Computer Modeling of Molecular Genetic Events using Content Sensitive Analysis

by

Curtis Stodgell

A THESIS SUBMITTED IN PARTIAL FULLFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF HONOURS IN COMPUTER SCIENCE

in

Irving K. Barber School of Arts and Sciences

THE UNIVERSITY OF BRITISH COLUMBIA OKANAGAN CAMPUS
March 2007

Abstract:

The problem of motif finding plays an important role in understanding the development, function and evolution of organisms. The Planted (*l, d*)-Motif Problem first introduced by Pevzner and Sze [13] is a variant model of motif finding that has been widely studied. Nevertheless, despite the many different algorithms constructed to solve this problem, it is far from being solved entirely [5]. We analyze a number of algorithms including: the Basic Voting Algorithm, the Winnower and the Closest String. One thing that has become ubiquitous among these algorithms is the tradeoff between efficiency and accuracy. We formulate the motif-finding problem as a constraint satisfaction problem and introduce a special form of constraint consistency. By using a fixed-parameter algorithm for the closest string problem [4], we develop a propagation algorithm that enforces the new constraint consistency with the same worst-case time complexity as that of the standard path consistency algorithm. Experiments on randomly generated sequences indicate that our approach is effective and efficient.

Table of Contents

1 Introduction:

Transcription factors are proteins that bind to DNA in order to regulate the expression of genes through the activation or inhibition of transcription mechanisms. Locating these sites is an integral part of understanding the regulatory process. Computational tools are an invaluable method used to locate these particular sites and this general problem has been termed motif finding.

Essentially the motif finding problem is the problem of finding common patterns among a set of sequences. A simple model for the problem is as follows: Given a sample of random sequences can we find a common unknown pattern (motif) that is hidden at random locations in the sequences? If the motif was not subject to mutations it would be relatively easy to locate using a basic brute force algorithm; however, in biology the sequences being analyzed are subject to mutations and therefore the motif cannot be assumed to be exact. As such, it makes sense to construct a model that will allow the patterns we are searching for to have an arbitrary number of mutations [13].

Buher and Tompa [8, 7] found limitations for this particular problem. They found that when the number of sequences, the length of sequences and the length of the pattern are fixed, if the number of mutations in the pattern is larger than a particular threshold then it is highly improbable that any algorithm would be able to find such a pattern due to an abundance of random patterns [8, 7]. However, there have been many algorithms suggested for instances of this problem that do not exceed this threshold [8].

In this thesis we analyze a number of different algorithms that attempt to solve the problem in several different ways. These include A voting algorithm by Leung and Chin [8] that locates common motifs by enumerating all *neighbors* of each l length substrings in a given set of sequences, as well as a combinatorial approach by Pevzner and Sze [13] that essentially reduces the motif finding problem to locating cliques in multi-partite graphs. Closely related to the planted-motif problem are the closest string problem and the closest substring problem. Both of these problems are NP-hard, but the closest string problem can be solved in linear time assuming that the number of mutations is a fixed constant [9].

In the Winnower Pevzner and Sze designed an algorithm that prunes spurious similarities, which enables us to easily locate the signal we are searching for. They did so with an algorithm that has a time complexity of $O(N^{k+1})$ and found that the algorithm was accurate for k=3. Our contribution includes a variation of the Winnower combined with the closest string. We have designed an algorithm that essentially runs as the Winnower for k=2 along with the closest string algorithm to construct an algorithm with a time complexity of $O(N^3 + d^d)$ where d is the number of mutations and constant.

This thesis is organized as follows: In chapter 2, we will discuss numerous algorithms that approach The Planted (*l, d*)-Motif Problem in various ways, each algorithm achieving a different level of both accuracy and efficiency. In chapter 3, we present our approach to the problem, including the CSP formulation, the propagation algorithm for the stronger notion of path consistency and its implementation. In chapter 4 we present our results and allegorize the pruning power and efficiency of our approach. In chapter 5, we will conclude our results and analyses, as well as propose future challenges.

2 Planted Motif Problem

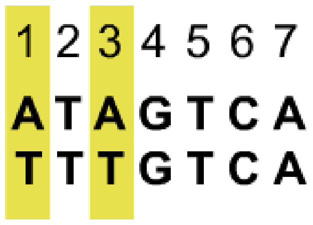
In this chapter we introduce formally the Planted (*l, d*)-Motif Problem [13] as well as discuss a variety of existing algorithms designed to solve this problem.

2.1 Definitions and Notation

In order to understand how the motif finding problem can be solved we will give a formal definition based on a variant model of the problem presented by Pevzner and Sze [13], as well as define notation we will use throughout this thesis.

The methods we will be using to solve this particular problem will usually be executed partially by comparing the Hamming distance between two strings.

Definition 1 (Hamming Distance): Hamming distance is a simple concept in an area of mathematics termed information theory. It is defined simply as the number of indices that two strings differ. For example given two strings "ATAGTCA" and "TTTGTCA" we can see that they differ at indices 1 & 3; therefore, the hamming distance between these two strings is 2



In this paper the notation $D_H(s, s')$ will be used to denote the hamming distance between a string *s* and *s'*.

Definition 2 (Planted (l, d)-Motif Problem): A motif is a single or repeated pattern. The problem of motif finding deals with locating common patterns among a set of sequences. What makes the problem difficult is the fact that this pattern is not exact and is allowed to differ by a specific number (referred to as mutations). Formally we can describe the problem as follows:

Assume there exists a motif *m* of length *l*; given *t* number sequences each of length *n*, the problem is to find a planted motif *m'* of length *l* in each of the *t* sequences such that *m'* differs in at most *d* positions from *m*.

A more general problem is the closest substring problem which has been shown to be NP-hard [9]. The difference between the planted motif problem and the closest string problem is that for the planted motif problem, it is guaranteed that there exists at least one common substring that occurs in each of the sequences. Whereas for the closest string problem, it is not guaranteed that there will be such a common substring among each of the sequences. The parameters *l* and *d* in the Planted (*l, d*)-Motif Problem have a significant impact on the practical difficulty of the problem.

Some particularly difficult Planted (*l, d*)-Motif problem instances proven to be probabilistically challenging are (9,2), (11, 3) and (15, 4) [1]. These problems are considered challenging because given any two instances of a mutated (*l, d*)-signal there is a high probability that non-signals (spurious similarities) will be randomly generated such that they very closely resemble the proper signal. Many techniques have been designed to solve these challenging problems which will be looked at below.

2.2 Winnowing Approach: Locating Signals via Winnowing.

Pavel A. Pevzner and Sing-Hoi Sze [4] take a combinatorial approach to the motif finding problem by reducing it to a clique finding problem. Pevzner et al. recognize that in each sequence there are spurious signals that occur at random that tend to hide the real signal. Most algorithms tend to focus directly on the process of locating the real signal; however, the winnower takes an opposite approach. It focuses on the spurious signals (rather than the signal itself) and removes these spurious similarities incrementally until the signal we are looking for is no longer difficult to find. This elimination process is, however, non-trivial and time consuming.

