Algorithms and Dynamic Data Structures for Basic Graph Optimization Problems

by

Ran Duan

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

Doctoral Committee:

Assistant Professor Seth Pettie, Chair
Professor Anna C. Gilbert
Professor Quentin F. Stout
Associate Professor Martin Strauss

TABLE OF CONTENTS

LIST OF FIGURES

Figure

ABSTRACT

Algorithms and Dynamic Data Structures for Basic Graph Optimization Problems

by

Ran Duan

Chair: Seth Pettie

Graph optimization plays an important role in a wide range of areas such as computer graphics, computational biology, networking applications and machine learning. Among numerous graph optimization problems, some basic problems, such as shortest paths, minimum spanning tree, and maximum matching, are the most fundamental ones. They have practical applications in various fields, and are also building blocks of many other algorithms. Improvements in algorithms for these problems can thus have a great impact both in practice and in theory.

In this thesis, we study a number of graph optimization problems. The results are mostly about approximation algorithms solving graph problems, or efficient dynamic data structures which can answer graph queries when a number of changes occur. There are several different models of dynamic graphs. Much of my work focuses on the *dynamic subgraph model* in which there is a fixed underlying graph and every vertex can be flipped "on" or "off". The queries are based on the subgraph induced by the "on" vertices. Our results make significant improvements to the previous algorithms or structures of these problems.

The major results are listed below.

- *Approximate Matching.* We give the first linear time algorithm for computing approximate *maximum weighted matching* for arbitrarily small approximation ratio.

- *d-failure Connectivity Oracle.* For an undirected graph, we give the first space-efficient data structure that can answer connectivity queries between any pair of vertices avoiding d other failed vertices in time polynomial in $d \log n$.

- *(Max, Min)-Matrix Multiplication* We give a faster algorithm for the (max, min)-matrix multiplication problem, which has a direct application to the all-pairs bottleneck paths (APBP) problem. Given a directed graph with a capacity on each edge, the APBP problem is to determine, for all pairs of vertices s and t, the path from s to t with maximum flow.

- *Dual-failure Distance Oracle.* For a given directed graph, we construct a data structure of size $\tilde{O}(n^2)$ which can efficiently answer distance and shortest path queries in the presence of two node or link failures.

- *Dynamic Subgraph Connectivity.* We give the first subgraph connectivity structure with worst-case sublinear time bounds for both updates and queries.

- *Bounded-leg Shortest Path.* In a weighted, directed graph an L-bounded leg path is one whose constituent edges have length at most L. We give an algorithm for preprocessing a directed graph in $\tilde{O}(n^3)$ time in order to answer *approximate* bounded leg distance and bounded leg shortest path queries in merely sub-logarithmic time.

CHAPTER I

Introduction

This thesis studies several graph optimization problems. Graph optimization plays an important role in a wide range of areas such as computer graphics, computational biology, networking applications and machine learning. Among numerous graph optimization problems, some basic problems, such as shortest paths, minimum spanning tree, and maximum matching, are the most fundamental ones. They have practical applications in various fields, and are also building blocks of many other algorithms. Much of my research concerns computing shortest paths and maximum matching. The shortest path problem is essential in web mapping and network routing applications, while the maximum matching problem has applications to assignment problems. They are also important in solving other graph optimization problems like the min-cost maximum flow problem or edge disjoint paths problem. Improvements in algorithms for these problems can thus have a great impact both in practice and in theory.

As we see in the example of web mapping applications, the maps in real world are vulnerable to changes caused by traffic congestions, road failures, or construction of new roads. Instead of re-computing all the information when a change occurs, we may keep as much information of the previous graph as possible in order to improve the running time. A common way to deal with this is building data structures on such

dynamic graphs, which have fast algorithms for updating the structure and answering queries about some graph optimization problem. The running times for updates and queries are usually faster than the original static algorithm on the same problem.

In this thesis we study different variations of several basic graph optimization problems, including bounded-leg shortest paths, data structures maintaining shortest paths or connectivity for failure-prone graphs, worst-case dynamic structure for connectivity, and also algorithms to find all-pair bottleneck paths and approximate maximum weighted matching.

1.1 Basic Concepts and Notations

In this thesis, we denote the primary graph we are working on by $G = (V, E)$, where V is the set of vertices and E is the set of edges in G. Let $n = |V|$ and $m = |E|$. The graph can be directed or undirected. A *path* p is a sequence of consecutive edges. In a graph with weight function $w : E \to \mathbb{R}$ on edges, the shortest path problem considers the path minimizing $\sum_{e \in p} w(e)$ between two vertices, while the connectivity problem only considers whether there is a path connecting two vertices. In this thesis, all the connectivity problems are in undirected graphs, whereas shortest paths and bottleneck paths are in directed graphs.

