GOING CRITICAL:
AN INVESTIGATION OF DIAMETER-CRITICAL GRAPHS

by

JOSHUA MADDEN

B.A., University of Oregon, 1994

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

in

THE FACULTY OF GRADUATE STUDIES

Department of Mathematics
Institute of Applied Mathematics

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

April 1999

© Joshua Madden, 1999
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Mathematics

The University of British Columbia
Vancouver, Canada

Date 30 April 1999
Abstract

We define a graph $G = (V, E)$ with $m = |E|$ edges, $n = |V|$ vertices, maximum degree $D$, and diameter $d$, to be $d$-critical if it has the property that for any edge $e \in E$, the graph $G - e$ has diameter $> d$. We explore some results relating to a conjecture on the maximum possible number of edges in a 2-critical graph. We also present efficient algorithms for identifying spanning $d$-critical subgraphs of graphs having diameter $d$ where $d = 2$ or $d = 3$. The algorithm for $d = 2$ runs in $O(mn)$, and the algorithm for $d = 3$ runs in $O(mnD)$. 
# Table of Contents

Abstract ii  
Table of Contents iii  
List of Figures iv  
Acknowledgements v  

Chapter 1. Introduction 1  
1.1 Background and Motivation ........................................ 1  
1.2 Problem Domain ...................................................... 5  
  1.2.1 Weighted vs. Unweighted Graphs ................................ 5  
  1.2.2 Directed vs. Undirected Graphs .................................. 6  
1.3 Terminology and Notation .......................................... 7  

Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles 9  
2.1 2-Critical Graphs and Triangles .................................... 9  
  2.1.1 2-Critical Graphs with All Edges in Triangles ................. 14  
2.2 Examples of 2-Critical Graphs with $\Theta(n^3)$ Triangles .......... 15  
  2.2.1 The Graph Family $\mathcal{G}_p$ .................................. 19  
  2.2.2 The Graph Families $\mathcal{H}_p$ and $\mathcal{W}_p$ .................. 23  
2.3 Analysis of our Examples .......................................... 25  
  2.3.1 Associated Triangles ........................................... 26  
  2.3.2 Cliques, Triangles, and Edges ................................... 27  

Chapter 3. $d$-Critical Spanning Subgraphs 34  
3.1 Calculating the Diameter of a Graph ............................. 34  
  3.1.1 Breadth-First Search Trees .................................... 34  
  3.1.2 Matrix Multiplication ........................................... 35  
3.2 Recognizing $d$-Critical Graphs ................................... 40  
3.3 Identifying $d$-Critical Spanning Subgraphs ....................... 41  
  3.3.1 Properties of Breadth-First Search Trees ...................... 45  
  3.3.2 Algorithm for Diameter 2 Graphs ................................ 47  
  3.3.3 Algorithm for Diameter 3 Graphs ................................ 55  
  3.3.4 Algorithms for General Graphs .................................. 60  
3.4 Concluding Remarks .............................................. 63  
  3.4.1 Further Uses of Matrix Multiplication ......................... 63  
  3.4.2 Parallelization ................................................. 64  

Bibliography 65
## List of Figures

1.1 Transformation from a weighted edge to a series of unweighted edges .................. 6  
2.1 An example of an extremal triple ............................................................................. 12  
2.2 Members and non-members of $S_1$ ........................................................................ 12  
2.3 The family of graphs $A(r)$ .................................................................................. 15  
2.4 The graph $B(2, 3)$ ............................................................................................. 16  
2.5 Original concept for 2-critical graph with critical triangle $(x, y, z)$ ................. 18  
2.6 $G_1, G_2$ ............................................................................................................... 30  
2.7 $G_3, G_4$ ............................................................................................................... 31  
2.8 $H_1, H_2, H_3$ ....................................................................................................... 32  
2.9 $W_1, W_2, W_3$ ..................................................................................................... 33  
3.1 An example of a BFS tree and its virtual edges ..................................................... 45  
3.2 An example of case 2 for diameter 2: $T_i$ .............................................................. 49  
3.3 An example of case 3 for diameter 2: $T_k$ .............................................................. 51  
3.4 An example of case 2 for diameter 3: $T_i$ .............................................................. 56  
3.5 An example of case 3 for diameter 3: $T_k, l(T_k, i) = 1, l(T_k, j) = 2$ ............ 57  
3.6 An example of case 4 for diameter 3: $T_k, l(T_k, i) = 2, l(T_k, j) = 3$ ............ 58
Acknowledgements

I would like to thank a number of people for their encouragement and assistance in the writing of this work: my supervisor, Richard Anstee; my parents, Bill and Beth Madden; my leman, Megan Foster; my academic mentors, including (but not limited to) Frances Cogan, Bill Kantor, Tom McCormick, Andrzej Proskurowski, Hollis Williams, and Chris Wilson; Jack Snoeyink and Joel Friedman, for providing much of my financial support; Alan Mackworth, David Poole, and Craig Boutilier, for pointing the way to the next stage of my education; Alex West and Jeff Lassahn, for helping to keep my mathematical talents alive in the years between finishing my B.A. and starting my M.Sc.; the folks in the UBC Computer Science Theory Seminar (David Kirkpatrick, Hamish Carr and Andrea Mantler in particular), who suggested a number of improvements and extensions; and a number of innocent bystanders who held still long enough for me to explain what my thesis was about, despite their lack of background (or, occasionally, interest!) in mathematics. Their contributions have been invaluable.
Chapter 1
Introduction

1.1 Background and Motivation

A graph consists of a collection of objects (vertices) and the connections between them (edges). Graphs, which are more commonly known as networks in nonmathematical contexts, are ubiquitous in today's society. Some of their manifestations are explicit: for example, a modern national transportation system consists of cities and the available paths between them, e.g., highways, railroads, airplane routes, and so on. However, this does not exhaust the representational power of graphs by any means. Many problems in communication, resource management, scheduling, planning, and countless other subjects are very naturally represented by graphs, and the best solutions known to these problems have generally been arrived at by the study of graphs and their properties. Indeed, one could make the argument that the modern world is characterized by graphs, as the complex interconnections between people and corporations which graphs so naturally represent are an obvious concomitant of the emerging global society.

When one constructs a graph for a specific purpose, especially in the real world, one is often concerned with the expenditure necessary to build the associated structure which the graph represents. To return to our example of transportation, you probably would not want to build a separate road connecting each pair of cities in Canada; many of these roads would hardly be used. Nor is it likely that you would want to construct six separate
Chapter 1. Introduction

roads just to connect two cities. On the other hand, if you built the smallest possible
number of roads sufficient to connect all the cities, many routes would be inconveniently
long. Furthermore, the more roads that you have to take, the more likely you are to get
lost or take a wrong turn. Thus, we formulate the following problem:

“How many roads do we have to build so that you can get from any city in
Canada to any other using only a few roads?”

Using ideas from graph theory, this problem can be approached in two different ways:

• Find a graph that represents the system, satisfies the requirements of the system,
  and has the property that no subgraph of this graph satisfies the system’s require­
  ments. Such a graph is called a *minimal* graph with respect to the requirements.

• Find a graph that represents the system, satisfies the requirements of the system,
  and has as few edges as possible. This graph is called a *minimum cost* graph with
  respect to the requirements (assuming that the cost of a graph is defined as the
  number of edges which comprise it).

Generally speaking, finding a minimal solution to a problem in graph theory is not
difficult, but finding a solution of minimum cost to the same problem is prohibitively
time-consuming. (Note that a minimum cost solution is usually also a minimal solution
by default.) There are exceptions to this generalization, of course; unfortunately, as far as
we are aware, this particular problem is not one of them. (See Chapter 3 for more details
on this issue.) However, in many cases a minimal solution is what we are interested in,
or is at least good enough for our purposes.

One of the subjects of study in the field of extremal graph theory is those graphs which
are minimal with respect to some set of requirements. Formally, we say that a graph $G$
having property $P$ is *critical* with respect to $P$, or $P$-critical, if no subgraph of $G$ has
property $P$. 
Chapter 1. Introduction

A fundamental property of graphs whose behavior is not always well understood in general is diameter. One can readily calculate the distance between all pairs of vertices in a graph (where the distance between vertices \(i\) and \(j\) in a graph is defined as the length of the shortest path in the graph which connects \(i\) to \(j\)). The diameter of a graph is defined as being the longest distance between any two vertices in the graph. It is straightforward to calculate the diameter of a given graph, but it is sometimes difficult to make generalizations about the diameter of an arbitrary member of a family of graphs. One reason for this is that local changes in a graph (e.g., removing an edge) can have global effects on the diameter.

For an example of the practical relevance of the diameter of a graph, we consider those graphs which represent communications networks. The diameter of a network (graph) represents the length of the longest path that a signal must travel between two nodes in the network (vertices in the graph), assuming that a signal always uses the shortest available path. A signal propagates across a network in time proportional to the length of the path taken, and the quality of the signal degrades in proportion to the number of links in the network (edges in the graph) that the signal must traverse in order to reach its destination. Thus, the diameter of a network is a crucial measure of its performance.

Suppose, then, that we wish to design a communications network, subject to the requirement that there should be a path of no more than \(d\) links between any two nodes. A network that satisfies this requirement, and has the property that no subnetwork satisfies this requirement, is called diameter-critical in general, and \(d\)-critical if \(d\) is specified.

The purpose of this thesis is to explore some results relating to two issues in the theory of diameter-critical graphs. One of these issues is straightforward: we would like to be able to efficiently find a diameter-critical subgraph of a given graph. This is useful for improving the efficiency (in terms of number of edges needed) of a graph which represents
Chapter 1. Introduction

a network. If diameter is our chosen measure of performance, we would like to find a minimal graph of the same diameter as the graph that we were given. The algorithms that we have developed for this task are presented in Chapter 3.

The other issue is more subtle, and relates to a conjecture in this field regarding the maximum number of edges that a critical graph of diameter 2 (2-critical graph) can have. Briefly, the conjecture states that a 2-critical graph of \( n \) vertices can have no more than \( \frac{5n^2}{4} \) edges. In Chapter 2, we explore this conjecture by examining the properties of 2-critical graphs which have a large number of triangles. It turns out that the truth of the conjecture turns on whether it is possible to construct a 2-critical graph with \( \Theta(n^3) \) triangles that have certain specific characteristics.

As a practical example of the importance of the number of edges in a given (2-critical) graph, we observe that the design of a communications network involves balancing, among other things, the following three factors:

- Cost and complexity: every link added to a network increases its cost and its complexity. Complex networks are more prone to failure of individual components, and their errors are more difficult to diagnose and repair.

- The desire to have available alternate efficient paths between any two nodes in the network (in case a link fails or is busy).

- The intended topology of the network. The topology is the general scheme for how the nodes are connected to one another, which may be dictated to a degree by hardware restrictions. For example, there is usually a limit to the number of links that can be connected to a given network node.

We present a number of families of 2-critical graphs with many triangles, which therefore also have a large number of edges. The fact that there exist many such graphs with
distinct topologies gives a network designer a number of options if they wish to use a 2-critical graph with a large number of edges as the network template.

1.2 Problem Domain

1.2.1 Weighted vs. Unweighted Graphs

A weighted graph is a graph for which each edge has an associated (positive) weight. This weight can represent the cost of traversing that edge, or the time necessary to traverse it, or any similar scalar quantity. The diameter of a weighted graph is defined in similar fashion to that of an unweighted graph. The diameter is still considered to be the maximum of the distances between any pair of nodes in the graph, the distance between a pair of nodes \( (i, j) \) is defined to be the length of the shortest \( ij \)-path, and the length of an \( ij \)-path is defined to be the sum of the weights of the edges which comprise the path (rather than the number of edges which comprise the path, as in the unweighted case).

This thesis considers only unweighted graphs, and we offer the following justifications:

- When we consider communications networks, it is often the case that the crucial factor in determining the transmission time from one node to another has little to do with the lengths of the links between nodes in the network, because the signal travels at the speed of light along the link. Instead, the transmission time is largely determined by the time it takes for the router at a network node to decode the address of the intended recipient and send the signal along the correct link. This time may be quite large in comparison to the link transmission time, especially if there are several signals which are attempting to pass through the node simultaneously. Thus, the total transmission time is often proportional to the number of nodes on the path from the source node to the destination node, or, in other words, the number of links traversed. In this case, the graph representing the network is effectively unweighted.
Chapter 1. Introduction

• Suppose that we have a weighted graph $W$ with a weight function $w : E \rightarrow \mathbb{N}$ (i.e., each edge has positive integer weight). We define a mapping from $W = (V, E)$ to an unweighted graph $U = (V', E')$ as follows: replace each weighted edge $e = (i, j) \in E$ with $w(e)$ consecutive unweighted edges in $E'$ which join $w(e) - 1$ new vertices in $V'$ placed between $i$ and $j$. (See Figure 1.1, below.)