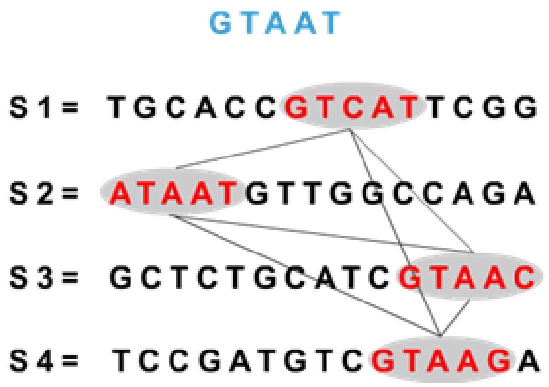
It is important to note that the Winnower algorithm only considers signals with Hamming distance as criterion for an edge and not insertions or deletions (signals will all be the same length). This is done for simplicity; however the algorithm could easily be modified to account for these constraints [8].

The Winnower is set up to reduce the Planted (l,d)-Motif problem to finding large cliques in a multi-partite graph. Given a set of t sequences S = $\{s_1, s_2, \ldots, s_t\}$ each of length n, a parameter l (length of the signal) and a number d (number of mismatches), construct a graph G=(V, E) in the following way:

Vertices will correspond to substrings of each sequence, that is, for every sequence s_i of the set S = $\{s_1, s_2, \ldots, s_t\}$ and for each index j in sequence s_i we construct a vertex $v \in V$ that will represent the substring s_{ij} of length l from index j to (n-1) + l; thus there is a total of $t((n\text{-}1) + l)$ vertices.

There is an edge between v_i and v_j if and only if $D_H(vi, vj) \leq 2d$ and v_i and v_j correspond to substrings from separate sequences.

A clique in a graph G = (V,E) is a subset C such that any two vertices in C are connected by an edge. Given any graph F = (V, E), constructed as described by the Winnower, the question if F contains a clique of at least size k, can be asked. Once such a clique has been found, it will also be the case that this clique contains the original signal being sought after. An example of 4 sequences (partites) forming an *extendable* clique can be seen in the figure below:

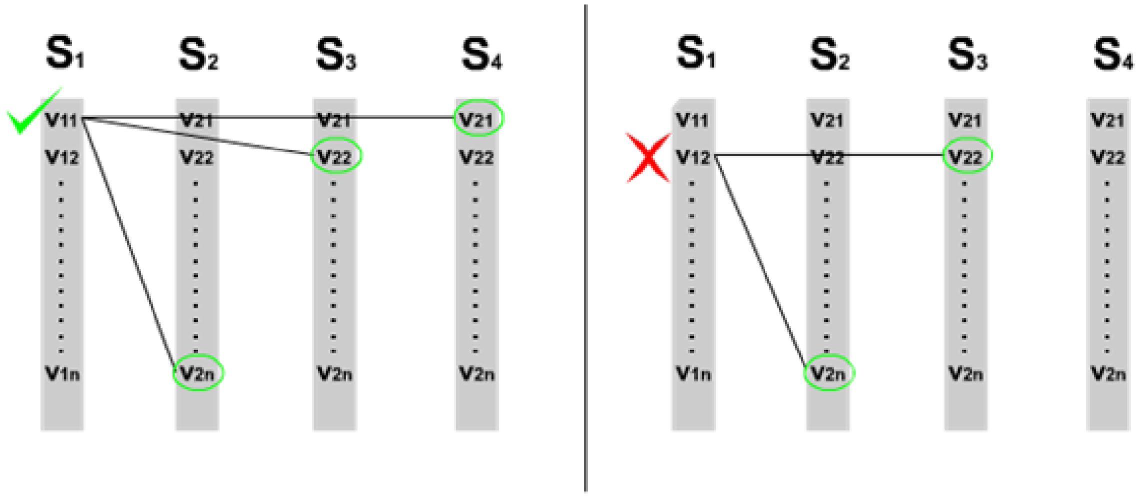


The clique finding problem is a graph-theoretical NP-complete problem [6]. There has been recent interest in studying the problem of finding hidden cliques in an otherwise randomly generated graph [6]. The planted motif-finding problem is a special case of the hidden clique problem in that the underlying graph is a multi-partite graph.

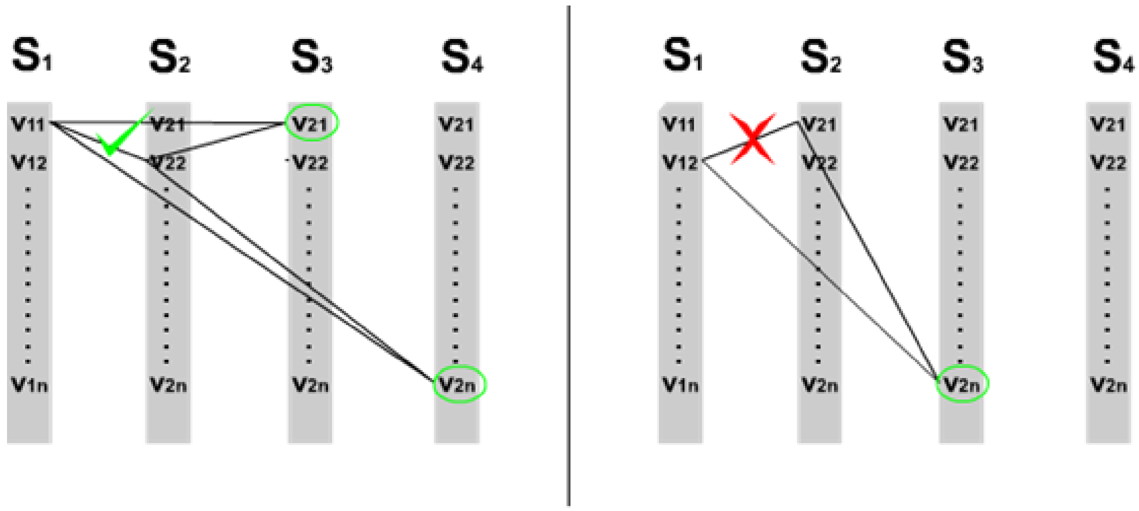
The Winnower algorithm uses the principle of exploring these particular graphs and will remove edges that are proven to not be part of the clique we are searching for. To describe how the winnower removes edges Pevzner and Sze use the notation of an *extendable clique.* "We say that a vertex u is a *neighbor* of a clique C = $\{v_1, \ldots, v_k\}$ if $\{v_1, \ldots, v_k, u\}$ is a clique in the graph. We define a clique to be *extendable* if it has at least one neighbor in every part of the multipartite graph G. We define an edge to be *spurious* if it does not belong to any extendable clique of size *k*. One way to impose increasingly strict conditions as *k* increases is to ensure that all spurious edges are deleted after winnowing and only extendable cliques remain in the final graph." [3, pg. 3]. The essential idea of the winnower is to allow only extendable cliques of size *k* to remain as *k* increases; in doing so *spurious* edges are effectively removed.

This is also the approach that was taken by Vingron & Agros[13, 14] and Vingron & Pevzner [13, 15] for k = 2. However it was observed by Pevzner & Sze that when $k \leq 2$, filtering edges that are not part of an extendable clique is too weak for pruning spurious edges [13]. Their observations indicate that only when k = 3 does the method become effective at removing enough spurious edges in such a way that the clique we're searching becomes evident.

The *winnower* removes edges in the following way: Given a multipartite graph G = (*V*,*E*) and a number k = 1, the algorithm checks every vertex in G and applies the following test. A vertex *u* will pass the test if there exists at least one neighbor v_i in each partite set of G (ei: (u, v_i) is an edge in each s_i), if vertex *u* fails this test, it will be deleted. In the illustration below, for a multipartite graph with four partite sets, vertex v_{11} will not be deleted (v_{11} has as edge in S_2, S_3, S_4); however, vertex v_{12} will be deleted (v_{12} has an edge in S_2 and S_3 but no edge in S_4).