A *matching* M in a graph G is a set of edges without common vertices. A vertex associated with an edge in the matching is called *matched*, otherwise it is *unmatched*. A matching in which all vertices are matched is called a *perfect matching*. In a weighted graph, the *maximum weighted matching* is the matching maximizing $\sum_{e \in M} w(e)$. Note that it is not necessarily perfect.

Usually there are several types of dynamic graph models. In a *fully dynamic* model we can add or delete edges/vertices arbitrarily. There are also *incremental* and *decremental* graphs in which we can only insert or delete edges/vertices, respectively. However, in this thesis we consider a dynamic graph model called the *dynamic*

subgraph model in which there is a fixed underlying graph, and every vertex in that graph can be "active" or "inactive". The distance/connectivity queries are based on the subgraph induced by the active vertices. We also study two types of this model based on whether there is a restriction on the number of inactive vertices. The structures in Chapter VI do not have such a restriction, that is, any vertex can change its status at any time. However, the results in Chapter III and V consider the dynamic subgraph model in which the number of inactive vertices is bounded by some number d. We can see this type of structure as static, which can preprocess the entire graph and answer the distance/connectivity queries given with several "failed" vertices. This is the "d-failure model". In the connectivity structure of Chapter III, d can be an arbitrary integer, while in the shortest path structure of Chapter V, d is at most 2.

In this paper, $\tilde{O}(\cdot)$ hides poly-logarithmic factors. For example $O(n^{1/2}\log n)$ can be written as $\tilde{O}(n^{1/2})$.

1.2 Overview of the Results

1.2.1 Shortest Path and Bottleneck Path

The all-pair shortest path problem is one of the most fundamental and most studied optimization problems in graph theory. It can be solved by applying the Dijkstra's algorithm from every vertex in the graph, which has a total running time of $O(mn + n^2 \log n)$. (See [17].) A faster running time of $O(mn + n^2 \log\log n)$ was achieved by Pettie [50]. For dense graphs, the Floyd-Warshall algorithm [13] provides a clearer way to achieve the time bound of $O(n^3)$. We can also see the all-pair shortest path problem as the transitive closure of the $(\min, +)$ matrix product. However, since $(\min, +)$ is not a ring, the fast matrix multiplication algorithms like [12] cannot be directly applied to it. However, Shoshan and Zwick [58, 69] gave algorithms of $o(n^3)$

running time for computing all-pair shortest paths in unweighted or small integer weighted graphs by fast matrix multiplication. The current best algorithm for real-weighted graph is given by Chan [9], which has a running time of about $O(n^3/\log^2 n)$.

In this thesis, we consider several variations of the all-pair shortest path problem: dynamic shortest path, bounded-leg shortest path and all-pair bottleneck path, which are discussed in the following.

1.2.1.1 Dual-failure Shortest Path Structure

In this problem we consider a data structure answering distance queries in a weighted directed graph $G = (V, E, w)$, where one or more nodes or edges are unavailable due to failure or other causes. Specifically, given source and target vertices x, y and a set $F \subset V$, the problem is to report $\delta_{G-F}(x, y)$, where $\delta_{G'}$ is the distance function w.r.t. a subgraph G' of G. In the absence of failure, the best oracle for answering distance queries in $O(1)$ time is a trivial $n \times n$ lookup table. Thus, a distance oracle that is sensitive to node failures should be considered (nearly) optimal if it occupies (nearly) quadratic space and answers queries in (nearly) constant time. Demetrescu et al. [16] showed that single-failure distance queries can be answered in constant time by an oracle occupying $O(n^2 \log n)$ space. Very recently Bernstein and Karger improved the construction time of [16] from $\tilde{O}(mn^2)$ to $\tilde{O}(nm)$ [6]. They also highlighted the problem of finding distance oracles capable of dealing with more than one failure.

In Chapter V we show that dual-failure distance queries can be answered in $O(\log n)$ time using $O(n^2 \log^3 n)$ space. This data structure and query algorithm are considerably more complicated than those of [16, 5] due to multiple possibilities of intersection of the "detour" avoiding the two failed vertices and the original shortest path. As a special case, this structure also allows one to answer dual-failure connectivity queries in $O(\log n)$ time.

1.2.1.2 Bounded-leg Shortest Path

In this problem, our input is a weighted directed graph $G = (V, E, w)$, where $|V| = n, |E| = m$, and $w : E \to \mathbb{R}^+$. An L-bounded leg shortest path is a shortest path in the graph restricted to edges with length at most L. If we wanted to compute point-to-point or all-pairs shortest paths and L is known the problem would be very simple: just discard all unavailable edges and solve the problem as usual. We consider the more realistic situation where the graph G is fixed and L-bounded leg distance/shortest path queries must be answered online. In other words, we need a data structure that can answer queries for *any* given leg bound L. Our goals are to minimize the construction time of the data structure, its space, its query time, and the *quality* of the estimates returned. We say that a distance estimate is α-approximate if it is within a factor of α of the actual distance.