![Figure 1.1: Transformation from a weighted edge to a series of unweighted edges](image)

This mapping preserves the structure of the graph: for any vertex $v \in V(W)$, there exists a corresponding vertex $v' \in V'$ such that $\text{deg}(v) = \text{deg}(v')$, where $\text{deg}(v)$ is the number of edges in $E$ incident to $v$. If one considers the weight of an unweighted edge to be 1, the sum of the weights of the edges on the new $ij$-path is exactly $w(e)$. So long as the weights in $W$ are not too large (i.e., the size of the maximum weight is polynomial in $m$ and $n$) the new unweighted graph provides an efficient representation of the old one.

1.2.2 Directed vs. Undirected Graphs

An undirected graph is one in which an edge $e = (i, j)$ is an unordered pair of vertices, and thus can be traversed in either direction (either from $i$ to $j$, or from $j$ to $i$). By contrast, a directed graph is a graph in which an edge $e = (i, j)$ is an ordered pair of
vertices, i.e., edge has an associated orientation and can only be traversed in the direction indicated by the ordering: \(i \rightarrow j\).

We consider only undirected graphs for several reasons:

- The conjecture which we address in Chapter 2 relates to undirected, unweighted graphs.

- For at least two of the major classes of applications for diameter-critical networks (transportation and communication) it is reasonable to suppose in many cases that network links (graph edges) may be traversed in either direction.

- In order to apply the results of Chapter 3 to directed graphs, we would need to make only a few minor modifications to our definitions and algorithms. For instance, the definition of diameter would remain the same for a directed graph \(D\): “the longest distance between any two vertices \(i\) and \(j\) in \(D\)”.

1.3 Terminology and Notation

- A graph \(G\) has edge set \(E\) or \(E(G)\), vertex set \(V\) or \(V(G)\), \(m = |E|\) edges, \(n = |V|\) vertices, and diameter \(diam(G) = d\).

- A graph which has diameter \(d\) and is critical with respect to the operation of edge removal is called \(d\)-critical. Formally, a graph \(G\) with diameter \(d\) is \(d\)-critical if and only if \(diam(G - e) > d\). (Plesnič, in [12], refers to such a graph as an edge minimal graph of diameter \(d\).)
Chapter 1. Introduction

- An edge \( e \) is called \( d \)-critical if and only if \( \text{diam}(G) = d \) and \( \text{diam}(G - e) > d \).

- The degree of a vertex \( v \), denoted \( \text{deg}(v) \), is the number of edges that are incident to \( v \) (or, equivalently, the number of vertices to which \( v \) is connected). We denote the maximum degree of any \( v \in V(G) \) as \( D \) or \( D(G) \). Note that \( D \) is in \( O(n) \).

- For \( x, y \in V(G) \), we use \( x \sim y \) to indicate that there exists an edge connecting \( x \) to \( y \); the edge is denoted \( xy \). We use \( x \not\sim y \) to indicate that no such edge exists.

- \textit{triangle}: a cycle in a graph of length 3.

- The set of triangles in a graph \( G \) is denoted \( \Delta \) or \( \Delta(G) \).

- \textit{simple graph}: a graph having no parallel edges or loops. All graphs that we consider are simple.

- \textit{ij-path}: a path, having no repeated vertices, from vertex \( i \) to vertex \( j \).

- Given \( S \subseteq V \), \( \langle S \rangle \) denotes the subgraph of \( G \) induced by \( S \).

- The \textit{neighbor set} \( N(v) \) of a vertex \( v \in V \) is defined to be the set of vertices adjacent to \( v \), \textit{i.e.}, those vertices \( u \in V \) such that \( v \sim u \).

- Given a connected graph \( G = (V, E) \), a \textit{spanning subgraph} of \( G \) is a connected subgraph \( S \) of \( G \) with vertex set \( V \) and edge set \( E' \subseteq E \).
Chapter 2

2-Critical Graphs with $\Theta(n^3)$ Triangles

The purpose of this chapter is to discuss some approaches to proving a conjecture regarding the maximum possible number of edges in a 2-critical graph. Pursuant to this, we will relate this conjecture to a result that frames the conjecture in terms of 2-critical graphs with $\Theta(n^3)$ triangles, present some examples of such graphs, and discuss the reasons why the examples are not counterexamples to the conjecture.

2.1 2-Critical Graphs and Triangles

Graphs that have a large number of triangles are of particular interest in the context of constructing 2-critical graphs. A triangle in isolation is not, itself, 2-critical. In fact, it can readily be shown that all diameter 2 graphs with no triangles are 2-critical.

**Proposition 2.1** A simple graph $G$ of diameter 2 which has no triangles is 2-critical.

**Proof**: Consider two vertices $i, j \in V$ connected by an edge $ij$. Suppose that we remove $ij$ from $G$. Any $ij$-path of length 1 would have to be an edge parallel to $ij$. Suppose, then, that there exists an $ij$-path of length 2 which passes through some vertex $k$; this would imply the existence of edges $ik$ and $jk$. However, if such edges existed, they would
form the triangle $ikj$ with the edge $ij$. Thus, $ij$ must be 2-critical, and therefore $G$ is 2-critical.

We can see from the proof that in order for a triangle to exist in a 2-critical graph, there must be some associated structure in the graph which requires the presence of each edge in the triangle in order to maintain diameter 2. Given these facts, one might suppose that 2-critical graphs have few triangles, if they have any at all.

We are interested in 2-critical graphs with many triangles not only because they are perhaps counterintuitive or unusual, but as a byproduct of our interest in dense 2-critical graphs as mentioned in Chapter 1. A theorem due to Mantel [8] places a bound on the number of edges that a general graph can have if it has no triangles. (This theorem was proved later in a more general form by Turán [13], and is generally known as Turán’s Theorem.)

**Theorem 2.2 (Mantel [8], Turán [13])** If a graph $G$ has no triangles, then

$$m \leq \lceil n^2 / 4 \rceil,$$

with equality holding if and only if

$$G \cong K_{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor}.$$

There is a conjecture with very similar form, proposed independently by Simon and Murty (referenced in [3]) and Plesník [10], that relates the number of edges to that of vertices in a 2-critical graph.

**Conjecture 2.3 (Simon and Murty [3], Plesník [10])** If $G$ is a 2-critical graph, then

$$m \leq \lfloor n^2 / 4 \rfloor,$$
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

with equality holding if and only if

$$G \cong K_{[n/2],[n/2]}.$$ 

A result of Füredi [7] has proven the conjecture for $n > n_0$. However, $n_0$ is an inconceivably (and inconveniently) huge number: it is a tower of 2s of height approximately $10^{14}$. Since, for practical purposes, we are usually interested in graphs which are smaller than this, further investigation is warranted.

The conjecture has been proven by Fan [6] for $n \leq 24$ and $n = 26$. Fan also proved that the upper bound on $m$ (for $n \geq 25$) is not greater than

$$\frac{n^2}{4} + \frac{n^2 - 16.2n + 56}{320} \approx 0.2532n^2.$$ 

Xu [14] presented a general proof of this conjecture; however, he retracted it later after finding a fault in his proof.

If we consider only graphs with no triangles, the proof of the conjecture follows trivially from Turán’s Theorem. The similarity between the conjecture and Turán’s Theorem suggests that if one could bound the number of triangles in a 2-critical graph, this might contribute to a general proof of the conjecture.

Pursuant to this goal, Caccetta and Häggkvist [3] demonstrated a result of particular interest: they showed that the conjecture is true for those graphs which have $O(n^{3-\epsilon})$ triangles with a certain property. In order to present this result, we must first establish some additional terminology.

- Let $(x, y, z)$ be an unordered triple of distinct vertices such that $|\{(x, y, z)\}| = 2$. (Recall that $\langle S \rangle$ denotes the subgraph of $G$ induced by $S \subseteq V$.) Without loss of generality, let $x \neq y$. Then if $N(x) \cap N(y) = \{z\}$, then $(x, y, z)$ is an extremal triple. Figure 2.1 shows an example of an extremal triple.
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

Figure 2.1: An example of an extremal triple

- $S_1 = \{(x, y, z) : E\{x, y, z\} = \{yz\}, N(x) \cap N(y) \cap N(z) = \{w\}, \text{and one or more of } \{(x, y, w), (x, z, w)\} \text{ is an extremal triple}\}.$

Figure 2.2: Members and non-members of $S_1$

(a) and (b) are examples of members of $S_1$; (c) is not, because neither $(x, y, w)$ nor $(x, z, w)$ is an extremal triple.

Let $\tau \in \Delta$ with vertices $(x, y, z)$. (Recall that $\Delta$ denotes the set of triangles in $G$.) Without loss of generality, we select the edge $xy$ from $\tau$. If $G$ is 2-critical, $xy$ must lie on the unique shortest path from some vertex $a \not\in \tau$ to either $x$ or $y$. Thus, there is a
vertex $a \in G - \tau$ such that $a \sim x$, or $a \sim y$, but not both, and $a \not\sim z$. Again, without loss of generality, let $a \sim y$. As a result, $N(x) \cap N(a) = \{y\}$. Therefore, $(a,x,y)$ is an extremal triple, and $(a,x,z) \in S_1$.

We say that $\tau$ is associated with $(a,x,z)$, and observe that no two triangles in $G$ can be associated with the same element of $S_1$. Also, because each element of $\tau$ must be critical, each edge of $\tau$ belongs to an extremal triple. Thus, $\tau$ must be associated with at least two elements of $S_1$. In Figure 2.2(a), the triangle $(w, y, z)$ is associated with one element of $S_1$, to wit, $(x, y, z)$.

Finally, we define

$$S_2 = \{(x,y,z) : (x,y,z) \in \Delta \text{ and is associated with exactly two elements of } S_1\}.$$ 

**Proposition 2.4 (Caccetta and H"{a}ggkvist [3])** If $G$ is a 2-critical graph,

$$m \leq \frac{n^2}{8} \left( 1 + \sqrt{1 + \frac{16|S_2|}{n^3}} \right).$$

This implies that the conjecture is true for the class of 2-critical graphs having the property that $|S_2|$ is in $O(n^{3-\epsilon})$, since in that case the value of the expression asymptotically approaches $n^2/4$ as $n$ increases without bound. Thus, we may restrict our search for graphs which might constitute counterexamples to the conjecture to those graphs with $\Theta(n^3)$ triangles. We note that while this is a necessary property of a counterexample, it is not a sufficient property; the proposition provides only an upper bound on the number of edges of such a graph, and not a lower bound.

While we were initially unaware of any examples in the literature of families of 2-critical graphs with $\Theta(n^3)$ triangles, we found one example of such a family in a paper by
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

Plesník; see [12]. However, this family appears to have this characteristic as a coincidental emergent property, since the family was constructed for a different (although related) purpose, which we address briefly in Section 2.1.1. Furthermore, Plesník made no mention of this additional property in his paper. Therefore, we hope that the examples of 2-critical graphs with $\Theta(n^3)$ triangles that we present, and our notes and observations on their construction and properties, may provide some useful insights into possible methods for proving (or disproving) the conjecture.

Of course, Füredi's result guarantees that there are no infinite families of graphs which violate the conjecture. Thus, the only possible counterexample to the conjecture would consist of a finite set of 2-critical graphs with $> \frac{n^2}{4}$ edges, rather than a family of such graphs. Nevertheless, we develop methods to construct families of 2-critical graphs with $\Theta(n^3)$ triangles because we hope that examining such graphs, and determining why they do not have enough edges to violate the conjecture, will shed some light on the question of whether it is possible to construct any graph which violates the conjecture.

2.1.1 2-Critical Graphs with All Edges in Triangles

It is possible to construct families of 2-critical graphs which have the property that each edge is in at least one triangle in the graph. Plesník [12] presented several such families, and designated them 2-MT graphs. We will show by means of examples that although 2-MT graphs might appear to be related to graphs with $\Theta(n^3)$ triangles, neither property is either a sufficient or a necessary condition for the other.

The family of graphs $A(r)$ depicted in Figure 2.3 on page 15 clearly has $O(n)$ triangles; by inspection, $|\Delta(A(r))| = r + 6$.

Figure 2.4 on page 16 shows a member of the graph family $B(s, t)$, where $s = deg(u) - 1$ and $t$ is the number of children of $v_i$ ($i > 0$; $v_0$ has $t + 1$ children). Using a proof similar
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

![Figure 2.3: The family of graphs $A(r)$](image)

to the one that we will develop in the next section, we can show that $B(s, t)$ indeed has $\Theta(n^3)$ triangles (where $n$, the number of vertices in $B(s, t)$, is $(s+1)(t)+s+2$). However, [12] did not contain a proof that $|\Delta(B(s, t))|$ is in $\Theta(n^3)$, nor did it discuss the number of triangles in any family of graphs.

At any rate, it is clear that while the classes of 2-MT graphs and 2-critical graphs with $\Theta(n^3)$ triangles are not disjoint, neither are they strongly related. The 2-MT property is neither necessary nor sufficient to yield a graph with $\Theta(n^3)$ triangles.