For k = 2 the algorithm will now check each edge (u, w) in the graph G. The edge (u, w) will pass the test if there is a vertex *vi* for each partite s_i such that {u, v_i, w} is a triangle. If there is no such vertex v_i, edge (u, w) will be deleted. In the illustration below, edge (v_{11}, v_{22}) will not be deleted; however, edge (v_{12}, v_{21}) will be deleted.



So, as k increases the conditions for edge removal become much stronger. The algorithm continues in this fashion for k = 3; the algorithm selects each triangle (u, w, x) to apply the similar test for the other values of k. This triangle will pass the test if there exists at least one vertex v_i in each partite set s_i if $\{u, v_i, w, x\}$ is a clique of size 4. Otherwise the edges (u, w), (w, x) & (u, x) will be removed from the graph.

Winnower is an iterative algorithm that removes inconsistent edges from a graph until locating extendable cliques of size k become a trivial problem. The construction of the graph for Winnower takes O(N^2) time, where N is the length of all sequences in the set S. The pruning power of Winnower increases as k increases, but so does the running time since the running time complexity of the Winnower is O(N^{k+1}). Pevzner and Sze point out that for many instances of the Planted-(l, d) Motif Problem k = 3 is sufficient for finding extendable cliques which are meaningful. However the Winnower also requires a large amount of time and memory; as a result it becomes very slow for large samples [13].

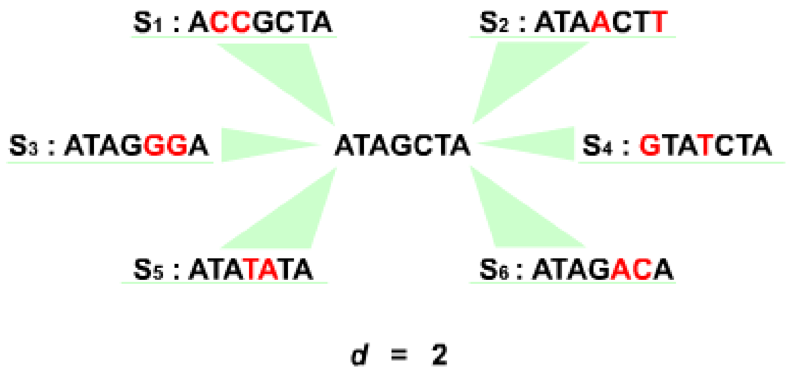
2.3 A Fixed-Parameter Algorithm for the Closest String Problem.

The Closest String problem is an important problem in many areas such as computational biology, coding theory and consensus word analysis [4][9]. The Closest String problem can be given the following general definition:

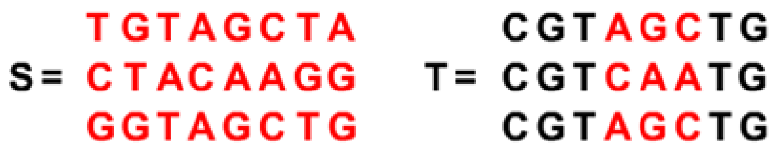
Definition 3 (Closest String Problem): Given a set S = {$s_1, s_2, \ldots, s_k$} of t strings of length n, and a number d the problem is to find a *center string* s' such that $D_H(s', s_i) \leq d$ for all $s_i \in$ S.

Closest string is an NP-complete problem; however, when we are dealing with applications of computational biology d is quite small. Jens Gram & Rolf Niedermeirer [4] present a fixed-parameter algorithm with an exponential growth complexity of $O(d^d)$. This is particularly important for the motif finding problem considering the value d is constant and quite small (range of 1 ~ 7).

Definition 4 (Closest string notation): For a given string s of length l, s[p] will denote the character at position p in s, where $1 \leq p \leq l$. Given a set of strings S = {$s_1, s_2, \ldots, s_k$}, where each string in the set is of length l, we say a string s is an *optimal closest string* for S if and only if there is no string s' such that $D_H(s', s_i) < D_H(s, s_i)$ for all $s_i \in$ S. The example below shows a *central string* s with $D_H(s, s_i) = 2$ for all $s_i \in \{s_1, s_2, s_3, s_4, s_5, s_6\}$.



Given a set of k strings of length l, we can create a k x l character matrix from this set of strings. The term *columns* of a *Closest String* matrix are in reference to this particular matrix. We say a particular *column* of this matrix is a *dirty column* if there are two or more differing characters in the *column.* For the example below the S matrix has 8 dirty columns while the T matrix has only 3 dirty columns.



Bounded search tree is one of the basic techniques used in designing fixed-parameter algorithms for a variety of problems. The algorithm for Closest String in Appendix A is a recursive procedure presented by Gramm & Niedermeirer that is proven to find a central string *s* if such a string exists for a given set of strings [4].

Initially, before calling the procedure, Gramm & Niedermier point out that if there are more than *kd* dirty columns, of the Closest String matrix S, then the instance is rejected (no center string can be found for given value of k & d). Otherwise the method is called with any arbitrary string $s_i \in$ S as the candidate string *s* and an integer value *d* as parameters. If there is another string $s_j \in$ S (where $s_i \neq s_j$), where $D_H(s, s_j) = h$ and $h > d$, then a position is selected in the candidate string and changed to match that of string s_j at the same position such that $D_H(s, s_j)$ = (*h*-1). The algorithm continues in this fashion until either *s* moves to far away from s_i or we find a solution to the problem (a center string *s'* is found). Gramm & Niedermier point out that when careful sub cases of this recursion are selected the upper limit on the size of this search tree is $O(d^d)$, where d is constant.

2.4 Basic Voting Algorithm

This section will describe the Basic Voting Algorithm (BSV) created by Leung and Chin [8] which is a variant of an algorithm first proposed by Waremann et al. [8, 16]. We will describe the BVR using the same notation as defined by the Planted (*l*,*d*)-Motif Problem.

Given a set of sequences S and two numbers *l* and *d*, the BVA will iterate through each sequence $s_i \in$ S and generate *l*-length substrings from each position in s_i from index 0 to index $(n-1) - l$. For each *l*-length substring the algorithm will generate all *d* variants and cast a vote on each variant generated. Any variant generated will get one and only one vote, per sequence, regardless of the fact that an identical variant is generated at a later step. When casting votes on variants in this way, it will be the case that motif *m* will have *t* number of votes.

The BVR trades a lower time complexity $O(nt(3l)^d)$ for an increased space complexity $O(n(3l)^d + nt)$, in comparison to a brute force algorithm which has complexities $O(nt4^l)$ and

O(*nt*) respectively [8]. Leung et al. show the BVR to be efficient for solving many Planted (*l, d*) Motif Problem instances [8].

3 Overview of Content Sensitive Consistency (CSC)

The problem of motif finding can be formulated as a constraint satisfaction problem (CSP). Formulating in this way allows us to detect and remove spurious similarities via the use of constraint propagation techniques that enforces local consistency. This process works similar to the methods used in the winnower approach with equivalent time complexities.

A CSP is a combinatorial search problem – well studied in the area of Artificial Intelligence. Essentially a CSP can be defined as: given a set of variables one must assign each variable a value that will satisfy a predefined number of constraints. Often a CSP requires a heuristic and combinatorial search method to be solved in reasonable time.