The bounded-leg shortest path problem (BLSP) was studied most recently by Roditty and Segal [53]. (See also [7].) They showed that an $\tilde{O}(n^{2.5})$-space data structure could be built in $O(n^4)$ time that answers $(1+\epsilon)$-approximate bounded leg shortest path queries. They also showed that when the graph is induced by points in a d-dimensional l_p metric that a more time and space-efficient data structure could be built for answering $(1+\epsilon)$-approximate BLSP queries. Specifically, the construction time and space are $O(n^3(\log^3 n + \epsilon^{-d}\log^2 n))$ and $O(n^2\epsilon^{-1}\log n)$, respectively. Roditty and Segal's construction made use of complicated algorithms for computing sparse geometric spanners.

In Chapter VII, we give a new, efficiently constructible $(1+\epsilon)$-approximate BLSP data structure for arbitrary directed graphs. The construction time and space of our data structure improve significantly on Roditty and Segal's structure for arbitrary directed graphs and basically match the time and space usage of their structure for l_p^d metrics. In $O(n^3\epsilon^{-1}\log^3 n)$ time we can build a $O(n^2\epsilon^{-1}\log n)$-space data structure that answers distance queries in $O(\log(\epsilon^{-1}\log n))$ time and BLSP queries

in $O(\log(\epsilon^{-1}\log n))$ per edge. One of the main advantages of our algorithm is its simplicity. It is based on a generalized version of the Floyd-Warshall algorithm and retains its streamlined efficiency.

1.2.1.3 All-pair Bottleneck Path

Besides the shortest path problem, we also study another fundamental type of path: the bottleneck path. Given a directed graph with a capacity on each edge, the *all-pairs bottleneck paths* (APBP) problem is to determine, for all vertices s and t, the path with maximum flow that can be routed from s to t. Note that it is essentially different from the traditional maximum flow problem, where the flow can be composed of multiple paths. For dense graphs this problem is equivalent to that of computing the $(\max, \min)$-transitive closure of a real-valued matrix. It is shown that APBP can be computed in $O(n^{2+\mu}) = O(n^{2.575})$ time on vertex capacitated-graphs [57] and $O(n^{2+\omega/3}) = O(n^{2.792})$ time on edge capacitated graphs [65]. (Here $\omega = 2.376$ is the exponent of binary matrix multiplication [12] and $\mu \geq 1/2$ is a constant related to rectangular matrix multiplication.)

Shapira et al. [57] and Vassilevska et al. [65] generalized APBP to the *all pairs bottleneck shortest paths* problem (APBSP, also known as the maximum capacity paths problem) in graphs with real *capacities* assigned to edges/vertices. In APBSP, one asks for the maximum capacity path among shortest paths. Shapira et al. [57] gave an APBSP algorithm running in $O(n^{(8+\mu)/3}) = O(n^{2.859})$ time. An unpublished algorithm of Vassilevska [63] computes APBSP on edge-capacitated graphs in $O(n^{(15+\omega)/6}) = O(n^{2.896})$ time.

In Chapter IV we develop faster algorithms for $(\max, \min)$-product, APBP in edge-capacitated graphs, and all-pairs bottleneck shortest paths in both vertex and edge-capacitated graphs. We introduce a simple technique called *row balancing* (or *column balancing*) that decomposes a matrix into a sparse component and a dense

component with uniform row (or column) density. Using this technique we exhibit an extremely simple algorithm for computing the *dominance product* on sufficiently sparse matrices in $O(n^\omega)$ time, as well as an algorithm for somewhat denser matrices that runs in time $O(\sqrt{mm'}n^{(\omega-1)/2})$. (This last bound was claimed earlier in [65]; it was based on a more complicated algorithm [64].) Using the sparse dominance product and row balancing we show how to compute the $(\max, \min)$-product (and, therefore, APBP) in $O(n^{(3+\omega)/2}) = O(n^{2.688})$ time. This improves on the previous $O(n^{2+\omega/3}) = O(n^{2.792})$ time algorithm [65]. We also give algorithms to compute APBSP in $O(n^{(3+\omega)/2})$ time on edge-capacitated graphs and $O(n^{2.657})$ time on vertex-capacitated graphs, which are significant improvements over [63, 57], which run in $O(n^{(15+\omega)/6}) = O(n^{2.896})$ and $O(n^{(8+\mu)/3}) = O(n^{2.859})$ time, respectively.