### 2.2 Examples of 2-Critical Graphs with $\Theta(n^3)$ Triangles

Proposition 2.4 showed that Conjecture 2.3 would be proven to be true if there did not exist any graphs such that $|S_2|$ was in $\Theta(n^3)$. Each element of $S_2$ is a triangle with a specific property, as we discussed in Section 2.1. Thus, an obvious step towards proving (or disproving) the conjecture is to construct families of 2-critical graphs where $|\Delta|$ is in $\Theta(n^3)$, and examine their properties. We have constructed and will present three
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

We explored two approaches to constructing graphs with large numbers of triangles. The first approach involved using greedy heuristics in an attempt to find 2-critical subgraphs of $K_n$ with many triangles; the second approach was based on the same greedy heuristics, starting with graphs that had triangles which were guaranteed to exist in any 2-critical spanning subgraph.

The first approach is described by the algorithm below.

**Input:** The complete graph $K_n$.

**Output:** A 2-critical graph $G_c \subseteq K_n$. 

```
graph 2_critical_triangular(graph G)
{
    graph Gc;
    edge e, e_T;
    vertex_set V;
    edge_set E, E_c;
```
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

integer $T_G, T_e$;
boolean critical;

$V \leftarrow$ vertices($G$);
$E \leftarrow$ edges($G$);
$E_c \leftarrow E$;
$T_G \leftarrow 0$;
critical $\leftarrow$ FALSE;

while (critical = FALSE) do
{
    $G_c \leftarrow$ define_graph($V, E_c$);
    $e_T \leftarrow \emptyset$;
    for each $e \in E$ do
        if (diam($G_c - e$) $\leq$ 2) 
        {
            $T_e \leftarrow$ num_triangles($G_c - e$);
            if ($T_e \geq T_G$) 
            {
                $e_T \leftarrow e$;
                $T_G \leftarrow T_e$;
            }
        }
    if ($e_T \neq \emptyset$)
        $E_c \leftarrow E_c - e_T$;
    else
        critical $\leftarrow$ TRUE;
}
return $G_c$;

The heuristic can be changed from "maximize triangles" ("at each stage, find the non-critical edge whose removal induces a subgraph with the maximum number of triangles") to "minimize triangles" by replacing the ($T_e \geq T_G$) comparison with ($T_e \leq T_G$).

It turns out that both the maximizing and minimizing heuristics yield (apparently due to some cosmic graph-theoretic joke, as far as I can tell) graphs with no triangles. The first heuristic yields $K_{1,n-1}$; the second yields $K_{m,n-m}$ where $m = 2^k, k = \max_k \left\{ k : \frac{n}{3} \leq \right\}$.
The second approach is this: build a graph around a triangle \( \tau \) in such a fashion that each edge of \( \tau \) is critical. Add as many other edges to the graph as possible, subject to the restriction that the edges of \( \tau \) remain critical. Using the resultant graph as a starting point, apply one of the greedy heuristics described in Approach 1. Figure 2.5 shows the basic idea behind such a graph. If we define the vertex sets \( A = \{a_i\} \) and \( B = \{b_i\} \), then in order for this graph to be 2-critical, we must have \( A \cap B = \emptyset \), but that some of the vertices of \( A \) may coincide. (Note that this graph has diameter > 2; this is a conceptual framework on which a 2-critical graph is to be constructed, rather than the graph itself.)

![Diagram of 2-critical graph](image)

Figure 2.5: Original concept for 2-critical graph with critical triangle \((x, y, z)\)
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

2.2.1 The Graph Family $G_p$

After some experimentation with this basic structure, we were able to construct a family $G_p$ of 2-critical graphs. The first four members of it are shown in Figure 2.6 on page 30 and Figure 2.7 on page 31.

The properties of this family are as follows:

- The vertices divide naturally into three disjoint subsets $A$, $B$, $C$. The three outermost vertices comprise subset $A = \{a_1, a_2, a_3\}$; the three vertices of the middle "layer" comprise subset $B = \{b_1, b_2, b_3\}$, and the remaining vertices comprise subset $C$. (Note that subset $C$ includes the original triangle $\{(x, y, z)\}$ of our template). Subset $C$ is further divided into three subsets $C_1 = \{c_{11}(=x), c_{12}, \ldots, c_{1p}\}$, $C_2 = \{c_{21}(=y), c_{22}, \ldots, c_{2p}\}$, and $C_3 = \{c_{31}(=z), c_{32}, \ldots, c_{3p}\}$.

- The edges also divide naturally into four disjoint subsets $E_1$, $E_2$, $E_3$, $E_4$.

  - $E_1 = \{(a_i, b_j) : a_i \in A, b_j \in B\}$
  - $E_2 = \{(b_i, c_{ij}) : b_i \in B, c_{ij} \in C_i, C_i \subset C\}$
  - $E_3 = \{(c_{ik}, c_{jk}) : c_{ik} \in C_i, c_{jk} \in C_j, j \neq i\}$
  - $E_4 = \{(c_{ij}, c_{ik}) : i \neq 1, j \neq k\}$

Given $G_p$, the inductive step that creates $G_{p+1}$ consists of adding one vertex to each of the vertex subsets $C_i$. The new vertices are connected to the vertices of $G_p$ via new edges in $E_2$, $E_3$, and $E_4$.

Thus, a member $G_p$ of this family of graphs can be constructed as follows: we define $G_0$ to be the graph in this family that has no vertices in $C$ and consequently no edges in $E_2$, $E_3$, or $E_4$. Starting with $G_0$, attach $p$ vertices to each vertex of $B$. Connect $c_{ij}$ to $c_{kj}$ $\forall k \neq i$. Finally, connect the vertices of $C_2$ and of $C_3$ so as to form two copies of $K_p$. 

19
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

To summarize, a member $G_p$ of this family of graphs has the following properties:

1. $\forall a_i \in A : N(a_i) = B$ via $E_1$
2. $\forall b_i \in B : N(b_i) = A \cup C_i$ via $E_2$
3. $\forall j$ such that $c_{ij} \in C_i : c_{1j} \sim c_{2j}, c_{1j} \sim c_{3j}, c_{2j} \sim c_{3j}$ via $E_3$
4. $\forall j, k$ such that $c_{ij}, c_{ik} \in C_i, j \neq k, i \neq 1 : c_{ij} \sim c_{ik}$ via $E_4$
5. $|A| = |B| = 3, |C_i| = p, |V(G_p)| = 3p + 6$

Proposition 2.5 $diameter(G_p) = 2, \forall p \in \mathbb{N}$.

Proof: We prove this proposition by examining the shortest path lengths $d_{G_p}(u, v)$ between vertices $u$ and $v$ in all pairs of vertex subsets. We will make use of the properties of $G_p$ listed above.

- **Case 1:** $u \in A, v \in A$.
  
  By property 1, $\exists b_i \in B$ such that $u \sim b_i$ and $v \sim b_i$; thus there exists a path $u, b_i, v$ of length 2.

- **Case 2:** $u \in A, v \in B$.
  
  By property 1, $u \sim v$.

- **Case 3:** $u \in A, v \in C_i$.
  
  By property 1, $u \sim b_i$; similarly, by property 2, $v \sim b_i$. Thus, there exists a path $u, b_i, v$ of length 2.

- **Case 4:** $u = b_i, v \in C_i$.
  
  By property 2, $u \sim v$. 

20
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

- **Case 5**: $u = b_i, v \in C_j, j \neq i$.

  Let $v = c_{jk}$. By property 2, $u \sim c_{ik}$; by property 3, $v \sim c_{ik}$. Thus, there exists a path $u, c_{ik}, v$ of length 2.

- **Case 6**: $u \in C_i, v \in C_i$.

  By property 2, $u \sim b_i$ and $v \sim b_i$. Thus, there exists a path $u, b_i, v$ of length 2.

- **Case 7**: $u = c_{ik}, v = c_{jk}, i \neq j$.

  By property 3, $u \sim v$.

- **Case 8**: $u = c_{ik}, v = c_{jl}, i \neq j, k \neq l$.

  By property 3, $u \sim c_{jk}$; similarly, by property 4, $v \sim c_{jk}$. Thus, there exists a path $u, c_{jk}, v$ of length 2.

**Proposition 2.6** $G_p$ is 2-critical $\forall p \in \mathbb{N}$.

**Proof**: Since we have already demonstrated that $G_p$ has diameter 2, we need only demonstrate that the edges in each of $E_1$, $E_2$, $E_3$, $E_4$ are 2-critical. We will again make use of the properties of $G_p$ listed above.

- **Case 1**: $(u, v) \in E_1$.

  Without loss of generality, let $u = a_i$ and $v = b_j$. By property 1, $u$ is adjacent only to vertices in $B$. By properties 2 and 3, $v$ is not adjacent to any vertex in $B$. Thus, $(u, v)$ lies on the unique path of length $\leq 2$ connecting $u$ to $v$ and is therefore 2-critical.

- **Case 2**: $(u, v) \in E_2$.

  Without loss of generality, let $u = b_i$ and $v = c_{ij}$, and select an arbitrary vertex $a_k \in A$. By property 1, $a_k$ is adjacent only to vertices in $B$, so any path of length
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

$\leq 2$ from $a_k$ to $v$ must pass through some vertex $b_l \in B$. In particular, any such path must include a path of length 1 from $b_l$ to $v$. By properties 2 and 3, the only vertex in $B$ that is adjacent to $v$ is $u$. Thus, $(u, v)$ lies on the unique path of length $\leq 2$ connecting $a_k$ to $v$, and is therefore 2-critical.

- **Case 3**: $(u, v) \in E_3$.

Without loss of generality, let $u = c_{ik}$ and $v = c_{jk}$. By properties 2 and 3, the vertex $b_i$ is adjacent only to vertices in $A$ and $C_i$. By property 1, vertices in $A$ are adjacent only to vertices in $B$. Thus, any path of length $\leq 2$ from $b_i$ to any vertex in $C$ must pass through a vertex $c_{il} \in C_i$. By property 4, the only vertex in $C_i$ that is adjacent to $v$ is $u$. Thus, $(u, v)$ lies on the unique path of length $\leq 2$ connecting $b_i$ to $v$, and is therefore 2-critical.

- **Case 4**: $(u, v) \in E_4$.

Without loss of generality, let $u = c_{ij}$ and $v = c_{ik}$, $i \neq 1$. By properties 3 and 4, the vertex $c_{ij}$ is adjacent to $\{b_1, u, c_{lj}(l \neq i)\}$. By properties 2 and 3, $b_1$ is not adjacent to any vertices in $C_i(i \neq 1)$, so there is no path of length $\leq 2$ from $c_{ij}$ to $v$ passing through $b_1$. Similarly, by properties 3 and 4, $c_{ij}$ is adjacent to only one vertex in $C_i(u)$ so there is no path of length $\leq 2$ from $c_{ij}$ to $v$ passing through $c_{ij}$. Thus, $(u, v)$ lies on the unique path of length $\leq 2$ connecting $c_{ij}$ to $v$, and is therefore 2-critical.

Now that we have established that $G_p$ is 2-critical, we will calculate $|\Delta(G_p)|$ and $m = |E(G_p)|$.

By property 3 of $G_p$, the edges in $E_3$ form $p$ triangles. By property 4 of $G_p$, the edges in $E_4$, in conjunction with the edges in $E_2$ that connect $b_2$ to the vertices of $C_2$ and $b_3$ to the vertices of $C_3$, form 2 cliques of order $p + 1$. There are no other triangles in $G_p$. Thus we have
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

$$|\Delta(G_p)| = p + 2\binom{p+1}{3}.$$  

By property 5 of $G_p$,

$$|V(G_p)| = n = 3p + 6 \Rightarrow p = \frac{n - 6}{3}.$$  

This gives us a formula for $|\Delta(G_p)|$ in terms of $n$, which we can simplify to:

$$|\Delta(G_p)| = \frac{n^3 - 18n^2 + 126n - 324}{81}.$$  

Note that we have established that there exists a family of 2-critical graphs $G_p$ with $\Theta(n^3)$ total triangles, not that $|S_2(G_p)| \in \Theta(n^3)$, so this result is not sufficient to disprove the conjecture. In fact, $m < n^2/4$ for $G_p$, as we will now demonstrate.

Since $E = \{E_1, E_2, E_3, E_4\}$, we can calculate the size of each $E_i$ individually and then calculate $\sum |E_i| = m$. By properties 1 and 5 of $G_p$, $|E_1| = 9$. By properties 2 and 3 of $G_p$, $|E_2| = |E_3| = 3p$. Finally, by property 4 of $G_p$, $E_4$ is comprised of 2 cliques of order $p$, so $|E_4| = 2\binom{p}{2} = p^2 - p$. Thus we have

$$m = 9 + 3p + 3p + p^2 - p = p^2 - 5p + 9.$$  

As above, by property 5 of $G_p$, $p = \frac{n - 6}{3}$. This gives us the formula for $m$ in terms of $n$:

$$m = \frac{n^2 + 3n + 27}{9}.$$  

2.2.2 The Graph Families $\mathcal{H}_p$ and $\mathcal{W}_p$

The central idea behind constructing the family $G_p$ turned out to be the idea of replicating triangles in a graph in such a fashion that for each replicated triangle, two of the vertices.
must form distinct cliques with all other similar vertices. Since cliques of order $\Theta(n)$ have $\Theta(n^3)$ triangles, and we added a constant number of vertices to cliques with each increase in the order of $G_p$, the number of triangles in $G_p$ must be $\Theta(n^3)$.