Definition 6 (CSP): A CSP consists of: A set of variables X = $\{X_1, X_2, \dots, X_n\}$ and a set of constraints C = $\{C_1, C_2, \dots, C_m\}$.

Each variable X_i has a set of i possible domains D = $\{D_1, D_2, \dots, D_i\}$ where for each value *j*, $D_j \in$ D defines a possible value. Each *constraint* is defined over a subset of variables and specifies the allowable combinations of values for that subset. Each *state* of this problem is defined by the assignment of values to a partial or a complete set of variables.

We say an assignment is *consistent* when a variable takes on a value that does not violate any constraint $C_i \in$ C for all of i. An assignment is *complete* when all variables have a value assigned (not necessarily satisfying all constraints). A *solution* to a CSP occurs when an assignment is both *consistent* and *complete.*

A general way to solve a CSP is to incrementally make assignments to variables in a *consistent* way until a *complete* assignment is achieved. There are many methods available to perform this operation; however, because the CSP is NP-hard, most algorithms have a time complexity that is exponential. There are however, many algorithms that can achieve lower running times when based on particular problem specific features. The CSP can be used to formulate many different existing problems. We show how to formulate the clique finding problem as a CSP.

Definition 7 (Motif finding as a binary CSP): Given a set of sequences S for an instance of the motif finding problem, we can define an instance of a CSP whose variables X correspond to S.

Each variable $x_i \in$ X will have a set of possible domains D = $\{D_1, D_2, \ldots, D_{(n-1)-l}\}$ where each domain value will correspond to each l-length substring. For each pair of variables, there is a constraint requiring that the two substrings corresponding to the two domain values, assigned to the two variables, must have a Hamming distance less than or equal to 2d.

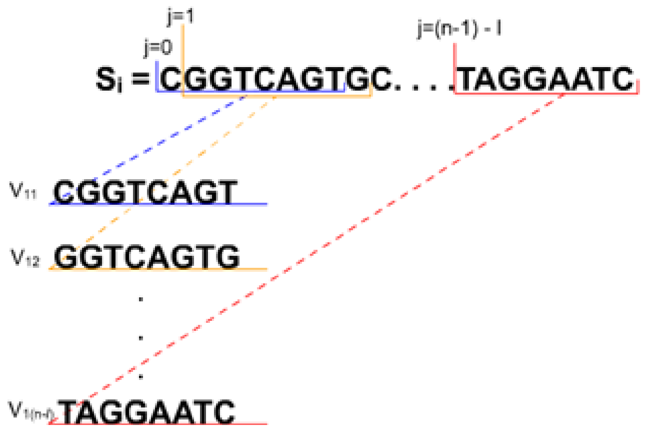
A binary CSP can be represented as a constraint graph where each node represents a variable and each arc represents constraint between two variables. The constraint for an arc is that the Hamming distance between the two l-length substrings (relative to the nodes) must be less than 2d. This constraint graph is constructed in much the same way as the graph in the winnower.

3.1 Constructing the Constraint Graph:

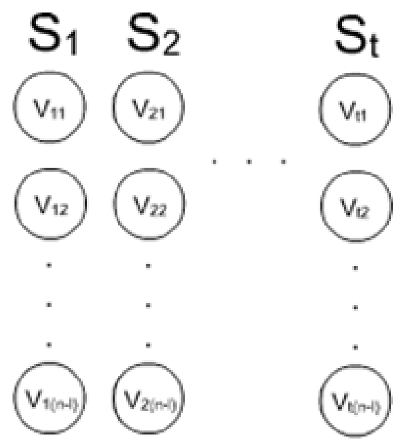
Our graph construction algorithm takes in as input a set of t sequences S = $\{s_1, s_2, \ldots, s_t\}$ (each of length n), a parameter l (length of signal) and a number d (number of mismatches). Given this input, a graph G=(V,E) will be constructed in the following way:

The algorithm iterates through each sequence in the set S and constructs vertices that correspond to substrings of each sequence, that is, for every sequence s_i of the set S and for each index j in sequence s_i we construct a vertex $v \in V$ that will represent the substring s_{ij} of length l from index j to $(n-1) + l$; for a total of $t((n-1) + l)$ vertices.

For example given any arbitrary sequence s_i the figure below illustrates how the sequence would be split into the vertices corresponding to substrings.



After running this procedure across all t sequences we would achieve a multi-partite graph similar to figure below:



After this vertex constructing process the algorithm iterates through each vertex v_i in each *j*th partite S_j and compares v_i against every other vertex in each remaining partite. An edge between two vertices will correspond to vertices that have a hamming distance less than 2*d*, that is, we connect vertices v_{ij} and v_{ab} by an edge if and only if $D_H(v_{ij}, v_{ab}) \leq 2d$ where v_{ij} and v_{ab} are from separate sequences.

Constructing a graph in this way will encapsulate the signal in a clique; if we select any arbitrary edge between any two vertices, corresponding to a signal, that edge will have a maximum hamming distance of at most 2*d* [13].

However constructing the graph in this manner also gives rise to many spurious edges that are not part of the clique we want to find. In order to prune these edges we use constraint propagation. The main idea of constraint propagation is to repeatedly reduce the domain of each variable in order to be consistent with its arcs. Reducing the domains in this manner is what makes constraint propagation an effective pruning mechanism. Applying constraint propagation is what is known as achieving local consistency.

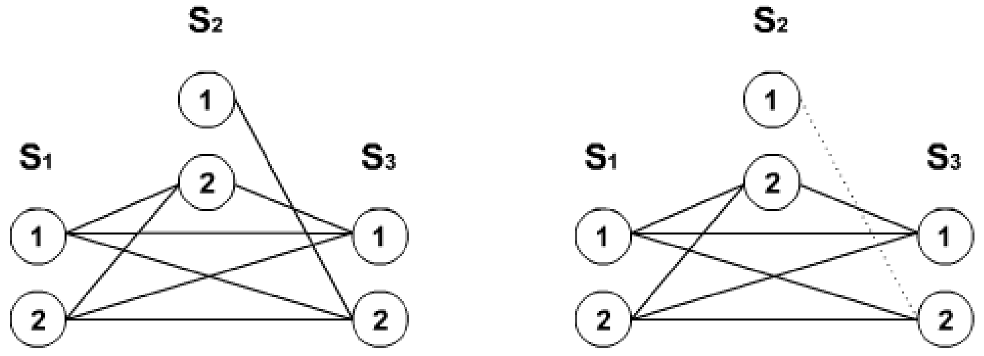
The requirement of local consistency conditions are to take a CSP problem P and map that problem into another problem P’ without altering the original problem’s solutions. This mapping is referred to as constraint propagation. The main idea here is to create a problem that is less complex by reducing the domains of variables by applying constraints or adding new ones. There are several such local consistency conditions that exist; the ones we will concern with here are *arc consistency, path consistency* and the general case *k-consistency*.