1.2.2 Dynamic Connectivity

Dynamic connectivity and shortest path problems have been studied for a long time. Most of the previous research on this topic focused on the "general model" of dynamic graph, that is, one can delete vertices and edges or insert new ones in an arbitrary way. However, the dynamic connectivity model considered in this thesis is based on what is called the *dynamic subgraph* model, in which we assume that there is some fixed underlying graph and that updates consist solely of making vertices and edges *active* or *inactive*. The model in Chapter III also restricts the number of inactive vertices at any time. In this model, we can preprocess the underlying graph to obtain more efficient updates and queries.

Dynamic connectivity with edge updates is the most basic problem among these kinds of dynamic structures and is well studied. Holm, Lichtenberg, and Thorup have introduced a linear space structure supporting $O(\log^2 n)$ amortized update time [38, 59]. With this structure, we can get a trivial dynamic subgraph connectivity structure with amortized vertex update time $\tilde{O}(n)$. Then two hard directions related to this

problem arise: dynamic subgraph connectivity with sublinear vertex update time, and dynamic structures with worst-case edge/vertex update time bounds.

For the fully dynamic subgraph model, in which we can flip a vertex at any time, Frigioni and Italiano [32] gave a dynamic subgraph connectivity structure having amortized polylogarithmic vertex update time in planar graphs. Recently, Chan, Pătraşcu and Roditty [10] gave a subgraph connectivity structure for general graphs supporting $\tilde{O}(m^{2/3})$ vertex update time with $\tilde{O}(m^{4/3})$ space, which improves the result given by Chan [8] having $\tilde{O}(m^{0.94})$ update time and linear space.

However, the dynamic structures mentioned above all have amortized update time. In general, worst-case dynamic structures have much worse time bounds than amortized structures. The best dynamic edge-update connectivity structure in the worst-case scenario has update time $O(n^{1/2})$ [29, 28]. Improving this time bound is still a major challenge in dynamic graph algorithms. For the d edge failure model, Pătraşcu and Thorup [49] gave a data structure that can process any d edge deletions in $O(d \log^2 n \log\log n)$ time and then answer connectivity queries in $O(\log\log n)$ time. Using those worst-case edge update structures, we give two natural generalizations in this thesis: the first efficient d-vertex failure connectivity oracle with update and query time polynomial of $\log n$ and d, and the first dynamic subgraph connectivity structure with sublinear vertex update time in the worst-case scenario.

For a survey of recent *fully* dynamic graph algorithms (i.e., not dynamic subgraph algorithms), refer to [38, 54, 61, 55, 15, 60].

Our Results In Chapter III, we present a new, space efficient data structure that can quickly answer connectivity queries after recovering from d *vertex failures*. The recovery time is polynomial in d and $\log n$ but otherwise independent of the size of the graph. After processing the failed vertices, connectivity queries are answered in $O(d)$ time. There is a tradeoff in our oracle between the space, which is roughly mn^{ϵ}, for

$0 < \epsilon \leq 1$, and the polynomial query time, which depends on ϵ. Our data structure is the first of its type. To achieve comparable query times using existing data structures we would need either $\Omega(n^d)$ space [19] or $\Omega(dn)$ recovery time [49]. As a byproduct, we also give a new d edge failure oracle with $O(d^2 \log\log n)$ processing time, which is much simpler than Pătraşcu and Thorup's structure. [49]

In Chapter VI, we study the fully dynamic subgraph connectivity problem for undirected graphs. We give the first subgraph connectivity structure with worst-case sublinear time bounds for both updates and queries. Our worst-case subgraph connectivity structure supports $\tilde{O}(m^{4/5})$ update time, $\tilde{O}(m^{1/5})$ query time and occupies $\tilde{O}(m)$ space. We also give another dynamic subgraph connectivity structure with amortized $\tilde{O}(m^{2/3})$ update time, $\tilde{O}(m^{1/3})$ query time and linear space, which improves the structure introduced by Chan, Pătraşcu, and Roditty [10] that takes $\tilde{O}(m^{4/3})$ space.

1.2.3 Matching

Although the maximum matching problem has been studied for decades, the computational complexity of finding an optimal matching remains quite open. In 1965 Edmonds presented elegant polynomial time algorithms for finding matchings in general graphs with maximum cardinality (MCM) [27] and maximum weight (MWM) [26]. Early implementations of Edmonds's algorithm required $O(n^3)$ time [41, 36, 43] using elementary data structures. Following the approach of Hopcroft and Karp's MCM algorithm for bipartite graphs [39], Micali and Vazirani [47] presented an MCM algorithm for general graphs running in $O(m\sqrt{n})$ time.