The recognition of this pattern led to the speculation that we could use this heuristic to construct additional families of 2-critical graphs with $\Theta(n^3)$ triangles. We were successful in this, and as a result we were able to construct the families $\mathcal{H}_p$ and $\mathcal{W}_p$. We will not present the properties of these families in detail as we did for $G_p$, but we do make note of $|\Delta|$ and $|E|$ for each family.

The graphs in Figure 2.8 on page 32 are the first three elements of the family $\mathcal{H}_p$ of 2-critical graphs.

- $\mathcal{H}_p$ has three disjoint sets of triangles: the replicated triangles, a clique of order $p$ and another clique of order $p + 2$.
- $|\Delta(\mathcal{H}_p)| = \frac{n^3 - 9n^2 + 72n - 162}{81}$
- $|E(\mathcal{H}_p)| = \frac{n^2 + 9n - 18}{9}$

The graphs in Figure 2.9 on page 33 are the first three elements of the family $\mathcal{W}_p$ of 2-critical graphs.

- $\mathcal{W}_p$ has three disjoint sets of triangles: the replicated triangles and two cliques of order $p + 1$.
- $|\Delta(\mathcal{W}_p)| = \frac{n^3 - 12n^2 + 66n - 136}{81}$
- $|E(\mathcal{W}_p)| = \frac{n^2 + 7n - 17}{9}$

We define a triangle $\tau \in G$ to be vertex-isolated if it has no vertices in common with any other triangle in $G$, and edge-isolated if it has no edges in common with any other triangle in $G$. Note that vertex-isolation implies edge-isolation.
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

We observe a few facts about the families of graphs that we have generated:

- In each case, $|\Delta| = \frac{n^3}{81} + o(n^3)$ and $m = |E| = \frac{n^2}{9} + o(n^2)$.

- The replicated triangle in $G_1$ and $W_1$ is vertex-isolated; the replicated triangle in $H_p$ is only edge-isolated.

Our experiments in generating these families suggest that in order for a triangle in a 2-critical graph to act as a “seed”—a triangle whose replication generates a family of 2-critical graphs with $\Theta(n^3)$ triangles—it must be edge-isolated. Attempts to use a non-edge-isolated triangle as a seed resulted in graph families that did not exhibit the clique growth that we see in our examples, and that did not have a large number of triangles or edges.

“Seed” triangles which are edge-isolated but not vertex-isolated (such as $H_p$) seem to result in an asymmetry in the sizes of the generated cliques. We denote a vertex which is shared by several triangles in the original graph (including the “seed”) as $v_s$, and observe that the asymmetry is apparently due to the necessity of connecting the replicated copies of $v_s$ to each vertex of the triangles of which $v_s$ was a part in the original graph. These connections cause the clique including $v_s$ to be larger (by a constant number of vertices) than the clique whose members are the copies of a vertex which is part of only one triangle in the original graph. However, since the difference in the clique sizes is only a constant, the presence or absence of the property of vertex-isolation in the “seed” does not appear to affect the asymptotic number of triangles or edges of a graph family.

2.3 Analysis of our Examples

We have presented several families of 2-critical graphs with $\Theta(n^3)$ triangles; nevertheless, none of them have had enough edges to provide a counterexample to the conjecture. In
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

the following two sections, we will explore in detail the reasons why this is the case here, and why it may be not be possible to construct such a counterexample.

2.3.1 Associated Triangles

Recall that Proposition 2.4 states that a counterexample to the conjecture would not only have $\Theta(n^3)$ triangles, but would have $\Theta(n^3)$ members of $S_2$ (see page 11 for the definition of $S_2$ and related terms).

The families of graphs that we have constructed, as well as Plesnik’s $B(s, t)$ in [12], have all originated in the replication of some appropriate triangle in the original 2-critical graph, and at the concomitant growth of cliques whose vertices are comprised, in part, of the similar copies of the vertices of the replicated triangles. Thus, each vertex of one of these cliques is incident to at least two other vertices not in the clique: the vertices of the replicated triangle.

As an example, consider, without loss of generality, the triangle $(c_{21}, c_{22}, c_{23})$ in $G_4$ (whose picture is in Figure 2.7 on page 31). As we demonstrated in the proof that $G_p$ is 2-critical, the edge $c_{22}c_{21}$ lies on the unique path of length $\leq 2$ from $c_{12}$ to $c_{21}$. Since $c_{12} \not\sim c_{21}$ and $N(c_{12}) \cap N(c_{21}) = \{c_{22}\}$, we know that $(c_{12}, c_{21}, c_{22})$ is an extremal triple. Furthermore, since $N(c_{12}) \cap N(c_{21}) \cap N(c_{23}) = \{c_{22}\}$, we know that $(c_{12}, c_{22}, c_{23}) \in S_1$. Thus, the triangle $(c_{21}, c_{22}, c_{23})$ is associated with $(c_{12}, c_{22}, c_{23})$. Using similar arguments, we can show that $(c_{21}, c_{22}, c_{23})$ is also associated with $(c_{32}, c_{23}, c_{21})$, $(c_{33}, c_{21}, c_{22})$, and $(c_{13}, c_{21}, c_{22})$, which are also members of $S_1$. Thus, $(c_{21}, c_{22}, c_{23})$ is associated with at least four members of $S_1$, and is therefore clearly not a member of $S_2$.

This argument applies to any triangle in any of the cliques in the examples of 2-critical graphs with $\Theta(n^3)$ triangles that we have seen. Since the number of triangles in these families of graphs that are not in the cliques is clearly in $O(n^{3-\epsilon})$, none of these families
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

are candidates for providing a counterexample to the conjecture. Furthermore, since the existence of the cliques was made possible by the replication of the attached triangles, it seems unlikely that one can construct a graph which has enough triangles to violate the conjecture without "disqualifying" most of the triangles by forcing them to be associated with too many members of $S_1$.

2.3.2 Cliques, Triangles, and Edges

Recall that Conjecture 2.3 states that the number of edges in a 2-critical graph is $\leq \frac{n^2}{4}$, and that the only 2-critical graph with $\frac{n^2}{4}$ edges is $K_{[n/2],[n/2]}$.

We denote an arbitrary family of 2-critical graphs among those that we have discussed as $X_p$. For any $X_i$, then, we construct $X_{i+1}$ by adding a triangle $\tau$ to $X_i$. Of course, some additional edges beyond those of $\tau$ are also used to connect $\tau$ to $X_i$ and to insure that $X_{i+1}$ has diameter 2, but we wish to emphasize that three vertices are added for every increase in $p$ of 1. By contrast, if we set $p = n/2$, then for $K_{[p],[p]}$, we know that $p$ increases by 1 for every two vertices that we add.

Furthermore, for every such $\tau$ added to $X_p$, two vertices of $\tau$ contribute to disjoint cliques of order $\Theta(p)$ in $X_p$, and thus these vertices have degree $\Theta(p)$. We define the set $E_K$ to be the set of edges in $X_p$ which are part of one of these cliques, and note that the remaining vertex in $\tau$ invariably has degree 3. Thus, asymptotically, we can characterize $|E(X_p)|$ by considering only $|E_K(X_p)|$; it seems clear that $|E_K(X_p)| \in \Theta(|E(X_p)|)$. Therefore, we have the following:

$$|E(X_i)| \geq \sum_j \left( \frac{q_j n}{b} \right)$$

where $b$ is the number of vertices in the graph which we add to $X_i$ in order to form $X_{i+1}$ ($\tau$ is an example of such a graph), $q_j$ is the number of vertices which contribute to the
growth of clique \( j \), and \( n \) is, as usual, \( |V(\mathcal{X}_j)| \). (Note that in general, \( \sum_j q_j \neq b \).)

As we explore the use of clique-growing in the construction of families of 2-critical graphs which might provide counterexamples to the conjecture, we must consider several possible choices of \( q_j \) and \( b \):

- \( q_1 = 1, q_2 = 1, b = 3 \). These are the values of \( q_j \) and \( b \) for all the examples that we have presented. In this case, \( |E(\mathcal{X}_p)| = \frac{n^2}{9} + o(n^2) \).

- \( q_1 = 1, q_2 = 1, q_3 = 1, b = 3 \). In this case, \( |E(\mathcal{X}_p)| = \frac{n^2}{6} + o(n^2) \).

- \( q_1 = 1, q_2 = 1, b = 2 \). In this case, \( |E(\mathcal{X}_p)| = \frac{n^2}{4} + o(n^2) \). Of course, in this case we are no longer adding triangles but 2-cliques (two vertices connected by an edge).

- \( q_1 = 2, b = 3 \). In this case, two vertices of three contribute to a single clique, and \( |E(\mathcal{X}_p)| = \frac{2n^2}{9} + o(n^2) \).

None of these cases have enough edges to violate the conjecture, with the possible exception of the third (since we might asymptotically approach \( \frac{n^2}{4} \) from above). However, since \( |E(K_{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor})| = \frac{n^2}{4} \), we suspect that any 2-critical graph whose number of edges is \( \frac{n^2}{4} + o(n^2) \) may be isomorphic to \( K_{\lfloor n/2 \rfloor, \lfloor n/2 \rfloor} \).

- \( q_1 = 2, q_2 = 1, b = 3 \). In this case, \( |E(\mathcal{X}_p)| = \frac{5n^2}{18} + o(n^2) \).

- \( q_1 = b \). In this case, \( |E(\mathcal{X}_p)| = \frac{n^2}{2} + o(n^2) \) for any value of \( b \).

As a result of these observations, we make the following conjecture:

**Conjecture 2.7** Suppose that \( \mathcal{X}_p \) is a family of 2-critical graphs such that given \( \mathcal{X}_i \), we generate \( \mathcal{X}_{i+1} \) by adding a graph \( B = (V_B, E_B) \) to \( \mathcal{X}_i \) in such a way that some \( S \subseteq B \) combines with existing cliques in \( \mathcal{X}_i \) to form cliques in \( \mathcal{X}_{i+1} \). Then \( |E(\mathcal{X}_p)| > \frac{n^2}{4} \) only if \( \frac{|V_B|}{2} + \varepsilon \) vertices of \( B \) become part of a single clique in \( \mathcal{X}_{i+1} \) for any \( \varepsilon > 0 \).
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

This conjecture, if true, might seem to suggest that we should use techniques other than clique-growing in our attempts to construct a 2-critical graph with $> \frac{n^2}{4}$ edges. However, recall that Proposition 2.4 requires that any such graph must have $\Theta(n^3)$ triangles, and thus must have structures in it that are not overly different from cliques.

In summary, we hope that our focus of attention on certain well-defined issues arising from Conjecture 2.3 (and its relation to Proposition 2.4) will make it easier for researchers to prove or disprove it.
Figure 2.6: $G_1, G_2$
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

Figure 2.7: $G_3, G_4$
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

Figure 2.8: $\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3$
Chapter 2. 2-Critical Graphs with $\Theta(n^3)$ Triangles

Figure 2.9: $W_1, W_2, W_3$
Chapter 3

*d*-Critical Spanning Subgraphs

This chapter is devoted to the discussion of methods for identifying *d*-critical spanning subgraphs. We will review some methods of calculating the diameter of a graph and of recognizing a *d*-critical graph. We will then present efficient algorithms for identifying 2-critical and 3-critical spanning subgraphs of graphs of diameters 2 and 3, respectively. Finally, we will discuss the problem of efficiently identifying a *d*-critical spanning subgraph of a graph of diameter *d* where *d* ≥ 4.

3.1 Calculating the Diameter of a Graph

Recall that we defined the diameter of a graph *G* to be the longest distance between any two vertices *i* and *j* in *G*, where the distance *d*_{*G*}(*i*, *j*) is itself defined as the length of the shortest *ij*-path in *G*. From this definition, we derive two methods of calculating the diameter of a graph.

3.1.1 Breadth-First Search Trees

The first method is quite straightforward: we make use of the fact that, for a breadth-first search (BFS) tree *T* generated from a graph *G* having diameter *d* and rooted at *v* ∈ *V*(*G*), the shortest paths from *v* in *T* are exactly the shortest paths from *v* in *G*. Thus, a BFS tree rooted at any vertex in a graph *G* must have height ≤ *d*, and the tallest
of these BFS trees will have height = \( d \).

