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 $d \geq 4$.

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 $v \in V(G)$, 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 $\leq d$, and the tallest
Chapter 3. \textit{d-Critical Spanning Subgraphs}

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 $O(n^3/\log^2 n)$. In practice, the

35
Chapter 3. d-Critical Spanning Subgraphs

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

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

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 $\leq k + 1$. □

Proposition 3.2 $A_{k+1} \geq A_k$.

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 $A_k$. □

Lemma 3.3 $diam(G)$ is the smallest $k$ for which $A_k > 0$.

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 $\text{diam}(G) = d < k$. 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 $k$. ■

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

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

$(A_1)^2 \rightarrow A_2, (A_2)^2 \rightarrow A_4, \ldots, (A_{2^j})^2 \rightarrow A_{2^{j+1}}$, 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 $A_{2^j} > 0$ 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 $j = [\lg d]$. Therefore, thus far we have executed $\lg d$ iterations, each of which required $O(M(n) + n^2) = O(M(n))$ time, for a total of $O(M(n) \lg d)$ 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 $A_{2i-1}$, 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.

graph diameter_bsearch(matrix_set A, integer j)
{
    integer i, d;
    matrix B;

    $B \leftarrow A_{2^{j-1}}$;
    $d \leftarrow 2^{j-1}$;

    for i from $j - 2$ downto 0 do
    {
        if ($BA_{2^i} > 0$)
        {
            $B \leftarrow BA_{2^i}$;
            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 $A_{2^i}$ 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 $j = \lfloor \lg d \rfloor$. 

39
Chapter 3. $d$-Critical Spanning Subgraphs

Each iteration requires $O(1)$ matrix multiplications, $O(n^2)$ work to determine whether $BA_i > 0$, and one call to \texttt{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 $O(\min\{m^2n, m \cdot M(n) \lg d\})$ 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 \textit{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, \textit{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 (Plesník [11])} \textit{The following problem is NP-hard: given an edge-weighted graph $G$ with cost function $c : E \to \mathbb{R}$ 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 $G_c \subseteq G$.
graph naive_critical_subgraph(graph G, integer d)
{
    graph $G_c, G'$;
    edge $e$;
    vertex_set $V$;
    edge_set $E, E_c$;

    $V \leftarrow$ vertices($G$);
    $E \leftarrow$ edges($G$);
    $E_c \leftarrow E$;
    for each $e \in E$ do
    {
        $G' \leftarrow$ define_graph($V, E_c - e$);
        if (diam($G'$) $\leq$ $d$)
            $E_c \leftarrow E_c - e$;
    }
    $G_c \leftarrow$ 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 $O(\min \{m^2 n, m \lg d \cdot M(n)\})$. (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 $d$, naive_critical_subgraph(), as implemented above, runs in time $O(\min \{m^2 n, m \cdot M(n)\})$.) Specifically, we replace, in 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: is_2_critical($G'$) or 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 $O(m^2 n + m \cdot M(n))$ 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

$$\text{diam}(G) \equiv \max\{\text{height}(T_v) : T_v \in \mathcal{T}\}$$

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 ← vertices(G);
  E ← edges(G);
  E_c ← E;
  for each v ∈ V do
    T_v ← create_BFS_tree(V, E, v);
  for each e ∈ E do
    {
      critical ← FALSE;
      for each v ∈ V do
        {
          while (critical = FALSE) do
            {
              T'_v ← remove_edge(T_v, e);
              if (height(T'_v) ≥ d)
                critical ← TRUE;
          }
        }
      if (critical = FALSE)
        {
          E_c ← E_c − e;
          for each v ∈ V do
            T_v ← T'_v;
        }
    }
  G_c ← 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 \texttt{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 \textit{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
Chapter 3. *d*-Critical Spanning Subgraphs

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 $\text{adj}(T_i, k)$ 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 $\text{adj}(T_i, u)$, and $u$ is a member of $\text{adj}(T_i, v)$.

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.
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
continue searching.

- If there is no such $a$, $e$ is 2-critical. Abort the restructuring of the BFS trees, and restore all $T_i \in T$ 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 $v \in T_j$ 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 $v \in T_j$, 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 $T_j$, the total time required to reattach all vertices in $T_j$ is

$$O(D + \sum_{v \in T_j} I(T_i, v, e))$$

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 $T_j$ 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 $O(I(T_k, j, e))$. 

50
Chapter 3. \textit{d-Critical Spanning Subgraphs}

![Tree Diagram]

Figure 3.3: An example of case 3 for diameter 2: $T_k$

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 \textit{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 $adj(T, v)$ 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 $T_k \in T$ and all edges $e \in E$.

Claim 3.7

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

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
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
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 (\texttt{remove.edge}()), which is divided into more cases which are handled differently.

- **Case 1:** \( e \notin T_k \).
Chapter 3. \textit{d-Critical Spanning Subgraphs}

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. \textit{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.

\textbf{Claim 3.9}

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

\textbf{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
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 take \( O(mn) \) 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
Chapter 3. \(d\)-Critical Spanning Subgraphs

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 \(|\tau_j| = O(D)\), 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. d-Critical Spanning Subgraphs

3.4 Concluding Remarks

We have seen that there are efficient methods of identifying 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 \( O(m \lg d \cdot M(n)) \) (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 \( A_1, A_2, \ldots, A_{2^d}, \ldots, A_d \) (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

63
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.