For maximum weighted matching, the implementation of the Hungarian algorithm [42] using Fibonacci heaps [30] runs in $O(mn+n^2 \log n)$ time in bipartite graphs, a bound that is matched in general graphs by Gabow [33] using more complex data structures. Faster algorithms are known when the edge weights are bounded inte-

gers in $[-N, \ldots, N]$, where a word RAM model is assumed, with $\log(\max\{N, n\})$-bit words. Gabow and Tarjan [34, 35] gave bit-scaling algorithms for MWM running in $O(m\sqrt{n}\log(nN))$ time in bipartite graphs and $O(m\sqrt{n\log n}\log(nN))$ time in general graphs.

Approximation Algorithms Let a δ-MWM be a matching whose weight is at least a δ fraction of the maximum weight matching, where $0 < \delta \leq 1$, and let δ-MCM be defined analogously. There are simple ways to find $(1 - 1/k)$-MCM in $O(km)$ time. [39, 47] However, the best approximate MWM algorithms do not achieve similar approximation and time bounds. On real weighted graphs the Gabow-Tarjan algorithm [35] gives a $(1 - n^{-\Theta(1)})$-MWM in $O(m\sqrt{n}\log^{3/2} n)$ time, simply by retaining the $O(\log n)$ high order bits in each edge weight, treating them as polynomial size integers. It is well known that the *greedy* algorithm—iteratively choose the maximum weight edge not incident to previously chosen edges—produces a $\frac{1}{2}$-MWM. A straightforward implementation of this algorithm takes $O(m \log n)$ time. Preis [52, 18] gave a $\frac{1}{2}$-MWM algorithm running in linear time. Vinkemeier and Hougardy [67] and Pettie and Sanders [51] proposed several $(\frac{2}{3} - \epsilon)$-MWM algorithms (see also [46]) running in $O(m \log \epsilon^{-1})$ time; each is based on iteratively improving a matching by identifying sets of short weight-augmenting paths and cycles. No linear time algorithms with approximation ratio better than $\frac{2}{3}$ were known.

Our Results In Chapter II, we present the first near-linear time algorithm for computing $(1-\epsilon)$-approximate MWMs. Specifically, given an arbitrary real-weighted graph and $\epsilon > 0$, our algorithm computes such a matching in $O(m\epsilon^{-1}\log\epsilon^{-1})$ time, which improves our preliminary result appearing in FOCS 2010 of running time $O(m\epsilon^{-2}\log^3 n)$.

1.3 Publications Arising from this Thesis

Approximating Maximum Weight Matching in Near-linear Time. FOCS 2010 (IEEE Symposium on Foundations of Computer Science)

New Data Structures for Subgraph Connectivity. ICALP 2010 (International Colloquium on Automata, Languages and Programming)

Connectivity Oracles for Failure Prone Graphs. STOC 2010 (ACM Symposium on Theory of Computing)

Dual-Failure Distance and Connectivity Oracles. SODA 2009 (ACM-SIAM Symposium on Discrete Algorithms)

Fast Algorithms for (Max, Min)-Matrix Multiplication and Bottleneck Shortest Paths. SODA 2009 (ACM-SIAM Symposium on Discrete Algorithms)

Bounded-leg Distance and Reachability Oracles. SODA 2008 (ACM-SIAM Symposium on Discrete Algorithms)

CHAPTER II

Approximate Maximum Weighted Matching in Linear Time

2.1 Introduction

Our main result in this chapter is the first $(1-\epsilon)$-MWM algorithm for arbitrary weighted graphs whose running time is linear. In particular, we show that such a matching can be found in $O(m\epsilon^{-1}\log\epsilon^{-1})$ time, [1] leaving little room for improvement. This new result will be published in a journal article.

Technical Challenges The easiest among linear time approximate MWM algorithms is the greedy algorithm for 1/2-MWM, in which we choose the maximum weight edge not incident to previously chosen edges every time. Preis [52, 18] gave an algorithm achieving this approximation in linear time. There are two natural ways to extend the approximation ratio. The first one is to find longer alternating paths and cycles which can increase the total weights. The algorithms for 2/3-MWM in [51, 67] follow this approach, which are able to handle alternating cycles of length 4. However, since directly finding long weight-augmenting alternating paths or cycles is hard to achieve in almost linear time, we need alternative ways to achieve the approximation

[1] A preliminary result of $O(m\epsilon^{-2}\log^3 n)$ running time appears in Duan and Pettie's paper "Approximating Maximum Weight Matching in Near-linear Time" [24] in FOCS 2010.