Generating a BFS tree requires \( O(m) \) time, since each of the \( m \) edges in \( G \) is traversed once in a breadth-first search, and each traversal requires \( O(1) \) time. Since there are \( n \) vertices, the process of generating the BFS trees requires \( O(mn) \) time. Furthermore, we can keep track of the height of a BFS tree as we build it, and calculate the maximum height among all BFS trees, using no more than \( O(mn) \) additional time. Therefore, the total time required to calculate the diameter using this method is \( O(mn) \).

3.1.2 Matrix Multiplication

The ideas behind calculating the diameter of a graph using matrix multiplication are somewhat more involved. We define the adjacency matrix \( A \) of a graph \( G \) as follows:

\[
A(i, j) = \begin{cases} 1 & \text{if } e = (i, j) \in E \\ 0 & \text{otherwise} \end{cases}
\]

We further define \( A^k \) inductively as follows: \( A^1 = A \), and \( A^{k+1} = AA^k \). We denote the adjacency matrix \((A + I)^k\) as \( A_k \), and the \((i, j)\)th entry of \( A_k \) is denoted \( A_k(i, j) \). Finally, we say that \( A_k > 0 \) if every entry in \( A_k \) is \( > 0 \).

We denote the complexity of multiplying two \( n \times n \) matrices as \( M(n) \), since there are several algorithms for matrix multiplication, each with its own advantages. Coppersmith and Winograd [4] have established that \( M(n) \in O(n^{2.377}) \) for general matrices. However, we observe that an adjacency matrix is a Boolean matrix: all elements are in \( \{0, 1\} \) and the sum and product operations on rows and columns are equivalent to the (bitwise) logical OR and AND operators, respectively. There is a practical algorithm for multiplying Boolean matrices, known as the "four Russians" algorithm (due to Arlazarov, et al. [2]), whose refinement, described in [9], has a time complexity of \( O(n^3 / \lg^2 n) \). In practice, the
Chapter 3. \textit{d-Critical Spanning Subgraphs}

refined “four Russians” algorithm tends to be more efficient on Boolean matrices than Coppersmith and Winograd’s algorithm does.

\textbf{Proposition 3.1} \(A_k(i, j) > 0\) only if there is a path from \(i\) to \(j\) of \(k\) or fewer edges.

\textbf{Proof:} We will prove this result by induction on \(k\).

Base case: \(A_1 > 0\) if and only if \(i\) and \(j\) are connected by an edge, by the definition of an adjacency matrix.

Inductive step: We must show that if \(A_k(i, j) > 0\) exactly when there exists an \(ij\)-path of length \(\leq k\), then \(A_{k+1}(i, j) > 0\) exactly when there exists an \(ij\)-path of length \(\leq k + 1\).

\(A_{k+1}(i, j) > 0\) exactly when there is at least one nonzero element \(A_k(i, l)\) and a corresponding nonzero element \(A_1(l, j)\), by the definition of matrix multiplication on nonnegative matrices. Furthermore, by definition, if \(A_1(l, j) > 0\), then \(l\) is connected to \(j\), and by inductive assumption, if \(A_k(i, l) > 0\), then there exists an \(il\)-path of length \(\leq k\). Thus, if \(A_1(l, j) > 0\) and \(A_k(i, l) > 0\), then there exists a path \((i, \ldots, l, j)\) of length \(\leq k + 1\).

\textbf{Proposition 3.2} \(A_{k+1} \geq A_k\).

\textbf{Proof:} This follows directly from Proposition 3.1. If there exists an \(ij\)-path of \(\leq k\) steps, there is certainly an \(ij\)-path of \(\leq k + 1\) steps, so \(A_{k+1}\) is bounded from below by \(A_k\). Furthermore, there may exist an \(ij\)-path of \(k + 1\) steps where there exists no \(ij\)-path of \(k\) steps, so \(A_{k+1}\) may be greater than \(A_k\).

\textbf{Lemma 3.3} \(\text{diam}(G)\) is the smallest \(k\) for which \(A_k > 0\).

\textbf{Proof:} \(A_k > 0\) means that there is a path of length \(\leq k\) between each pair of vertices.
Chapter 3. d-Critical Spanning Subgraphs

$i, j \in V$; or, in other words, that the distance between any pair of vertices in $V$ is bounded above by $k$.

Suppose that $k$ was the smallest power of $(A+I)$ such that $A_k > 0$, but $diam(G) = d < k$. This would imply that the distance between any pair of vertices was $\leq d$, but by this assumption, $A_d$ has some 0 entries, so there is some pair of vertices for which the shortest path is longer than $d \implies$ contradiction.

If $A_k > 0$, we know that the distance between any pair of vertices in $V$ is bounded above by $k$, so the diameter, being the maximum of all distances in $G$, can be no more than $k$.

\[ \bullet \]

**Theorem 3.4** We can calculate the diameter of a graph $G$ in $O(M(n) \log d)$ time, where $d = diam(G)$.

**Proof:** Recall that $d = \min_k \{A_k > 0\}$. We generate powers of $A_1$ by repeated squaring:

\[ (A_1)^2 \rightarrow A_2, (A_2)^2 \rightarrow A_4, \ldots, (A_{2^i})^2 \rightarrow A_{2^{i+1}}, \] stopping at iteration $j - 1$, when $A_{2^j} > 0$. We store the matrices that we have generated for use later in the algorithm. The $A_{2^i} > 0$ test at each iteration requires $O(n^2)$ time, and there are $j - 1$ iterations. By the definition of $d$ given above, we know that $j = \lceil \log d \rceil$. Therefore, thus far we have executed $\log d$ iterations, each of which required $O(M(n) + n^2) = O(M(n))$ time, for a total of $O(M(n) \log d)$ time.

Again, by the definition of $d$, we know that $d$ lies in the interval $[2^{j-1}, 2^j]$, so we can use binary search to find $d$. We make use of the following facts:

- The size of the interval that we are searching is always a power of 2: initially the interval size is $2^{j-1}$, and thereafter, each iteration of the binary search cuts the interval exactly in half.
Chapter 3. d-Critical Spanning Subgraphs

- The matrix $A_d$ can be decomposed into a product of matrices of the form $(A_1)^{2^i}$. In particular, matrix $(A_1)^{2^i}$ contributes to this product if and only if the $i$th bit of $d$ is 1.

One can interpret the stages of the binary search as determining, starting from the most significant bit of $d$, whether the successive bits (in decreasing order of significance) are 0 or 1. We already know that the most significant bit of $d$ is 1, by definition.

These facts, in conjunction with the generation of the matrices of the form $(A_1)^{2^i}$ in the first part of the algorithm, allow us to achieve our desired bound of $O(lg d)$ matrix multiplications for the algorithm. A more naive approach would involve doing binary search on the interval $[1, n]$ (note that $d \leq n - 1$). However, if $n$ is not a power of 2, the midpoint of the current interval (represented by the matrix $A_k$) would be most efficiently generated by a process of repeated squaring followed by binary search. This is exactly what our algorithm does to determine $A_d$, but the naive algorithm would require such a calculation at each stage of the binary search for $A_d$. This would require $O(lg n)$ matrix multiplications at each stage of the binary search, and would yield an overall bound of $O(lg^2 n)$, rather than $O(lg d)$, matrix multiplications for the algorithm.

To make it easier to deal with the boundary cases, we search for the largest $i$ such that $A_i \neq 0$, rather than the smallest $k$ such that $A_k > 0$. This $i$ is of course $d - 1$, so we merely add 1 to the result of the binary search to yield the diameter. The search then proceeds as follows: we initialize $A_k$ to $A_{2^{i-1}}$, initialize $i$ to $j - 2$, and continue as long as $i \geq 0$.

This algorithm is formalized below in the function `diameter_bsearch()`. It assumes that the matrix set $A$ has already been generated by the repeated squaring process described above. The function `set_bit(d, j, n)` sets the $i$th bit of the integer $d$ to $n$, where $n \in \{0, 1\}$. 

38
Chapter 3. \(d\)-Critical Spanning Subgraphs

Input: A set of matrices \(A = \{A_{2^k} : k = 0, 1, \ldots, \lfloor \lg d \rfloor \} \).

Output: The diameter \(d\) of the graph from which \(A\) was generated.

```c
graph diameter_bsearch(matrix_set A, integer j)
{
    integer i, d;
    matrix B;
    B <- A_{2i-1};
    d <- 2^{i-1};

    for i from j - 2 downto 0 do
    {
        if (BA_{2i} > 0)
        {
            B <- BA_{2i};
            set_bit(d, i, 1);
        }
    }
    return d + 1;
}
```

This is not exactly the usual form of a binary search routine, so perhaps some explanation is in order. A binary search routine proceeds by determining, at each stage, whether the search object is above or below the midpoint of the current interval. If above, the new interval is bounded by the midpoint and the upper bound of the current interval; if below, the new interval is bounded by the lower bound and midpoint of the current interval. Recall that we are looking for the largest \(i\) such that \(A_i \neq 0\). The matrix \(B\) is simply the current lower bound for this \(A_i\) in the search. The index \(i\) specifies the midpoint of the search interval by indicating the matrix \(A_{2i}\) which we will multiply by \(B\) to see whether the matrix product is \(> 0\). We do not need to represent the upper bound of the search interval explicitly.

There are \(j - 1\) iterations of this algorithm; we have already established that \(j = \lfloor \lg d \rfloor\).
Each iteration requires $O(1)$ matrix multiplications, $O(n^2)$ work to determine whether $BA_i > 0$, and one call to `set_bit()`, which we will assume requires $O(1)$ time. Thus, the binary search requires a total of $\lg d(M(n) + O(n^2)) = O(M(n) \lg d)$ time. Therefore, as the repeated squaring was also determined to require $O(M(n) \lg d)$ time, the algorithm as a whole requires $O(M(n) \lg d)$ time.

We observe that the BFS tree algorithm is asymptotically faster than the matrix multiplication algorithm for sparse graphs, and vice versa for dense graphs. As we mentioned earlier, the "four Russians" Boolean matrix multiplication algorithm described in [2] runs in $O(n^3/\lg^2 n)$ time. Thus, if $m = O(n^2/\lg^2 n)$ then we would use BFS trees to calculate the diameter, and if $m \in \Omega((n^2/\lg^2 n)^{1+\epsilon})$ then we would use matrix multiplications. Thus, we say that we can calculate the diameter of a graph in $O(\min\{mn, M(n) \lg d\})$ by determining the density of the graph (which requires $O(1)$ time) and choosing the appropriate algorithm. We note that this is an improvement over the bound given by Anstee and Caccetta in [1], who cited an $O(\min\{mn, M(n) \lg n\})$-time bound for diameter calculation.

3.2 Recognizing $d$-Critical Graphs

A naive approach to recognizing diameter-critical graphs consists of $m$ diameter computations: given a graph $G$ such that $\text{diam}(G) = d$, if $\exists e \in E$ such that $\text{diam}(G - e) > d$, then $G$ is diameter-critical. This algorithm runs in $O(\min\{m^2 n, m \cdot M(n) \lg d\})$ time.

In [1], Anstee and Caccetta developed several more efficient algorithms for recognizing diameter-critical graphs:

- An $O(M(n))$-time algorithm for recognizing 2-critical graphs.
- An $O(nm + M(n))$-time algorithm for recognizing 3-critical graphs.
Chapter 3. \textit{d-Critical Spanning Subgraphs}

- An $O(n^2m)$-time algorithm for recognizing $d$-critical graphs of any diameter $d$.

We are not aware of any algorithm which is more efficient than the general one given in [1] for graphs of diameter $\geq 4$.

These results are relevant in that they contribute to an understanding of the issues involved in the development of efficient algorithms for identifying $d$-critical spanning subgraphs. We will discuss this more fully in the next section.

3.3 Identifying $d$-Critical Spanning Subgraphs

We are interested in identifying, for a given graph $G$ with diameter $d$, spanning subgraphs of $G$ which are $d$-critical. We conjecture that the corresponding optimization problem, i.e., finding the $d$-critical spanning subgraph of $G$ which has minimum weight, is NP-hard; it is known that the following closely related problem is NP-hard.

\textbf{Theorem 3.5 (Plesnik [11])} The following problem is NP-hard: given an edge-weighted graph $G$ with cost function $c : E \rightarrow \mathbb{R}$ and an integer $B$, find a spanning subgraph $F$ of $G$ with cost $\sum_{e \in E(F)} c(e) \leq B$ and with $\text{diam}(F)$ minimized.

We consider instead finding efficient solutions to this problem:

\textbf{$d$-Critical Spanning Subgraph Problem}: Given a graph $G$ such that $\text{diam}(G) = d$, find a spanning subgraph $G_c$ of $G$ such that $G_c$ is minimal with respect to the operation of edge removal and has the property $\text{diam}(G_c) = d$.

The following algorithm provides a straightforward (if somewhat naive) general solution to this problem for any value of $d$.

\textbf{Input}: A graph $G = (V, E)$ having diameter $d$.

\textbf{Output}: A spanning $d$-critical graph $G_c \subseteq G$. 