3.2 Arc Consistency

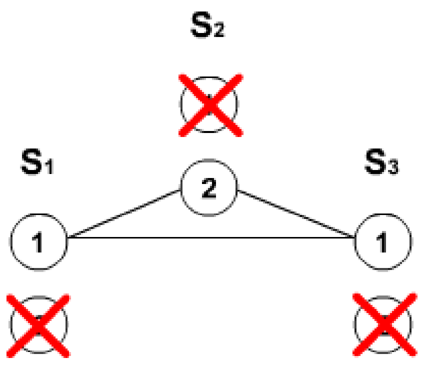
Arc consistency is one of the simplest forms of local consistency and is a process equivalent to the Winnower for k=1. To achieve arc consistency we apply an algorithm that will remove all unsupported values from the domains of individual variables to return a reduced CSP that is easier to solve (or find the problem to have no solution). Generally we say a CSP *P* is arc consistent if each of a variable's admissible values in *P* is consistent with every other variable's admissible values in *P*. Formally:

Definition 7 (Formal Arc Consistency): Given a set of variables X = $\{x_1, x_2, \ldots, x_n\}$ we say variable $x_i \in$ X is arc-consistent with another variable $x_j \in$ X (where $x_i \neq x_j$) if for every value a_i in the domain x_i (there exists) a value b_j in the domain of x_j such that (a_i, b_j) satisfies each constraint between x_i and x_j. We will call such a variable b_j a *support* of a_i.

Graphically we can illustrate how arc consistency works with the following binary network:



We see S_1 is arc consistent with S_2 & S_3. S_3 is arc consistent with S_1 & S_2. S_2 is arc consistent with S_3 however not with S_1. This is because the assignment $S_2 = 1$ does not correspond to any value for S_1 (no supporting variable for S_2=1). We can therefore remove 1 from the domain of S_2. Now S_3 is no longer arc consistent with S_2, because $S_3 = 2$ does not correspond to any value for S_2. We therefore remove 2 from the domain of S_3. We continue to remove values until we end up with a new problem which is arc consistent:



We can use this idea of arc consistency to prune away spurious edges. Given a graph (as constructed by our graph construction algorithm) G=(*V,E*) which is a multi-partite graph with *t* partites, we simply ensure that for every vertex *v* in G there exists a supporting variable *u* in each partite of G. That is, any given vertex *v* that remains after arc consistency is performed on G, will have the property such that there exists a *supporting* vertex *v* in each partite - meaning (*u, v*) is an edge. Any vertex that does not have this property is not arc consistent and therefore will be removed along with any edges that are connected to it. The algorithm we use in CSC is AC6 as proposed by Bessiere and Cordier [3].

3.2.1 Arc Consistency Implementation (AC6)

There are several ways to implement arc consistency. The GAC3 algorithm proposed by Mackworth [2, 10] is a non optimal algorithm that achieves arc consistency in O(er^3d^{r+1}) time and O(*er*) space (where *r* is the greatest rank among constraints) [2]. An algorithm that is shown to be optimal is the AC4 proposed by Mohr and Henderson, with a time complexity of O(ed^2) and a space complexity of O(ed^2) [2, 11, 12]. AC6 is an algorithm proposed by Bessiere and Cordier [3] that compromises between the GAC3 and the AC4 insofar that it maintains the optimal run time complexity of the AC4 (O(ed^2)) with a lower space complexity closer to that of GAC3 (O(*ed*)).

The main idea of the AC6 is not to count how many supporting vertices each vertex has (such as the GAC3 does), but rather to ensure each vertex has at least one supporting vertex in each partite. Upon initialization AC6 incrementally iterates through each vertex in the graph and, for each vertex *v* AC6 attempts to find a supporting variable *u* in each partite for vertex *v*. If such a vertex *u* is found in each partite, then *v* will remain, otherwise it will be deleted. Briefly the AC6 algorithm (Appendix A) works as follows: The AC6 maintains a list S where S[*xj, vj*] stores all supporting values for which v_j is the supporting variable for x_j. The algorithm checks that there is a support v_j for each variable x_j in graph G. If there is no such variable v_j, then x_j will be removed and stored in list Q for future propagation. The propagation loop in AC6 checks the consequences of removing the variables that are stored in Q. When a vertex a_i is chosen from Q, AC6 searches list S to check if a_i was a support for some other vertex x_j. It does so by checking list S for S[x_j, a_j]. If it was a supporting variable then a new support must be found for x_j, if none is found x_j will be removed and in turn be stored in list Q. If there is a support then this new support will replace a_j and stored in list S. The algorithm continues in this manner until the list Q is empty at which point every remaining vertex in graph G will have supporting variables in each partite of graph G, and thus arc consistent.

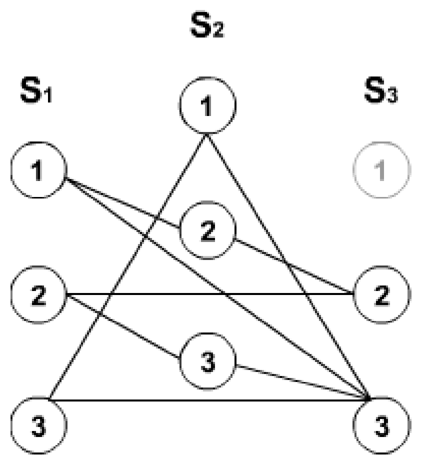
Arc consistency is an effective pruning measure for simple instances of the planted motif problem; however, a stronger consistency is required for more challenging instances [13].

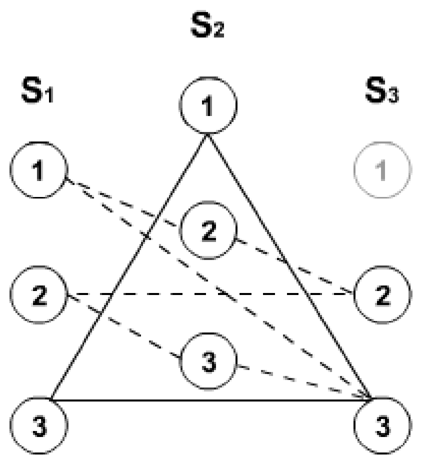
3.3 Path Consistency

Path consistency works much like arc consistency, but considers pairs of vertices rather than a single vertex which is essentially the same pruning mechanism used in the winnower for k=2. We define path consistency formally as:

Definition 8 (Formal Path Consistency): Given a set of variables X = $\{x_1, x_2, \ldots, x_n\}$ we say pair $(x_i, x_j) \in$ X are path consistent with another variable $x_k \in$ X (where $x_i \neq x_j$) if for every pair of values (*a*, *b*) that satisfies the constraint between (x_i, x_j) (there exists) a value *c* in the domain of x_k such that (*a, c*) and (*b, c*) satisfies each constraint between (x_i, x_k) and (x_j, x_k). We will call such a variable x_k a supporting variable of pair (x_i, x_j).

Graphically the figure below show's an example of enforced path consistency:





The figure on the left is an example of a graph that is arc consistent but not path consistent. The dotted edges in the figure on the right indicate edges that would be removed if the graph was to be made path consistent.

We use this idea of path consistency to prune away spurious edges. The same algorithm (AC6) is used with a slight modification. Instead of search for supporting variables for a single vertex x_i we will search for supporting variables for edges (x_i, x_j). That is, for every edge (x_i, x_j) we need to search each partite for a support v_j such that (x_i, x_j, v_j) forms a triangle.

Path consistency is a more powerful pruning mechanism than arc consistency; however, it is still insufficient for difficult instances of the planted motif problem [13]. One way to increase the pruning power is to increase to 4-consistency (k-consistency) which is equivalent to the winnower method for k=3.