ratio of $1-\epsilon$ for arbitrarily small ϵ. The other approach is to follow the scaling algorithms of Gabow and Tarjan [35], which solve the MWM problem at about $\log N$ scales. In each scale they follow a primal-dual relaxation on the linear programming formulation of MWM. This relaxed complementary slackness approach relaxes the constraint of the dual variables by a small amount, so that the iterative process of the dual problem will converge to an approximate solution much more quickly. While their algorithm takes $O(\sqrt{n})$ iterations of augmenting to achieve a perfect matching, we proved that we only need $O(\log N/\epsilon)$ iterations to achieve a $(1-\epsilon)$-approximation, where we can assume $N \leq n^2$. Also we make the relaxation "dynamic" by tightening the relaxation when the dual variables decrease by one half, so that finally the relaxation is at most ϵ times the edge weight on each matching edge and very small on each non-matching edge, which gives an approximate solution.

2.2 Definitions and Preliminaries

The input is a graph $G = (V, E, w)$ where $|V| = n, |E| = m$, and $w : E \to \mathbb{R}$. We use $E(H)$ and $V(H)$ to refer to the edge and vertex sets of H or the graph induced by H, that is, $V(E')$ is the set of endpoints of $E' \subseteq E$ and $E(V')$ is the edge set of the graph induced by $V' \subseteq V$. A *matching* M is a set of vertex-disjoint edges. Vertices not incident to an M edge are *free*. An alternating path (or cycle) is one whose edges alternate between M and $E \backslash M$. An alternating path P is *augmenting* if P begins and ends at free vertices, that is, $M \oplus P \stackrel{\text{def}}{=} (M \backslash P) \cup (P \backslash M)$ is a matching with cardinality $|M \oplus P| = |M| + 1$.

Since we only need $(1-\epsilon)$ approximate solutions, we can afford to scale and round edge weights to small integers. To see this, observe that the weight of the MWM is at least $w_{\max} = \max\{w(e) \mid e \in E(G)\}$. It suffices to find a $(1-\epsilon/2)$-MWM M with respect to the weight function $\tilde{w}(e) = \lfloor w(e)/\gamma \rfloor$ where $\gamma = \epsilon \cdot w_{\max}/n$. Note that

$w(e) - \gamma < \gamma \cdot \tilde{w}(e) \leq w(e)$ for any e. It follows from the definitions that:

$$\begin{aligned}
w(M) &\geq \gamma \cdot \tilde{w}(M) && \text{Defn. of } \tilde{w} \\
&\geq \gamma \cdot (1 - \epsilon/2)\tilde{w}(M^*) && \text{Defn. of } M,\ M^* \text{ is the MWM} \\
&> (1 - \epsilon/2)(w(M^*) - \gamma n/2) && \text{Defn. of } \tilde{w},\ |M^*| \leq n/2 \\
&= (1 - \epsilon/2)(w(M^*) - \epsilon \cdot w_{\max}/2) && \text{Defn. of } \gamma \\
&> (1 - \epsilon)w(M^*) && \text{Since } w(M^*) \geq w_{\max}
\end{aligned}$$

Since it is better to use an exact MWM algorithm when $\epsilon < 1/n$, we assume, henceforth, that $w : E \to \{1, 2, \ldots, N\}$, where $N \leq n^2$ is the maximum integer edge weight.

2.3 Weighted Matching and Its LP Formulation

The maximum weight matching problem can be expressed as the following integer linear program, where x represents the incidence vector of a matching.

$$\begin{aligned}
\text{maximize} \quad & \sum_{e \in E(G)} w(e)x(e) \\
\text{subject to} \quad & 0 \leq x(e) \leq 1, x(e) \text{ an integer} && \forall e \in E(G) \qquad (2.1) \\
& \sum_{e=(u,u') \in E(G)} x(e) \leq 1 && \forall u \in V(G)
\end{aligned}$$

Let $\mathcal{V}_{odd}$ be the set of all odd subsets of $V(G)$ with at least three vertices. Clearly all solutions to (2.1) also satisfy (2.2).