41
Chapter 3. d-Critical Spanning Subgraphs

graph naive_critical_subgraph(graph $G$, integer $d$)
{
    graph $G_c, G'$;
    edge $e$;
    vertex_set $V$;
    edge_set $E, E_c$;

    $V \leftarrow \text{vertices}(G)$;
    $E \leftarrow \text{edges}(G)$;
    $E_c \leftarrow E$;
    for each $e \in E$ do
    {
        $G' \leftarrow \text{define_graph}(V, E_c - e)$;
        if (diam($G'$) $\leq d$)
            $E_c \leftarrow E_c - e$;
    }
    $G_c \leftarrow \text{define_graph}(V, E_c)$;
    return $G_c$;
}

Since each edge is considered exactly once, and the graph considered in each call to the diameter computation routine is a subgraph of $G$, the complexity of this algorithm is that of $m$ diameter computations. Thus, we have an algorithm which runs in time $O(\min\{m^2n, m \cdot \lg d \cdot M(n)\})$. (Note that this is exactly the complexity of the naive approach to diameter-critical graph recognition.)

It is interesting to observe that, for graphs of diameter 2 or 3, we can achieve essentially the same time bounds for the $d$-critical spanning subgraph problem by using the $d$-critical graph recognition algorithms described in [1]. (Note that, for fixed $d$, $\text{naive_critical_subgraph}()$, as implemented above, runs in time $O(\min\{m^2n, m \cdot \lg d \cdot M(n)\})$.) Specifically, we replace, in $\text{naive_critical_subgraph}()$, the calculation and test of the diameter (the call to diam($G'$) and comparison to $d$) with a call to the appropriate recognition algorithm: $\text{is_2_critical}(G')$ or $\text{is_3_critical}(G')$. For graphs of diameter 2, this yields an algorithm which runs in $O(m \cdot M(n))$ time; for graphs of
Chapter 3. d-Critical Spanning Subgraphs

diameter 3, the resultant algorithm runs in \(O(m^2n + m \cdot M(n))\) time.

As with the recognition problem, we suppose that it is possible to construct an algorithm that is more efficient than these naive approaches. Neither calculating the diameter, nor testing the graph for diameter-criticality, after the removal of each edge seems terribly efficient. While we recognize that diameter is a global property of a graph, we would like to be able to eliminate some of the redundancy in the \(m\) diameter computations.

It turns out that we can indeed improve the efficiency of the naive algorithm, at least for graphs of diameter 2 or 3. In order to make improvements on the bounds thus far achieved, we will retain the original call to \(\text{diam}(G')\), and alter the function called so that it computes the diameter using BFS trees, and updates the BFS trees efficiently, as necessary, when an edge is deleted from the graph. We make use of the fact that we established earlier: that the diameter of \(G\) is exactly equal to the height of the tallest BFS tree generated from the vertices of \(G\). Formally, then, for a given graph \(G\), we construct a set \(\mathcal{T}\) of the \(n\) BFS trees rooted at the vertices of \(G\), and designate \(T_v\) to be the BFS tree rooted at vertex \(v\). Then we have

\[
diam(G) = \max\{\text{height}(T_v) : T_v \in \mathcal{T}\}
\]

This fact implies that, if we can efficiently update the BFS trees as edges of \(G\) are deleted, we can speed up the naive algorithm for identifying a diameter critical spanning subgraph of \(G\). We attempt to remove each edge \(e\) from \(G\) and obtain the BFS trees \(\mathcal{T}\) for the graph \(G - e\). We then check to see whether any \(T_v \in \mathcal{T}\) has height greater than \(d\). If there is any such \(T_v\), then \(e\) is critical and is not removed. Otherwise, we remove \(e\) from \(G\) and update the BFS trees.

**Input:** A graph \(G = (V,E)\) having diameter \(d\).

**Output:** A spanning \(d\)-critical graph \(G_c \subseteq G\).
Chapter 3. $d$-Critical Spanning Subgraphs

graph critical_subgraph(graph $G$, integer $d$)
{
    graph $G_c$;
    vertex_set $V$;
    edge_set $E$, $E_c$;
    vertex $v$;
    edge $e$;
    bfs_tree_list $T,T'$;
    boolean critical;

    $V \leftarrow \text{vertices}(G)$;
    $E \leftarrow \text{edges}(G)$;
    $E_c \leftarrow E$;
    for each $v \in V$ do
        $T_v \leftarrow \text{create_BFS_tree}(V,E,v)$;
    for each $e \in E$ do
        critical $\leftarrow$ FALSE;
        for each $v \in V$ do
            while (critical = FALSE) do
            {
                $T'_v \leftarrow \text{remove_edge}(T_v,e)$;
                if (height($T'_v$) $\geq$ $d$)
                    critical $\leftarrow$ TRUE;
            }
        if (critical = FALSE)
        {
            $E_c \leftarrow E_c - e$;
            for each $v \in V$ do
                $T_v \leftarrow T'_v$;
        }
    $G_c \leftarrow \text{define_graph}(V,E_c)$;
    return $G_c$;
}
3.3.1 Properties of Breadth-First Search Trees

There are a few properties of BFS trees (beyond those which we have mentioned previously) which are worthy of mention here, as they have a direct bearing on the implementation of the restructuring routine, called here remove_edge().

- A BFS tree of a graph $G$ has only $n - 1$ edges, while the number of edges in $G$ may be $\Theta(n^2)$. Therefore, there may be as many as $\Theta(n^2)$ edges in $G$ that do not appear in a particular $T_v$. We refer to such edges as virtual edges for the tree $T_v$. Figure 3.1 on page 45 shows a BFS tree with its virtual edges; the BFS tree edges are solid, and the virtual edges are dotted.

- We denote the level of a vertex $k$ as $l(k)$, or, if it is not clear which BFS tree we are considering, $l(T_i, k)$, where the ordered pair $(T_i, k)$ designates vertex $k$ in BFS tree $T_i$. The level of a node is defined recursively as follows:

  - $l(T_i, i) = 0$.
  - Let $j$ be the parent of $k$ in $T_i$. Then $l(T_i, k) = l(T_i, j) + 1$. Thus $l(T_i, k)$ is the number of edges on the path in $T_i$ from the root (vertex $i$) to $k$; or, in other
words, the length of the shortest \(ik\)-path.

- Since the BFS tree \(T_k\) consists of the shortest path tree from \(k\) to all other vertices in \(G\), a vertex will never be repositioned nearer the root of its BFS tree as a result of the deletion of the edge connecting it to its parent. In fact, as we are considering only undirected graphs (so \((u, v) \in E \iff (v, u) \in E\)), there are only three types of virtual edges in these BFS trees which are incident to a vertex \(v\):

  1. Edges which connect \(v\) to a vertex of level \(l(v) - 1\). There are no such edges if \(v\) is a child of \(k\) in \(T_k\), i.e., \(v\) is a child of the root.

  2. Edges which connect \(v\) to a vertex of level \(l(v)\).

  3. Edges which connect \(v\) to a vertex of level \(l(v) + 1\).

Thus, for a given vertex \(v\) in a graph of diameter \(d\), \(l(v)\) will monotonically increase up to \(O(d)\) times over the course of the identification algorithm. This becomes important in our discussion of the algorithm for general \(d\).

Note that although virtual edges only connect \(v\) to vertices one level different from its own, it is possible for the deletion of an edge to result in an increase in \(l(v)\) of \(O(n)\). As an example, consider the graph consisting of a simple cycle. If we remove an edge \(e = (i, j)\) from this graph, the shortest distance between \(i\) and \(j\) increases from 1 to \(n - 1\).

We will show that the efficiency of this identification algorithm depends on the efficiency of the implementation of the \texttt{remove_edge()} function. In particular, we will show how to improve the performance of our algorithm by an efficient implementation of the \texttt{remove_edge()} function. If the BFS trees were simply rebuilt from scratch each time an edge was removed, the complexity of the algorithm would be in \(O(m^2n)\): \(O(m)\) time is necessary to generate a BFS tree, and there are \(n\) BFS trees to be (re)generated as each
of $m$ edges is tested for criticality. Note that this is, unsurprisingly, just the complexity we noted earlier for the naive approach. To do better, we must do as little restructuring of the BFS trees as possible. The following theorems will specify how this restructuring should proceed for graphs of diameters 2 or 3.

### 3.3.2 Algorithm for Diameter 2 Graphs

**Theorem 3.6** Let $G = (V, E)$ be a diameter 2 graph having $m$ edges and $n$ vertices. We can identify a diameter 2 critical spanning subgraph in $O(mn)$ time.

**Proof:** As above, we generate a set $T$ of the $n$ BFS trees rooted at the vertices of $G$, and designate $T_i$ to be the BFS tree rooted at vertex $i$. As the BFS trees are generated, we record the level of each vertex in each BFS tree. As we established earlier, the process of generating the BFS trees requires $O(mn)$ time.

For each $T_i$, we generate an adjacency list representation of the entire graph $G$: each vertex $(T_i, k)$ is assigned a list $adj(T_i, k)$ of the vertices adjacent to $k$. The crucial idea is that each adjacency list is divided into disjoint sublists by level; since the level of a vertex is stored at the vertex, and the ordering of vertices within a sublist is unimportant, only $O(1)$ time is needed to direct each vertex to its proper sublist. The combined size of all adjacency lists of $T_i$ is in $O(m)$, since vertex adjacency lists are simply an implicit listing of the edges of a graph, and each edge $e = (u, v)$ is represented twice: $v$ is a member of $adj(T_i, u)$, and $u$ is a member of $adj(T_i, v)$.

Generating the adjacency lists for all the BFS trees requires $O(mn)$ time: there are $n$ BFS trees, and $O(m)$ work is required to build the adjacency list for each BFS tree.

Once the BFS trees and adjacency lists have been constructed, we examine each edge $e = (i, j) \in G$ in conjunction with each $T_k \in T$ to see whether its removal will increase the height of any $T_k$. We create a temporary copy of each $T_k$, denoted $T'_k$, and, assuming
that $e$ is not a virtual edge for $T'_k$, remove $e$ from $T'_k$. $T'_k$ must be restructured so as to restore the BFS tree property, i.e., the vertices disconnected from $T'_k$ by the removal of $e$ must be reconnected to $T'_k$. If all of the restructured BFS trees have height $\leq d$, then $e$ is not critical, each $T'_v$ becomes the new $T_v$, and the algorithm continues on the graph $G - e$. Otherwise, the algorithm retains the current copies of $T_v$ and continues with the next edge in $G$, until all edges have been tested for criticality.

Thus far, we have simply described in detail the algorithm specified above, and demonstrated that the setup (generating the BFS trees and adjacency lists) requires $O(mn)$ time. In order to make the BFS tree restructuring which occurs in remove_edge() efficient, we will use a technique known as lazy deletion, in which the deletion of elements of a data structure is delayed to improve the efficiency of the deletion routine. If the presence or absence of the element during later stages of the algorithm’s execution is irrelevant, the deletion may be delayed indefinitely, or never occur.

We use lazy deletion so that we may avoid doing unnecessary updates to the BFS trees and their adjacency lists. We observe that, in the end, the only information about the BFS trees that we are interested in is their height, and as long as we know that the height of a restructured BFS tree is $\leq d$, that is sufficient. Thus we update the BFS trees and associated adjacency lists only when necessary; the details are given below. We also note that if we attempted to update all the adjacency lists of a BFS tree immediately in response to an edge deletion, this would require $O(m)$ time per BFS tree, leading to the $O(m^2n)$ time bound of the naive algorithm.

For each BFS tree, there are three cases:

- Case 1: $e \notin T_k$.

  By definition, $e$ is a virtual edge for $T_k$. Therefore, $T_k$ is the same for the graphs $G = (V, E)$ and $G' = (V, E - e)$. Thus, removing $e$ from $G$ has no immediate effect
Chapter 3. d-Critical Spanning Subgraphs

Figure 3.2: An example of case 2 for diameter 2: $T_i$

on $T_k$. In keeping with the principle of lazy deletion, $T_k$ is not updated at this time to reflect the deletion of $e$; update may take place at a later stage if necessary. (For details on when and how updates occur, see cases 2 and 3.)

• **Case 2:** $e = (i, j) \in T_i, T_j$.

Without loss of generality, consider the BFS tree $T_i$. Since $e = (i, j)$, $e$ is adjacent to the root in $T_i$. Thus, removing $e$ from the tree disconnects $O(D)$ vertices in the subtree $\tau_j$ of $T_i$ of which $j$ is the root. (Recall that $D$ is defined as being the maximum degree of any $v \in V$, and is therefore an upper bound on the size of $\tau_j$.) For each $v \in \tau_j$, search $adj(T_i, v)$ for an adjacent vertex $a$ in the level 1 adjacency sublist. Also, check the current level of $a$, since $a$ may have moved due to a previous restructuring of $T_i$.

