-domain DNA strand displacement. In *Proc. of Developments in Computational Models (DCM 2010)*, volume 26 of *Electronic Proceedings in Theoretical Computer Science*, pages 47–61, 2010.

[73] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754, 2009.

[74] Z.J. Lu, D.H. Turner, and D.H. Mathews. A set of nearest neighbor parameters for predicting the enthalpy change of RNA secondary structure formation. *Nucleic acids research*, 2006.

[75] K. Lund, A.T. Manzo, N. Dabby, N. Michelotti, A. Johnson-Buck, J. Nangreave, N. Taylor, R. Pei, M.N. Stojanovic, N.G. Walter, E. Winfree, and H. Yan. Molecular robots guided by prescriptive landscapes. *Nature*, 465: 206–210, 2010.

[76] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Conference on String processing and information retrieval*, pages 229–241, 2007.

[77] U. Manber and S. Wu. Approximate String Matching With Arbitrary Costs for Text and Hypertext. In *IAPR International Workshop on Structural and Syntactic Pattern Recognition*, pages 22–33, 1992.

[78] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, 1990.

[79] D.H. Mathews, J. Sabina, M. Zuker, D.H. Turner, et al. Expanded sequence dependence of thermodynamic parameters improves prediction of RNA secondary structure. *Journal of molecular biology*, 288(5):911–940, 1999.

[80] J Maňuch, C Thachuk, L Stacho, and A Condon. NP-completeness of the direct energy barrier problem without peudoknots. *Proc. of the 15th International Meeting on DNA Computing and Molecular Programming (DNA15)*, 2009.

[81] J. Maňuch, C. Thachuk, L. Stacho, and A. Condon. NP-completeness of the energy barrier problem without pseudoknots and temporary arcs. *Natural Computing*, 10(1):391–405, 2011.

[82] Ján Maňuch, Chris Thachuk, Ladislav Stacho, and Anne Condon. NP-completeness of the direct energy barrier problem without pseudoknots. In *Proc. of DNA15*, 2009.

[83] S. R. Morgan and P. G. Higgs. Barrier heights between ground states in a model of RNA secondary structure. *Journal of Physics A: Math General*, 31:3153–3170, 1998.

[84] A. Mortazavi, B.A. Williams, K. McCue, L. Schaeffer, and B. Wold. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nature methods*, 5(7):621–628, 2008.

[85] J.I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. on Computing*, 31(3):762–776, 2002.

[86] J.I. Munro, V. Raman, and S.S. Rao. Space Efficient Suffix Trees. In *Proceedings of the 18th Conference on Foundations of Software Technology and Theoretical Computer Science*, page 196, 1998.

[87] G. Navarro. Improved approximate pattern matching on hypertext. *Theoretical Computer Science*, 237(1-2):455–463, 2000.

[88] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.

[89] R Nussinov and A B Jacobson. Fast algorithm for predicting the secondary structure of single-stranded RNA. *Proceedings of the National Academy of Sciences of the United States of America*, 77(11):6309–6313, 1980.

[90] Enno Ohlebusch and Simon Gog. A compressed enhanced suffix array supporting fast string matching. In *Symposium on String Processing and Information Retrieval*, pages 51–62, 2009.

[91] Enno Ohlebusch, Simon Gog, and Adrian Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Symposium on String Processing and Information Retrieval*, pages 347–358, 2010.

[92] T. Omabegho, R. Sha, and N.C. Seeman. A bipedal DNA brownian motor with coordinated legs. *Science*, 324(5923):67–71, 2009.

[93] J.G. Paez, P.A. Janne, J.C. Lee, S. Tracy, H. Greulich, S. Gabriel, P. Herman, F.J. Kaye, N. Lindeman, T.J. Boggon, et al. EGFR mutations in lung cancer: correlation with clinical response to gefitinib therapy. *Science*, 304(5676):1497, 2004.

[94] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994.

[95] K. Park and D.K. Kim. String Matching in Hypertext. In *Symposium on Combinatorial pattern matching*, page 318, 1995.

[96] D. Parkhomchuk, T. Borodina, V. Amstislavskiy, M. Banaru, L. Hallen, S. Krobitsch, H. Lehrach, and A. Soldatov. Transcriptome analysis by strand-specific sequencing of complementary DNA. *Nucleic Acids Research*, 2009.

[97] L. Qian and E. Winfree. A simple DNA gate motif for synthesizing large-scale circuits. *J. R. Soc. Interface*, 2011.