3.4 k-consistency

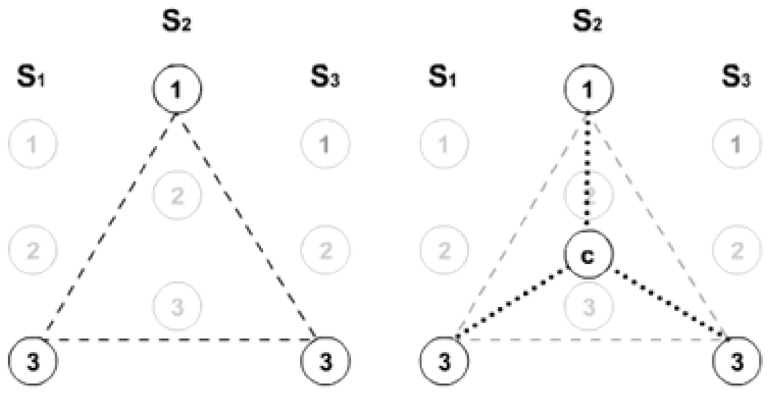
K-consistency generalizes the notion of arc consistency and path consistency to sets of variables:

Definition 8 (k-consistency): Given a set of variables X = $\{x_1, x_2, ..., x_n\}$ we say a sub set of variables $X' \in$ X of size k-1 are k-consistent if for every set of values A = $(a_1, a_2, ..., a_t)$ that satisfies the constraint between all variables in X' there exists a value c in a variable u_k such that no constraint is violated for $(a_1, a_2, ..., a_t, c)$. Such a variable u_k is a supporting variable of X'.

Given this formal definition, arc consistency is equivalent to 2-consistency and path consistency is equivalent to 3-consistency. For the winnower, Pevzner et al. show that k=3 is sufficient as an effective pruning mechanism, which would be equivalent to 4-consistency for our CSP. The tradeoff for this high degree of accuracy is a rather high time complexity of O(N^4). We will now look at a new approach which combines path consistency along with the notion of closest string for a time complexity of O($N^3 + d^d$).

3.5 Content Sensitive Analysis

Instead of increasing to 4-consistency we add a new constraint to our path consistency. Not only do we enforce path consistency but for every pair of nodes that have support we also ensure there is a center string among these three nodes we are analyzing. The figure below shows a graph that is path consistent on the left. The figure on the right shows that a center string c must also exist if the edges in the graph on the left are to remain.



How the algorithm works is described below.

3.5.1 Content Sensitive Analysis Implementation (PC6+CS)

The PC6+CS maintains an array of lists S where S[v_j] stores all a list for which v_j is the supporting variable for all pairs of nodes (x_i, x_j). The algorithm checks that there is a support v_j for each edge (*xi, xj*) in graph G. If there is no such variable v_j, then edge (x_i, x_j) will be removed and stored in list Q for future propagation. If there is a support v_j found the algorithm then uses the CS algorithm to find if there exists a center string *c* for nodes (x_i, x_j, v_j). If such a string exists (x_i, x_j) is stored in list S under otherwise (x_i, x_j) is removed from G and (x_i, x_j) is stored in list Q for future propagation.

The propagation loop in PC6+CS checks the consequences of removing the variable pairs that are stored in Q. When a pair (x_i, x_j) is chosen from Q, PC6+CS searches list S to check if (x_i, x_j) was a supporting edge for some other pair of vertices (x_i, a) or (x_i, a). It does so by checking list S[x_i] for (x_j, a), and S[x_j] for pair (x_i, a). If it was a supporting edge then a new support must be found for both (x_j, a) and (x_i, a), if none is found (x_j, a) and (x_i, a), will be removed and in turn be stored in list Q. If there is a support v_j then, as before, a center string c will be sought after. If such a string c is found then v_j will replace a_j and stored in list S. The algorithm continues in this manner until the list Q is empty at which point every remaining vertex in graph G will have supporting variables in each partite of graph G, and that satisfies our constraints.

4 Experimentation and Results.

In this section we will describe how we generated samples as well as describe our results of running AC6, PC6 and PC6+CS on this these samples. All analysis was based on implementations in Java.

4.1 Sample Generation

To test the effectiveness of our algorithm it is important to be able to generate random samples. Our sample generation algorithm works as follows:

The algorithm takes in as input numbers *t*, *n*, *l* and *d*. Given these numbers the algorithm will generate set of sequences S = $\{s_1, s_2, ..., s_t\}$ where each s_i is of length *n* and each position in s_i is a nucleotide selected at random from alphabet 'A', 'C', 'T', or 'G'. A string *s* of length *l* will be randomly generated using this same alphabet. A set of strings M = $\{m_1, m_2, ..., m_t\}$ are generated where each m_i is a *d*-variant of *s* (i.e: $D_H(s, m_i) = d$ for all *i*). The algorithm then randomly selects a position *j* in each sequence s_i and plants string m_i at this position replacing all nucleotides in sequences s_i from *j* to *j+l*.

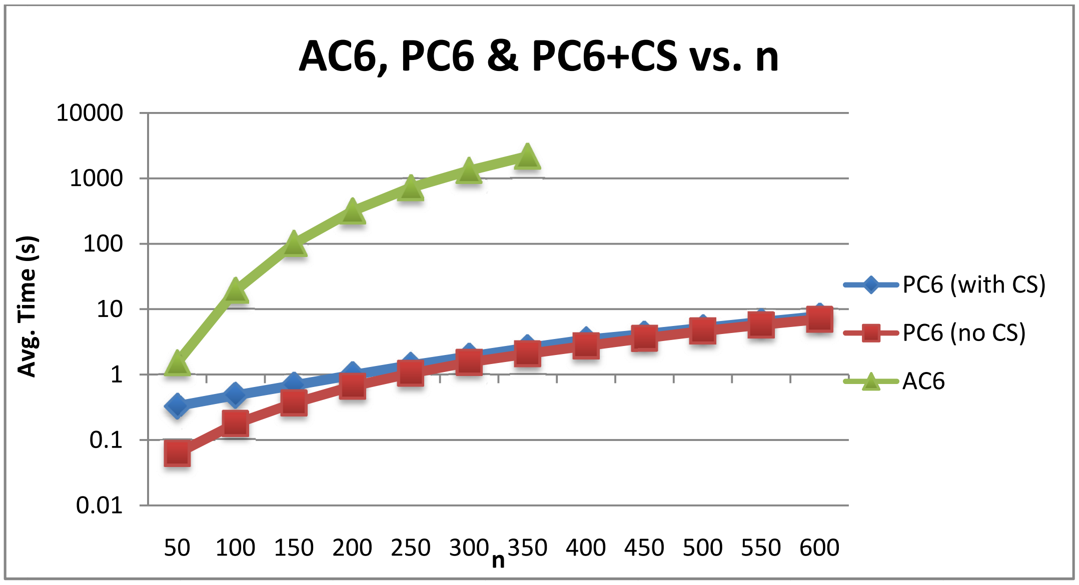
4.2 Results

We test the performance and limitations of AC6, PC6 and PC6+CS on several instances of the Planted (l, d)-Motif Problem.

First we will represent our results based on the (3, 15) problem. We run tests for each algorithm based on parameters *t*=20, *l*=15 and *d*=3. In instances of the (3,15) problem all three algorithms pruned away sufficient edges to expose the signal we are searching for. The chart below is based on the average results obtained from running each algorithm independently on 50 different samples each. The '-' indicates a time that is greater than 1 hour.