- If $a = j$, ignore it (since $j$ will no longer be at level 1 once the restructuring of $T_i$ is done), and continue searching.
- If $a$ is still at level 1, attach $v$ to $a$ in $T_i$.
- If $a$ is no longer at level 1, remove it from the level 1 adjacency sublist, put it in the sublist appropriate to its new level, and continue searching.
- If the edge $(v, a)$ is no longer in $G$ (having been determined to be noncritical at an earlier stage, and then deleted), then remove $a$ from $adj(T_i, v)$, and
Chapter 3. \textit{d-Critical Spanning Subgraphs}

continue searching.

- If there is no such \( a \), \( e \) is 2-critical. Abort the restructuring of the BFS trees, and restore all \( T_i \in \mathcal{T} \) to their prior states using backup copies. We must also undo the changes made to the adjacency lists as a result of the assumed deletion of \( e \). The mechanism for this, and the reason for not simply reverting to the backup copies of the adjacency lists, are described in the proof of Claim 3.7.

The deletion of \( e \) requires each \( v \in \tau_j \) to be repositioned in \( T_i \). We define \( I(T_i, v, e) \) to be the number of invalid elements of \( \text{adj}(T_i, v) \) which are encountered in the search for an element of \( \text{adj}(T_i, v) \) to which \( v \) may be reattached. (Invalid elements are those which are skipped, moved to a different level sublist, or deleted according to the rules given above.) The time required to reattach each \( v \in \tau_j \), then, is \( O(1) \) for the reattachment + \( O(1) \) for each invalid element, or \( O(1 + I(T_i, v, e)) \). Thus, since there are \( O(D) \) vertices in \( \tau_j \), the total time required to reattach all vertices in \( \tau_j \) is

\[
O(D + \sum_{v \in \tau_j} I(T_i, v, e))
\]

Note that we are not searching all elements of \( T_i \) to see whether \( j \) is now incorrectly listed as being on level 1 in their adjacency lists. It is a characteristic of the lazy deletion to which we referred earlier that we update the adjacency lists only when we encounter members which have been invalidated in the course of a previous iteration of the algorithm.

- Case 3: \( e = (i, j) \in T_k, k \neq i, l(i) = 1, l(j) = 2. \)

Since \( j \) is at level 2 of \( T_k \), the subtree \( \tau_j \) which is disconnected from \( T_k \) by the removal of \( e \) must consist of exactly one vertex, \( j \). We use the procedure described in case 2 to reattach \( j \). Thus, the time required in this case is \( O(I(T_k, j, e)) \).
We observe that the work done in this algorithm can be separated into three parts for purposes of calculating the time complexity:

- creating the BFS trees and adjacency lists
- repositioning of vertices in BFS trees due to edge deletion
- searching for valid entries and disposing of invalid entries in adjacency lists, pursuant to vertex repositioning

We have already established that the setup for this algorithm requires $O(mn)$ time. We can easily demonstrate that the vertex repositioning requires a total of $O(mn)$ time (recall that repositioning a vertex requires $O(1)$ time, not including the time required to search for a valid adjacency list entry). Case 1 requires no vertex repositioning. Case 2, which can occur in two BFS trees for a given edge, requires the repositioning of $O(D)$ vertices. Case 3 can occur in $O(n)$ BFS trees for a given edge, and requires the repositioning of a single vertex. Thus we have a total of $O(mD + mn) = O(mn)$ vertices that are repositioned over the course of the algorithm, and therefore the vertex repositioning requires $O(mn)$ time.

In order to establish that the traversal of the adjacency lists also requires $O(mn)$ time, we use a technique known as amortized analysis. Amortized analysis is an analysis of
algorithm complexity that bounds the total cost of a series of operations, rather than attempting to bound the cost of a single operation. This is a useful technique when

- there is a wide variation in the costs of individual operations, or

- broadly speaking, if a particular operation is expensive, subsequent operations will be relatively cheap (and vice versa).

The time complexity of cases 2 and 3 depends on $I(T_k, v, e)$, the number of members of $\text{adj}(T_i, v)$ which are invalidated in the process of restructuring $T_k$ in response to the removal of edge $e$. In the worst case, $O(D)$ edges incident to $v$ have been removed from $G$ and $v$'s adjacency list has never yet been updated, so $I(T_k, v, e)$ is in $O(D)$. This is where the amortized analysis comes in: instead of looking at the individual $I(T_i, v, e)$, we consider the total number of obsolete adjacency list entries that we may encounter, measured over all $T_k \in \mathcal{T}$ and all edges $e \in E$.

**Claim 3.7**

$$
\sum_{e \in E} \sum_{T_k \in \mathcal{T}} \sum_{v \in T_j} I(T_k, v, e) = O(mn).
$$

**Proof:** This sum represents the number of invalid adjacency list entries that are encountered over the course of the execution of the algorithm. We will prove that the number of adjacency list entries that are rendered invalid by the algorithm is bounded by $O(mn)$. Thus, we must first justify the use of this value to bound the number of adjacency list entries that we encounter during the course of the algorithm.

We must demonstrate that an invalid adjacency list entry is encountered (at most) once. At first glance, this seems trivial; when we encounter an invalid adjacency list entry, we fix it in $O(1)$ time as described in case 2. However, if the candidate edge $e$ is found to be critical, we must restore the BFS trees to the state that they were in before we
Chapter 3. *d*-Critical Spanning Subgraphs

started testing \(e\) for criticality. We must do some restoration of the adjacency lists, as well, or there would be adjacency list entries that were improperly marked as invalid. However, if we undo all the changes made to the adjacency lists, then we may encounter a given invalid adjacency list entry \(O(m)\) times. For an example of this worst-case behavior, consider a graph with exactly one noncritical edge, which is deleted in the first iteration. The algorithm will then attempt to restructure the BFS trees in response to the deletion of each edge, and since all remaining edges are critical, it will revert to the previous copies of the adjacency lists after each iteration. Thus, each adjacency list entry which was invalidated in the first iteration may be encountered once for each succeeding (noncritical) edge that the algorithm examines.

Therefore, for each candidate edge \(e\), we keep a record of the changes that are made to the adjacency lists, and separate them into two disjoint categories: changes that are made in response to the tentative deletion of \(e\), and changes that were made in previous iterations (in response to the deletion of edges that have been proven to be noncritical). Using this information, we can undo the changes of the first category while preserving those of the second category. All changes made to the BFS trees fall into the first category, so we can simply revert the BFS trees to their previous state if \(e\) is found to be critical.

Recall that the changes to the adjacency lists are of two types: deleting entries that refer to edges that are no longer in the graph, and shifting entries to the adjacency sublist of the proper level. If we have a record of these modifications, we can readily reinsert adjacency list entries, or return them to their original sublists, in \(O(1)\) time.

We have demonstrated that each invalid adjacency list entry is only encountered once (and then permanently fixed). We have also shown that we can undo the modifications of the adjacency lists that are made in response to the tentative deletion of a critical edge in no more time than was required to make the modifications in the first place. Thus, we
Chapter 3. $d$-Critical Spanning Subgraphs

may bound the number of invalid adjacency list entries that we encounter by the number of invalid adjacency list entries that we generate, and use the latter value to prove the desired result.

An entry $a$ in an adjacency list is invalid when one of two things has occurred:

- The edge $(v, a)$ is no longer in $G$.
- The vertex at that entry is no longer at the level in this BFS tree which is indicated by the sublist of the adjacency list which it occupies.

Both of these are side effects of an edge being deleted. In the first case, this is obvious; in the second case, we simply observe that we only move a vertex within a BFS tree (and therefore possibly change its level) if it is in a subtree which has been disconnected by the removal of an edge. We further observe that if $v$ is already at level 2, it must be repositioned so that it is still at level 2. If it cannot be so repositioned, then the edge $e$ whose removal occasioned the repositioning of $v$ must be critical. Therefore, a successful repositioning of such a vertex $v$ cannot invalidate any adjacency list entries.

With this in mind, we consider (a) the number of vertices in each case whose repositioning can invalidate an adjacency list entry and (b) the number of adjacency list entries that can be invalidated by such the repositioning of such a vertex. In Case 1, no vertices are repositioned, and hence no adjacency list entries are invalidated. In Case 2, the vertex $j$, originally at level 1, may be repositioned at level 2, thus invalidating some adjacency list entries. All the other elements of $\tau_j$ (the subtree disconnected from its BFS tree by the removal of an edge $e = (i, j)$) are already at level 2. Finally, in Case 3, the single vertex to be repositioned ($j$) is already at level 2, and thus its repositioning cannot invalidate any adjacency list entries. We observe that a vertex may appear in $D$ adjacency lists (recall that $D$ is the maximum number of neighbors of any vertex in the graph), so the repositioning of a single vertex to a different level causes $O(D)$ adjacency
Chapter 3. d-Critical Spanning Subgraphs

list invalidations.

The deletion of an edge \( e = (i, j) \) invalidates the two entries in each BFS tree's set of adjacency lists which are associated with \( e \): the entry \( j \) in \( \text{adj}(T_k, i) \) and the entry \( i \) in \( \text{adj}(T_k, j) \), for a total of \( O(n) \) entries. We now consider the total number of additional invalidations over all BFS trees caused by the repositioning of the vertices \( v \in \tau_j \), in the light of the arguments above regarding the number of possible adjacency list invalidations that can take place. Since in Case 1 and Case 3, no adjacency list entries may be invalidated, we need not consider them here. Case 2, however, occurs in exactly two BFS trees, and in each instance \( O(D) \) adjacency list entries are invalidated. Thus, the total number of invalidations incurred over all \( T_k \in \mathcal{T} \) is \( O(D) \) for a given edge \( e \). Since there are \( m \) edges, the sum over all edges \( e \in E \) of the number of invalidated adjacency list entries is \( O(mn + mD) = O(mn) \).

We have already established that the setup and repositioning costs are \( O(mn) \) as well. Therefore, the complexity for the algorithm as a whole is \( O(mn) \).

3.3.3 Algorithm for Diameter 3 Graphs

**Theorem 3.8** Let \( G = (V, E) \) be a diameter 3 graph having \( m \) edges and \( n \) vertices. We can identify a diameter 3 critical spanning subgraph in \( O(mnD) \) time.

**Proof:** We construct the set of BFS trees \( \mathcal{T} \) and adjacency lists \( \text{adj}(T_i, k) \) for each \( T_i \in \mathcal{T} \) as in the diameter 2 algorithm described above; these processes require \( O(mn) \) time. The algorithm for diameter 3 graphs, in fact, is identical to the algorithm for diameter 2 graphs, with the exception of the restructuring step \( \text{remove_edge()} \), which is divided into more cases which are handled differently.

- **Case 1:** \( e \notin T_k \).
By definition, $e$ is a virtual edge for $T_k$. Therefore, $T_k$ is the same for the graphs $G = (V, E)$ and $G' = (V, E - e)$. Thus, removing $e$ from $G$ has no immediate effect on $T_k$. In keeping with the principle of lazy deletion, $T_k$ is not updated at this time to reflect the deletion of $e$; update may take place at a later stage if necessary. (For details on when and how updates occur, see cases 2, 3, and 4.)

- **Case 2**: $e = (i, j) \in T_i, T_j$.

Rebuild the BFS tree on the graph $G' = (V, E - e)$, and regenerate the adjacency lists for each vertex in the BFS tree. This requires $O(m)$ time.

- **Case 3**: $e = (i, j) \in T_k, l(T_k, i) = 1, l(T_k, j) = 2$.

In the discussion of BFS tree properties in section 3.3.1, we observed that a vertex could never be repositioned closer to the root as a result of an edge deletion. The children of $j$ are already at level 3, and so they must be repositioned at level 3 (or below, but if they were repositioned below level 3, that would indicate that the diameter of $G - e$ was > 3). Since $j$ is at level 2, it may be placed at level 2 or 3.

If we can find a vertex $p$ adjacent to $j$ which is at level 1, then we can simply reattach the subtree rooted at $j$ to $p$. The search for such a $p$ occurs before we attempt
Figure 3.5: An example of case 3 for diameter 3: \( T_k, l(T_k, i) = 1, l(T_k, j) = 2 \)

to reposition any other vertex in \( \tau_j \), because if it succeeds, no other repositioning is necessary. We recall that the definition of \( I(T_k, v, e) \) does not depend on the number of sublists of \( \text{adj}(T_k, v) \) which are searched, so the cost of this initial search of \( j \)'s level 1 sublist is included in \( I(T_k, j, e) \).

If this search fails, this case is then dealt with in a fashion identical to that of Case 2 of the diameter 2 algorithm, except that the level 2 sublist is searched rather than the level 1 sublist. Since the operations performed are identical (with the addition of the extra test noted above, whose cost has been accounted for) the complexity of this case is also that of Case 2 of the diameter 2 algorithm: \( O(D + \sum_{v \in \tau_j} I(T_k, v, e)) \).

(Recall that \( D \) is an upper bound on the number of vertices in \( \tau_j \).)