[98] L. Qian, D. Soloveichik, and E. Winfree. Efficient Turing-universal computation with DNA polymers (extended abstract). In *Proceedings of the 16th Annual conference on DNA computing*, pages 123–140, 2010.

[99] L. Qian, E. Winfree, and J. Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475(7356):368–372, 2011.

[100] Lulu Qian and Erik Winfree. Scaling up digital circuit computation with dna strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

[101] R. Raman, V. Raman, and S.S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *ACM-SIAM Symposium on Discrete Algorithms*, pages 233–242, 2002.

[102] J.H. Reif, S. Sahu, and P. Yin. Complexity of graph self-assembly in accretive systems and self-destructible systems. In *Proceedings of the Eleventh Annual Conference on DNA-Based Computing, Springer-Verlag Lecture Notes in Computer Science 3892*, pages 257–275, 2006.

[103] J.I. Risinger, A.K. Hayes, A. Berchuck, and J.C. Barrett. PTEN/MMAC1 mutations in endometrial cancers. *Cancer Research*, 57(21):4736, 1997.

[104] A Roth and R R Breaker. The structural and functional diversity of metabolite-binding riboswitches. *Annual Rev. Biochem.*, 78:305–34, 2009.

[105] P.W.K. Rothemund. Folding DNA to create nanoscale shapes and patterns. *Nature*, 440(7082):297–302, 2006.

[106] P.W.K. Rothemund and E. Winfree. The program-size complexity of self-assembled squares. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 459–468, 2000.

[107] Boris Rotman. Measurement of activity of single molecules of β-D-galactosidase. *Proceedings of the National Academy of Sciences of the United States of America*, 47(12):1981, 1961.

[108] R. Russell, X. Zhuang, H.P. Babcock, I.S. Millett, S. Doniach, S. Chu, and D. Herschlag. Exploring the folding landscape of a structured RNA. *Proceedings of the National Academy of Science*, 99:155–160, 2002.

[109] L.M. Russo, G. Navarro, and A.L. Oliveira. Indexed Hierarchical Approximate String Matching. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, page 154, 2008.

[110] Kunihiko Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12 – 22, 2007.

[111] C. Savage. A survey of combinatorial Gray codes. *SIAM Review*, 39(4): 605–629, 1997.

[112] Joseph Schaeffer. The multistrand simulator: Stochastic simulation of the kinetics of multiple interacting DNA strands. Master's thesis, California Institute of Technology, 2012.

[113] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees. In *Symposium on Combinatorial Pattern Matching*, pages 40–50, 2010.

[114] L.J. Schulman and D. Zuckerman. Asymptotically good codes correcting insertions, deletions, and transpositions. *IEEE Transactions on Information Theory*, 45:2552–2557, 1999.

[115] Erik A. Schultes and David P. Bartel. One sequence, two ribozymes: Implications for the emergence of new ribozyme folds. *Science*, 289(5478): 448–452, July 2000. doi: 10.1126/science.289.5478.448.

[116] G. Seelig, D. Soloveichik, D.Y. Zhang, and E. Winfree. Enzyme-free nucleic acid logic circuits. *Science*, 314(5805):1585–1588, 2006.

[117] I. Shcherbakova, S. Mitra, A. Laederach, and M. Brenowitz. Energy barriers, pathways, and dynamics during folding of large, multidomain RNAs. *Current Opinion in Chemical Biology*, 12(6):655–666, 2008.

[118] J. Shendure and H. Ji. Next-generation DNA sequencing. *Nature Biotechnology*, 26(10):1135–1145, 2008.

[119] J.-S. Shin and N.A. Pierce. A synthetic DNA walker for molecular transport. *Journal of the American Chemical Society*, 126:10834–10835, 2004.

[120] Friedrich C. Simmel and Wendy U. Dittmer. DNA nanodevices. *Small*, 1:284–299, 2005.

[121] D. Soloveichik, M. Cook, E. Winfree, and J. Bruck. Computation with finite stochastic chemical reaction networks. *Natural Computing*, 7(4): 615–633, 2008.

[122] D. Soloveichik, G. Seelig, and E. Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Science*, 107(12):5393–5398, 2010.

[123] David Soloveichik. Personal communication, 2012.

[124] Alan Tam, Edward Wu, Tak-Wah Lam, and Siu-Ming Yiu. Succinct text indexing with wildcards. In *Symposium on String Processing and Information Retrieval*, pages 39–50, 2009.