Sequence Length	Time (s)		
n	AC6	PC6	PC6+CS
50	1.805	0.063	0.331
100	19.73	0.179	0.482
150	101.601	0.369	0.687
200	317.01	0.666	0.989
250	720.453	1.045	1.393
300	1320.43	1.552	1.902
350	2190.797	2.095	2.569
400	-	2.76	3.431
450	-	3.621	4.168
500	-	4.64	5.188
550	-	5.792	6.398
600	-	7.058	7.848

The graph below illustrates average time (in seconds) as a function of sequence length n, for locating the signal.



For this (3, 15) problem the graph illustrates an interesting phenomenon that occurs. Though the AC6 has the lowest time complexity, of all three algorithms, it has the largest running time in practice. An explanation for this interesting phenomenon is that for PC6 and PC6+CS the condition for edge removal is much stricter, therefore edges are removed more quickly resulting in less iterations and thus a shorter running. A similar observation was also made by Pevzner et al. [8].

AC6 tends to become impractical for n > 350 in comparison to PC6 and PC6+CS. The PC6 and PC6+CS find the signal under 10 seconds for relatively long sequence length (n=600). For this instance they both have roughly the same running time.

Secondly we run tests for the (4, 15) problem. For this problem each algorithm is executed based on parameters t=20, l=15 and d=4. We first compare the AC6 with the PC6+CS. We randomly generated 50 samples for every value of n; for each sample we run the AC6 and then pass this arc consistent graph to the PC6+CS to assess how many edges were not pruned by the AC6. The chart below is based on the average results obtained from running both algorithms sequentially. We see that AC6 is accurate in finding the clique we are searching for n =50 however AC6 is insufficient for n > 50, and will be non effective for removing any edges at all for n ≥ 400.

Sequence Length	**TIME (s)**		**EDGES**		
n	AC6	PC6 + CS	Initial Edges	AC6 prune	PC6 + CS Prune
50	1.512	0.401	31520	31140	0
100	9.612	0.597	163346	151251	11709
150	2.013	4.441	401516	6753	394374
200	0.81	9.523	748116	996	746725
250	0.736	23.255	1202603	132	1202066
300	1.164	63.515	1763251	23	1762820
350	1.639	190.054	2433053	12	2432635
400	2.515	834.385	3211088	0	3210672

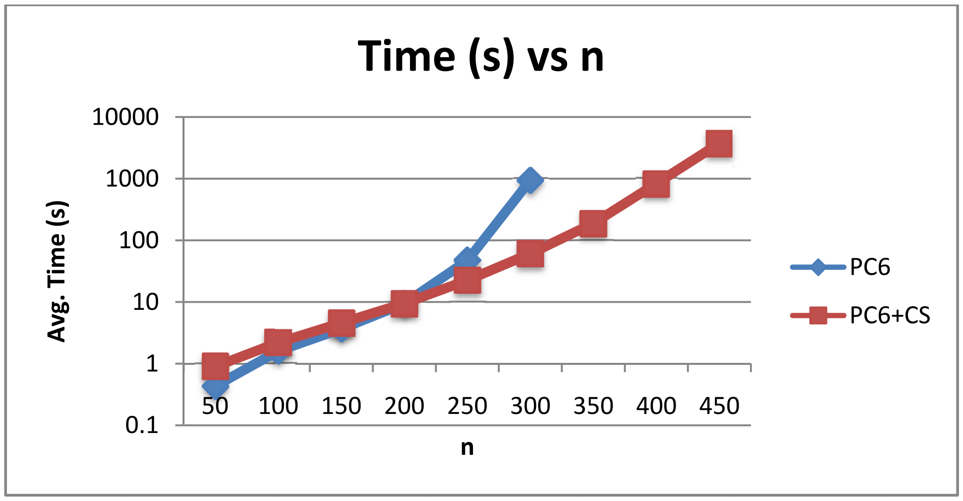
We then executed another 50 tests on the (4, 15) problem comparing accuracy and efficiency between PC6 and PC6+CS. First we run PC6 on each sample, once PC6 is finish the resulting path consistent graph is sent to PC6+CS to examine how many edges PC6+CS will prune from an already path consistent sample. Interestingly the PC6+CS is slightly more accurate and becomes increasingly accurate as n increases, removing a trivial number of edges after the graph has been made path consistent. The running time for of the PC6 become impractical (over 4 hours) for n > 300.

Sequence Length	TIME (s)		EDGES		
n	PC6	PC6 + CS	Initial Edges	PC6 prune	PC6 + CS Prune
50	0.432	0.393	31353	30970	1
100	1.604	0.451	163793	163403	1
150	3.666	0.503	402334	401941	2
200	9.421	0.593	749111	748711	4
250	46.869	0.725	1202661	1202250	8
300	941.822	0.852	1764138	1763728	7

Finally we run a separate test on the (4, 15) problem using only the PC6+CS. For our implementation of the PC6+CS, the running time becomes impractical for n> 450.

Sequence Length	TIME (s)	EDGES	
n	PC6 + CS	Initial Edges	PC6 + CS Prune
50	0.898	31570	31188
100	2.209	163444	163057
150	4.562	402265	401872
200	9.497	748845	748450
250	22.534	1201833	1201438
300	608.37	1764560	1764160
350	188.156	2433053	2432647
400	834.385	3211088	3210672
450	3754.396	4092968	4092541

The graph below compares the running times of the both PC6 and PC6+CS with time as a function of n and illustrates that in practice, PC6+CS is a slightly more efficient pruning mechanism than the PC6.



5 Discussion and Conclusion

The problem of motif finding plays a very important role in understanding gene regulatory mechanisms. An area of bioinformatics is concerned with modeling this problem in a way that will allow computational algorithms to lend aid in locating these motifs. One such model includes the Planted (*l, d*)-Motif Problem. In this thesis we have analyzed the efficiency and accuracy of several algorithms designed to solve this particular Planted (l, d)-Motif Problem, including: the Basic Voting Algorithm, the Winnower and the Closest String. We have presented a new formulation that composes the Planted (l, d)-Motif Problem as a CSP as well as introduced a new form of center string constraint consistency. Experimental results on a Java implementation, based on this formulation, locate signals accurately and efficiently for simulated data. However, using this implementation, we were unable to conclude the relative strength of this center string consistency when comparing it to the notion of Hamming distance. The impractical running times for this implementation did not allow us to test for large values of *n* where the PC6 is known to fail. This may be due to either a non optimal implementation or the relatively slow performance of Java. Future work may include an implementation in C so that we could achieve a practical running time with a large enough value of *n* such as $n \geq 700$ for the (4, 15) where the PC6 is known to fail [8]. Running the PC6+CS for these values of *n* is required to analyze the relative pruning strength of this new notion of consistency.