- **Case 4:** \( e = (i, j) \in T_k, l(T_k, i) = 2, l(T_k, j) = 3 \).

Because \( j \) is on level 3 of \( T_k \), the subtree \( \tau_j \) which is disconnected from \( T_k \) by the removal of \( e \) must consist of exactly one vertex, \( j \). Thus this case is identical to that of Case 3 of the diameter 2 algorithm, and the complexity is therefore \( O(I(T_k, j, e)) \).

The analysis of this algorithm proceeds in similar fashion to that of the diameter 2 algorithm. We use an amortized analysis to bound the number of invalid adjacency list
Chapter 3. d-Critical Spanning Subgraphs

Figure 3.6: An example of case 4 for diameter 3: $T_k, l(T_k, i) = 2, l(T_k, j) = 3$

entries that we may encounter over the course of the execution of the algorithm.

Claim 3.9

$$\sum_{e \in E} \sum_{T_k \in T} \sum_{v \in T_j} I(T_k, v, e) = O(mnD).$$

Proof: As before, we will prove the claim for the number of adjacency list entries that are rendered invalid over the course of the algorithm. Recall that this is an upper bound for the number of invalid adjacency list entries that we encounter. (See the proof of Claim 3.7 for a justification of this statement.)

Using the same analysis as for the diameter 2 case, we observe that the number of vertices repositioned which may result in the invalidation of adjacency list entries are 0 (Case 1), 1 (vertex $j$ in Case 3), and 0 (Case 4). Case 2 is considered separately: although $O(n)$ vertices are repositioned by the rebuilding of the BFS tree, all adjacency lists for this tree are also updated. (One might hope that this "free" cleanup of the adjacency lists would help us at later stages of the algorithm. We will discuss this notion in the concluding section of this chapter.) We also observe that the number of adjacency list entries that may be invalidated by the repositioning of $j$ in Case 3 is $O(D)$. 

58
Chapter 3. d-Critical Spanning Subgraphs

The deletion of an edge \( e = (i, j) \) invalidates the two entries in each BFS tree's set of adjacency lists which are associated with \( e \): the entry \( j \) in \( \text{adj}(T_k, i) \) and the entry \( i \) in \( \text{adj}(T_k, j) \), for a total of \( O(n) \) entries. We now consider the invalidations caused by the repositioning of vertices \( v \in \tau_j \) (the vertices disconnected from their BFS trees by the deletion of \( e \)). We need only consider Case 3, since by the argument above, this is the only case in which any vertex repositioning can result in the invalidation of adjacency list entries. Case 3 may be encountered in any number of BFS trees from zero to \( n - 2 \); in each case, \( O(D) \) entries are invalidated, which yields a total of \( O(nD) \) adjacency list entries which are invalidated. Thus, the total number of adjacency list entry invalidations (measured over all \( T_k \in \mathcal{T} \)) that take place in response to a given edge deletion is \( O(n + nD) = O(nD) \). Since there are \( m \) edges, the sum over all edges \( e \in E \) of the number of invalidated adjacency list entries is \( O(mnD) \).

We have already established that the initial construction of the BFS trees and generation of the adjacency lists takes \( O(mn) \) time. The remaining costs to be calculated, then, are the time required for repositioning the vertices in the BFS trees in response to the deletion of an edge, and the time required to restructure the BFS trees from scratch in instances of Case 2.

As in the algorithm for diameter 2, we observe that the time required for repositioning a vertex is \( O(1) \) if we do not include the time required to search through the adjacency lists for a valid entry. Thus, we need only consider the number of vertices that must be repositioned over the course of the algorithm. Case 1 requires no vertex repositioning. The costs for Case 2 are considered separately. Each instance of Case 3, which may occur in \( O(n) \) BFS trees for a given edge, requires \( O(D) \) vertices to be repositioned. Each instance of Case 4, which may occur in \( O(n) \) BFS trees for a given edge, requires \( O(1) \) vertices to be repositioned. Thus, since there are \( m \) edges to be considered, the total number of vertices to be repositioned over the entire algorithm is \( O(mnD) \).
Case 2 occurs twice per edge deleted, and requires $O(m)$ time in each instance, for a total of $O(m^2)$ time for all edges. Thus we have a complexity, for all operations in the algorithm, of $O(mn + mnD + m^2) = O(mnD)$ (since $m = O(nD)$).

3.3.4 Algorithms for General Graphs

Recall that the complexity of the naive approach to identifying $d$-critical spanning subgraphs is $O(m^2n)$. We have not yet succeeded in developing an algorithm which is more efficient than this for graphs of diameter $d \geq 4$. The purpose of this section is to explore the reasons why our current algorithms do not appear to have natural extensions to the graphs of general $d$, and discuss alternate methods that might be used to improve this bound.

A Generalization That Doesn’t Work

In order to discuss the generalization of the BFS-tree-based algorithms to graphs of general diameter, we must first establish some terminology. As above, we define $\tau_j$ to be the subtree that is disconnected from a given BFS tree $T_k$ by the removal of $e$. We also define $U(T_k, \lambda)$ to be the subset of $\tau_j$ whose vertices are at level $p \leq \lambda$ and which have not yet been reconnected to $T_k$. Initially, $\lambda$ is set to $l(j) - 1$.

We then use a slight modification of the procedure described in the algorithms for $d = 2$ and $d = 3$ to reattach each $u \in U(T_k, \lambda)$. We search the level $\lambda$ sublist of the adjacency lists for each such $u$. If we encounter an adjacency list element which is an element of $U(T_k, \lambda)$ (and hence of currently undefined level, since we have not yet reconnected it to $T_k$) we move it to the sublist for vertices of level $\lambda + 1$. If we manage to reconnect all $u \in U(T_k, \lambda)$, then we are done; all subtrees rooted at vertices in $U(T_k, \lambda)$ will also be at the closest possible distance to the root of $T_k$. Otherwise, we increment $\lambda$ and continue until either all vertices are reattached to $T_k$, or we find that there exists a
vertex $u \in U(T_k, \lambda)$ such that $l(T_k, u) \geq d$ (thus proving that $e$ is critical).

We observe that a vertex in $U(T_k, \lambda)$ can only change its level $O(d)$ times in response to the deletion of a given edge $e$. (One might think that a vertex in any $T_k$ could move down at most $d$ times over the course of the entire algorithm. However, if $e$ is critical, all $T_k \in \mathcal{T}$ are returned to the state that they were in before we started restructuring them in response to the anticipated deletion of $e$.) Since $|\mathcal{T}| = O(D)$, and $|\mathcal{T}| = O(n)$, the total amount of work done over the course of the algorithm in repositioning vertices is $O(dmnD)$. Note that, for fixed $d$, this portion of the algorithm requires asymptotically as much time as the algorithm for $d = 3$.

Any adjacency list entry can be invalidated $O(d)$ times in response to the removal of an edge $e$, since each time an invalid entry is encountered in the level $\lambda$ sublist, it is repositioned in the level $\lambda + 1$ sublist. Since there are a total of $mn$ adjacency list entries associated with the BFS trees, the number of invalid adjacency list entries that we may encounter is $O(dm^2n)$. As before, the extra factor of $O(m)$ arises, ironically, from the fact that if $e$ is discovered to be critical, the BFS trees and adjacency lists are returned to their previous states.

Thus, this strategy does not seem to generalize gracefully; the total complexity of this approach is $O(dm^2n)$, which is certainly no better than the complexity of the original naive approach: $O(m^2n)$. Given the fact that this generalized strategy closely resembles the tree-growing that results from a naive restructuring of the BFS trees, this result is perhaps not terribly surprising.

Possible Improvements

When we attempt to improve on the efficiency of a given algorithm, we should remember that there are two ways of doing so. The first, and most straightforward, is to design an
algorithm that uses a different approach to solving the problem. The second makes use of the fact that, in many cases, there is no proof that the anticipated worst-case behavior of the algorithm actually ever arises in practice (and sometimes not even in theory, once the theory is sufficiently closely examined). Thus, it is sometimes possible to "improve" the running time of an algorithm by demonstrating that tighter bounds apply than those that have already been proven.

In the case of these algorithms, we have not yet succeeded in generating a graph that demonstrates the worst-case behavior that we predict. As a result, we would like to investigate the following possibilities:

- In the \( d = 3 \) case, the BFS trees are restructured whenever the edge \( e = (i,j) \) that we are deleting is found in the BFS trees \( T_i \) and \( T_j \). Since the adjacency lists for these trees are brought up to date at this point, it is possible that a more careful analysis of the number of invalid adjacency list entries would reveal that there cannot be as many as we have predicted. If so, perhaps we might improve on the complexity of the algorithm for general \( d \) described above.

Recall that the process that determines the complexity of the algorithm is that of searching through the adjacency lists for a valid entry. By judicial rebuilding of the BFS trees (either when we have reason to believe that there are sufficient numbers of invalid adjacency list entries to make it worthwhile, or at regular intervals), we might improve on the worst-case predictions of the number of invalid adjacency list entries that we could encounter.

- We calculate the number of adjacency list entries that are invalidated, and use this to bound the number of invalid adjacency list entries that are actually encountered. Perhaps the number of invalid entries that the algorithm must deal with is asymptotically fewer than the number of invalid entries that are generated.
Chapter 3. \textit{d-Critical Spanning Subgraphs}

3.4 Concluding Remarks

We have seen that there are efficient methods of identifying \textit{d}-critical spanning subgraphs of graphs of diameter \( d = 2 \) or \( d = 3 \). Unfortunately, these methods do not appear to readily generalize to general \( d \). (This has prompted us to speculate that we have perhaps unfairly maligned the method we first discussed—tentatively removing each edge in turn and calculating the diameter of the resultant graph—by referring to it repeatedly as "naive".)

3.4.1 Further Uses of Matrix Multiplication

The success of Anstee and Caccetta [1] in adapting matrix multiplication to the problem of recognizing diameter-critical graphs leads us to believe that there may be benefits in the further exploration of its use in our closely related problem. We already have an algorithm for identifying \( d \)-critical spanning subgraphs that uses matrix multiplication to calculate the diameter of the graph as each edge is removed, and has a complexity of \( O(m \log d \cdot M(n)) \) (or, for a given value of \( d \), \( O(m \cdot M(n)) \)). It seems likely that this is as far as this particular use of matrix multiplication can be taken.

However, suppose that we have a graph \( G \) with diameter \( d \) represented by an adjacency matrix \( A \). If we calculate \( d \) by the method that we described in section 3.1.2, we generate a series of matrices \( A_1, A_2, \ldots, A_2, \ldots, A_d \) (where \( A_k = (A + I)^k \)). If we delete an edge \( e \), recalculate \( A_d \), and \( A_d \neq 0 \), then we know that \( e \) is critical. We would like to have some way of updating \( A_d \) in response to the deletion of an edge \( e \) that does not require as much work as matrix multiplication. (Of course, we expect to have to update at least some of the other matrices as well in order to be able to update \( A_d \).) We observe that deleting an edge changes only 2 entries in \( A \), and hypothesize that there may be some way of quickly updating the matrices that takes advantage of this fact and runs in, perhaps, \( O(n^2) \) or
Chapter 3. d-Critical Spanning Subgraphs

\(O(n^2 \lg d)\) time. This would give us an overall time of \(O(mn^2 \lg d)\).

3.4.2 Parallelization

The two different methods that we have discussed of calculating the diameter of a graph—BFS trees and matrix multiplication—yield efficient parallel algorithms for finding d-critical spanning subgraphs. The \(n\) BFS trees can be grown and manipulated by \(n\) different processors in parallel, or, if only \(k < n\) processors are available, the work can be distributed evenly among the \(k\) processors (so that each handles \(n/k\) BFS trees) with no difficulty. The advantages of this method are that it is very easy to implement as a parallel algorithm. Its implementation does not depend on the topology of the parallel machine, it requires no knowledge of special parallel programming tricks, and the algorithm can efficiently use any number of processors from 1 to \(n\). The parallel implementation of the naive algorithm using BFS trees would have a time complexity of \(O(m^2)\) (with \(O(n)\) processors). (We can, of course, achieve the same \(O(n)\) speedup with the algorithms that we have developed for graphs of diameter \(d = 2\) and \(d = 3\).)

On the other hand, we can multiply matrices using \(n^2\) processors in \(O(n)\) time. The parallel implementation of the naive algorithm using matrix multiplication would then have a time complexity of \(O(mn)\) (with \(n^2\) processors), which is clearly faster than the parallel implementation using BFS trees. This method does have some drawbacks, however. It is a more specialized use of parallel computation, since the implementation of the algorithm requires a very specific communication protocol among the processors. As a result, the efficiency of the algorithm is also dependent to a certain extent on the topology of the machine on which it is to be executed. It does not have any application to the faster algorithms for \(d = 2\) and \(d = 3\). Finally, it is not as readily scalable as the parallel implementation of the BFS tree algorithm; in other words, it is not immediately clear how to implement this algorithm on a system with < \(n^2\) processors.
Bibliography