[125] X. Tang, S. Thomas, L. Tapia, D. P. Giedroc, and N. M. Amato. Simulating RNA folding kinetics on approximated energy landscapes. *Journal of Molecular Biology*, 381:1055–1067, 2008.

[126] R Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[127] C. Thachuk. Succincter text indexing with wildcards. In *Symposium on Combinatorial Pattern Matching*, pages 27–40, 2011.

[128] C. Thachuk. Compressed Indexes for Text with Wildcards. *Theoretical Computer Science*, page (Accepted), 2012.

[129] C. Thachuk, J. Manuch, A. Rafiey, L.A. Mathieson, L. Stacho, and A. Condon. An algorithm for the energy barrier problem without pseudoknots and temporary arcs. In *Pacific Symposium of Biocomputing*, 2010.

[130] Chris Thachuk. A succinct index for hypertext. In *Proceedings of the 18th international conference on String processing and information retrieval*, SPIRE'11, pages 209–220, 2011.

[131] Chris Thachuk and Anne Condon. Space and energy efficient computation with dna strand displacement systems. In *Proceedings of the 18th Annual International Conference on DNA computing and Molecular Programming*, 2012.

[132] C. Trapnell, L. Pachter, and S.L. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105, 2009.

[133] D. K. Treiber and J. R. Williamson. Beyond kinetic traps in RNA folding. *Current Opinion in Structural Biology*, 11:309–314, 2001.

[134] DH Turner, N. Sugimoto, and SM Freier. RNA structure prediction. *Annual Review of Biophysics and Biophysical Chemistry*, 17(1):167–192, 1988.

[135] E. Ukkonen. Approximate string matching over suffix trees. In *Proceedings of the 4th annual symposium on combinatorial pattern matching*, volume 684, pages 228–242, 1993.

[136] S. Venkataraman, R.M. Dirks, P.W.K. Rothemund, E. Winfree, and N.A. Pierce. An autonomous polymerization motor powered by DNA hybridization. *Nature Nanotech*, 2(8):490–494, 2007.

[137] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[138] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: a revolutionary tool for transcriptomics. *Nature Reviews Genetics*, 10:57–63, 2009.

[139] P. Weiner. Linear pattern matching algorithms. In *14th Annunal Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[140] R. Williams. Non-linear time lower bound for (succinct) quantified boolean formulas. In *Electronic Colloquium on Computational Complexity (ECCC) TR08-076*, 2008.

[141] E. Winfree. *Algorithmic Self-Assembly of DNA*. PhD thesis, Caltech, 1998.

[142] Erik Winfree. Personal communication, 2012.

[143] M. T. Wolfinger. The energy landscape of RNA folding. Master's thesis, University Vienna, 2001.

[144] K. Wright, P. Wilson, S. Morland, I. Campbell, M. Walsh, T. Hurst, B. Ward, M. Cummings, and G. Chenevix-Trench. beta-catenin mutation and expression analysis in ovarian cancer: exon 3 mutations and nuclear translocation in 16% of endometrioid tumours. *International journal of cancer. Journal international du cancer*, 82(5):625, 1999.

[145] C Yanofsky. RNA-based regulation of genes of tryptophan synthesis and degradation, in bacteria. *RNA*, 13:1141–1154, 2007.

[146] P. Yin, H.M.T. Choi, C.R. Calvert, and N.A. Pierce. Programming biomolecular self-assembly pathways. *Nature*, 451:318–322, 2008.

[147] B. Yurke, A. J. Turberfield, A. J. Jr. Mills, F. C. Simmel, and J. L. Neumann. A DNA-fuelled molecular machine made of DNA. *Nature*, 406: 605–608, 2000.

[148] G. Zavattaro and L. Cardelli. Termination problems in chemical kinetics. In *Proceedings of the 19th International conference on Concurrency Theory*, pages 477–491, 2008.

[149] David Yu Zhang. *Dynamic DNA strand displacement circuits*. PhD thesis, California Institute of Technology, 2010.

[150] David Yu Zhang, Andrew J. Turberfield, Bernard Yurke, and Erik Winfree. Engineering Entropy-Driven Reactions and Networks Catalyzed by DNA. *Science*, 318(5853):1121–1125, 2007.

[151] D.Y. Zhang and G. Seelig. Dynamic DNA nanotechnology using strand displacement reactions. *Nature Chemistry*, 3:103–113, 2011.