Appendex A

Algorithm 1: AC6 [2, pg. 20]

```
X ← 2D constraint graph
Q ← ø
S[$x_j$ , $v_j$] = 0, $\forall v_j \in$ D(xj ), $\forall x_j \in$ X
**foreach** $x_i \in$ X, $c_{ij} \in$ C, $v_i \in D(x_i)$ **do**
        $v_j$ ← smallest value in D($x_j$) s.t. ($v_i$, $v_j$) $\in c_{ij}$
        **if** $v_j$ exists **then** add ($x_i$, $v_i$) to S[$x_j$ , $v_j$]
        **else** remove $v_i$ from D($x_i$) and add ($x_i$, $v_i$) to Q
        **if** D($x_i$) = ø **then** return **false**
**while** Q ≠; do
        select and remove ($x_j$ , $v_j$) from Q;
        **foreach** ($x_i$, $v_i$) $\in$ S[$x_j$, $v_j$ ] **do**
        **if** $v_i \in$ D($x_i$) **then**
                $v'_j$ ← smallest value in D($x_j$) greater than $v_j$ s.t. ($v_i$, $v_j$) $\in c_{ij}$
                **if** $v'_j$ exists **then** add ($x_i$, $v_i$) to S[$x_j$, $v'_j$]
                **else**
                        remove $v_i$ from D($x_i$); add ($x_i$, $v_i$) to Q;
                        **if** D($x_i$) = ø **then** return false
return **true**
```

Algorithm 2: Closest String [4, pg. 6]

recursive procedure: CSd(s, Δd)
Global variables: Set of strings S = {s_1, s_2, . . . , s_k}, integer d.
Input: Candidate string s and integer Δd.
Output: A strings with $\max_{i=1,\dots,k} d_H(s, si) \le d$ and $d_H(s, s) \le \Delta d$ if it exists, and "not found," otherwise.

```
**if** (Δd < 0), **then return** "not found";
**if** ($d_H$(s, $s_i$) > d + Δd) for some i $\in$ {1, . . . , k} **then return** "not found";
**if** ($d_H$(s, $s_i$) ≤ d) for all i = 1, . . . , k then **return** s;
Choose some i $\in$ {1, . . . , k} such that $d_H$(s, $s_i$) > d:
        P := { p | s[p] $\neq$ $s_i$[p] };
        Choose any P' $\subseteq$ P with |P'| = d + 1;
        **for all** p $\in$ P do
                s':= s;
                s'[p] := $s_i$[p];
                $s_{ret}$:= CSd(s , Δd − 1);
                **If** $s_{ret}$ $\neq$ "not found" **then return** $s_{ret}$
**return** "not found"
```

Algorithm 3: PC6+CS

```
// initialization
C ← a 2D constraint graph
Q ← ø // Q list stores all deleted pairs of variables
S[n] ← ø // where s[i] stores all pairs of variables supported by i.
D ← all pairs of variables that have not yet violated any constraint

// begin
foreach pair ($x_{ai}$, $x_{bj}$) do // where a and b indicate partites which x is a member
        foreach partite c where c ≠ a and c ≠ b do
                $v_c$ == checkForSupport($x_{ai}$, $x_{bj}$, c)) //return -1 if no support found
                if ($v_c$ ≠ -1) and (centerString($x_{ai}$, $x_{bj}$, $v_c$)) then
                                S[$V_c$] ← ($x_{ai}$, $x_{bj}$)
                else
                        C[$x_{ai}$][$x_{bj}$] = 0
                        Q ← ($x_{ai}$, $x_{bj}$)
                        remove ($x_{ai}$, $x_{bj}$) from D
                        if D empty return false // no feasible solution

// propagation
while Q ≠ ø do
        select and remove ($x_{ai}$, $x_{bj}$) from Q
        S' ← each triple ($y_{ai}$, $y_{bi}$, $v_c$) of variables for which edge ($x_{ai}$, $x_{bj}$) was a support
        foreach ($y_{ai}$, $y_{bj}$) ∈ S' do
                if ($y_{ai}$, $y_{bj}$) ∈ D then
                        $v_c$ == checkForSupport($y_{ai}$, $y_{bj}$, $v_c$)) //starting at index $v_c$ in c
                        if ($v_c$ ≠ -1) and (centerString($y_{ai}$, $y_{bj}$, $v_c$)) then
                                S[$V_c$] ← ($y_{ai}$, $y_{bj}$)
                else
                        C[$x_{ai}$][$x_{bj}$] = 0
                        Q ← ($y_{ai}$, $y_{bj}$)
                        remove ($y_{ai}$, $y_{bj}$) from D
                        if D empty return false // no feasible solution
return true
```

Algorithm 4: Basic Voting Algorithm [8, pg. 4]

Create two hash tables V and R and set the value of each entry to be 0
{Table V keeps the number of votes received by each length-*l* sequence *s*. Each length-*l* sequence, received *t* votes is a candidate for motif. Hash table R ensures that each length-*l* sequence receives at most one vote from each input sequence. $S_i\{j\}$ is the *j*-th character in the *i*-th input sequence S_i and H(*s*) is the hash value of a length-*l* sequence *s*.}

```
C ← ø
for i ← 1 to t
    do for j ← 1 to n – l + 1
        do for each length-l sequence is N(S_i[j … j + l – 1],d)
            do if R[H(s)] <> i
                then V[H(s)] ← V[H(s)] + 1
                    R[H(s)] ← i
for j ← to n – l + 1
    do for each length-l sequence s in N(S_i[j … j + l – 1],d)
        do if V{H(s)]= t
            then insert s into C
```

References

[1] S. Balla, J. Davila and S. Rajasekaran. On the Challenging Instances of the Planted Motif Problem. http://www.engr.uconn.edu/becat/reports/BECAT-CSE-TR-07-2.pdf

[2] C. Bessiere. Constraint Propagation. *Handbook of Constraint Programming.* Ch. 3, Pg. 29 – 85 ISBN 0-444-52726-5

[3] C. Bessiere, M.O. Cordier. Arc-consistency and arc-consistency again. *In Proceedings AAAI'93*, pages 108–113, Washington D.C., 1993.

[4] J. Gramm R. Niedermeier. Fixed-Paramter Algorithms for Closest String and Related Problems. *Algorithmica* [0178-4617] Gramm yr:2003 vol:37 iss:1 pg:25

[5] J. Hu, B Li, D. Kihara. Limitations and potentials of current motif discovery algorithms. *Nucleic Acids Res* 2005, 33(15):4899-4913.

[6] R. M. Karp. Reducibility Among Combinatorial Problems. In Complexity of Computer Computations, *Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights*, N.Y.. New York: Plenum, p.85-103. 1972.

[7] C. Lawrence, S. Altschul, M. Boguski, J. Liu, A. Neuwald and J. Wootton. Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. *Science*, 262:208-214, 1993.

[8] H. C.M. Leung, F.Y.L. Chin. Algorithms for challenging motif problems. *Journal of bioinformatics and computational biology* [0219-7200] Leung yr: 2006 vol: 4 iss: 1 pg: 43 -58

[9] M. Li, B. Ma, L. Wang. 2002. On the Closest String and Substring Problems. *Journal of the ACM*, Vol. 49, No. 2, March 2002, pp. 157-171.

[10] A.K. Mackworth. Consistency in networks of relations. *Technical Report 75-3, Dept. of Computer Science, Univ. of B.C. Vancouver*, 1975. (also in *Artificial Intelligence* 8, 99-118, 1977).

[11] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[12] R. Mohr and G. Masini. Good old discrete relaxation. *In ProceedingsECAI'88*, pages 651–656, Munchen, FRG, 1988.

[13] P.A. Pevzner, S.H. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. *Proc. Int. Conf. Intell.* Syst. Mol. Biol., 269–278, AAAI Press, 2000.

[14] M. Vingron, and P. Argos, 1991. Motif recognitionand alignment for many sequences by comparison of dot-matrices. *Journal of Molecular Biology* 218:33-43.

[15] M.Vingron, and P. Pevzner, 1995. Multiple sequence comparison and consistency on multipartite graphs. *Advances in Applied Mathematics* 16:1-22.

[16] M.S. Waterman, R. Arratia and D.J. Galas. Pattern recognition in several sequences: consensus and alignment. *Bull. Math. Biol.,* 46:515-527. 1984.