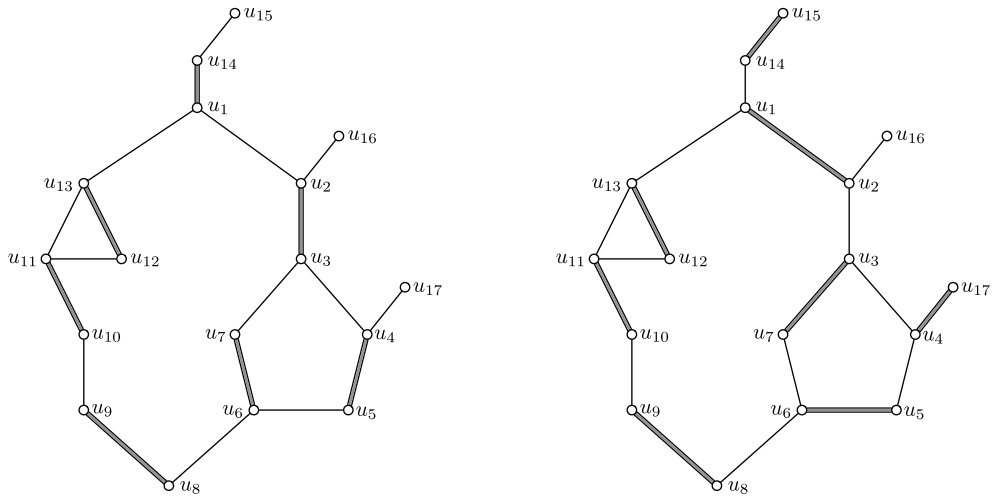
$$\sum_{e \in E(B)} x(e) \leq (|B| - 1)/2 \qquad \forall B \in \mathcal{V}_{odd} \qquad (2.2)$$

Edmonds proved that if we substitute (2.2) for the integrality requirement of (2.1), the basic feasible solutions to the resulting linear program are nonetheless integral. The dual of this linear program is as follows.

$$\begin{aligned}
\text{minimize} \quad & \sum_{u \in V(G)} y(u) + \sum_{B \in \mathcal{V}_{odd}} \frac{|B|-1}{2} \cdot z(B) \\
\text{subject to} \quad & yz(e) \geq w(e) && \forall e \in E(G) \\
& y(u) \geq 0, z(B) \geq 0 && \forall u \in V(G), \forall B \in \mathcal{V}_{odd} \\
\text{where, by definition,} \quad & yz(u,v) \stackrel{\text{def}}{=} y(u) + y(v) + \sum_{\substack{B \in \mathcal{V}_{odd}, \\ (u,v) \in E(B)}} z(B)
\end{aligned}$$

Despite the exponential number of primal constraints and dual z-variables, Edmonds showed that an optimum matching[2] could be found in polynomial time without maintaining information (z-values) on more than $n/2$ elements of $\mathcal{V}_{odd}$ at any given time. At intermediate stages of Edmonds's algorithm there is a matching M and a laminar (hierarchically nested) subset $\Omega \subseteq \mathcal{V}_{odd}$, where each element of Ω is identified with a *blossom*. Blossoms are formed inductively as follows. If $v \in V$ then the set $\{v\}$ is a trivial blossom. An odd length sequence $(A_0, A_1, \ldots, A_\ell)$ forms a nontrivial blossom $B = \bigcup_i A_i$ if the $\{A_i\}$ are blossoms and there is a sequence of edges $e_0, \ldots, e_\ell$ where $e_i \in A_i \times A_{i+1}$ (modulo $\ell + 1$) and $e_i \in M$ if and only if i is odd, that is, A_0 is incident to unmatched edges e_0, e_ℓ. See Figure 2.1. The *base* of blossom B is the base of A_0; the base of a trivial blossom is its only vertex. The set of *blossom edges* E_B are $\{e_0, \ldots, e_\ell\}$ and those used in the formation of $A_0, \ldots, A_\ell$. The set $E(B) = E \cap (B \times B)$ may, of course, include many non-blossom edges. A short proof by induction shows that $|B|$ is odd and that the base of B is the only unmatched

[2]Much of the literature deals with maximum (or minimum) weight *perfect* matchings, which requires the following modifications to the LP: $\sum_{e=(u,u') \in E(G)} x(e) = 1$ holds with equality, for $u \in V(G)$, and y is unconstrained in the dual.



(a) (b)

Figure 2.1: Thick edges are matched, thin unmatched. (a) A blossom $B_1 = (u_1, u_2, B_2, u_8, u_9, u_{10}, B_3)$ with base u_1 containing non-trivial sub-blossoms $B_2 = (u_3, u_4, u_5, u_6, u_7)$ with base u_3 and $B_3 = (u_{11}, u_{12}, u_{13})$ with base u_{11}. Vertices u_{15}, u_{16}, and u_{17} are free. The path $(u_{16}, u_2, u_3, u_7, u_6, u_5, u_4, u_{17})$ is an example of an augmenting path that exists in G but not G/B_1, the graph obtained by contracting B_1. (b) The situation after augmenting along $(u_{15}, u_{14}, B_1, u_{17})$ in G/B_1, which corresponds to augmenting along $(u_{15}, u_{14}, u_1, u_2, u_3, u_7, u_6, u_5, u_4, u_{17})$ in G. After augmentation B_1 and B_2 have their base at u_4.

vertex in the subgraph induced by B.

The set Ω of *active* blossoms is represented by rooted trees in our algorithm, where leaves represent vertices and internal nodes represent nontrivial blossoms. A *root blossom* is one not contained in any other blossom. The children of an internal node representing a blossom B are ordered by the odd cycle that formed B, where the child containing the base of B is ordered first. As we can see, it is often possible to treat blossoms as if they were single vertices. The *contracted graph* G/Ω is obtained by contracting all root blossoms and removing the edges in those blossoms. To *dissolve* a root blossom B means to delete its node in the blossom forest and, in the contracted graph, to replace B with individual vertices $A_0, \ldots, A_\ell$. Lemma 2.1 summarizes some useful properties of the contracted graph.

Lemma 2.1. *Let Ω be a set of blossoms with respect to a matching M.*

(i) If M is a matching in G then M/Ω is a matching in G/Ω.

(ii) Every augmenting path $\hat{P}$ relative to M/Ω in G/Ω extends to an augmenting path P relative to M in G. (That is, P is obtained from $\hat{P}$ by substituting for each non-trivial blossom vertex B in $\hat{P}$ a path through E_B. See Figure 2.1(a,b).)

(iii) If P is an augmenting path and P/Ω is also an augmenting path relative to M/Ω, then Ω remains a valid set of blossoms (possibly with different bases) for the augmented matching $M \oplus P$. See Figure 2.1(a,b).

(iv) The base u of a blossom $B \in \Omega$ uniquely determines a maximum cardinality matching of E_B, having size $(|B|-1)/2$. See Figure 2.1(a,b).

Implementations of Edmonds algorithm grow a matching M while maintaining Property 2.2, which controls the relationship between M, Ω and the dual variables.

Property 2.2. *(Complementary Slackness)*

(i) **(Nonnegativity of y, z)** *$z(B) \geq 0$ for all $B \in \mathcal{V}_{odd}$ and $y(u) \geq 0$ for all $u \in V(G)$.*

(ii) **(Active Blossoms)** *Ω contains all B with $z(B) > 0$ and all root blossoms B have $z(B) > 0$. (Non-root blossoms may have zero z-values.)*

(iii) **(Domination)** *$yz(e) \geq w(e)$ for all $e \in E$.*

(iv) **(Tightness)** *$yz(e) = w(e)$ when $e \in M$ or $e \in E_B$ for some $B \in \Omega$.*

If the y-values of free vertices become zero, it follows from domination and tightness that M is a maximum weight matching, as we can see from the following proof. Here M^* is any maximum weight matching.

$$
\begin{aligned}
w(M) &= \sum_{e \in M} w(e) \\
&= \sum_{e \in M} yz(e) && \text{tightness} \\
&= \sum_{u \in V(G)} y(u) + \sum_{B \in \Omega} \frac{|B|-1}{2} \cdot z(B) && \text{Note } \sum_{u \in V(G)} y(u) = \sum_{u \in V(M)} y(u) \\
&\geq \sum_{u \in V(M^*)} + \sum_{B \in \Omega} |E(B) \cap M^*| \cdot z(B) && y, z \text{ non-negative} \\
&= \sum_{e \in M^*} yz(e) \;\geq\; w(M^*) && \text{domination}
\end{aligned}
$$

2.4 A Scaling Algorithm for Approximate MWM

The algorithm maintains a *dynamic* relaxation of complementary slackness. In the beginning *domination* is weak but it becomes progressively tighter at each scale whereas *tightness* is weakened at each scale, though not uniformly. The degree to which a matched edge or blossom edge may violate tightness depends on *when* it last entered the blossom or matching. Define $\delta_0 = 2^{\lfloor \log(\epsilon' N) \rfloor}$ and $\delta_i = \delta_0/2^i$, where ϵ' will be fixed later so that the final matching is a $(1-\epsilon)$-MWM. At scale i we use the weight function $w_i(e) = \delta_i \lfloor w(e)/\delta_i \rfloor$. Note that $w_{i+1}(e) = w_i(e)$ or $w_i(e) + \delta_{i+1}$ and that $\epsilon' N/2^{i+1} < \delta_i \leq \epsilon' N/2^i$.

Property 2.3. *(Relaxed Complementary Slackness) There are $L+1$ scales numbered $0, \ldots, L$, where $L \stackrel{\text{def}}{=} \lceil \log N \rceil$. Let $i \in [0, L]$ be the current scale.*

(i) **(Granularity of y, z)** *$z(B)$ is a nonnegative multiple of δ_i, for all $B \in \mathcal{V}_{odd}$, and $y(u)$ is a nonnegative multiple of $\delta_i/2$, for all $u \in V(G)$.*

(ii) **(Active Blossoms)** *Ω contains all B with $z(B) > 0$ and all root blossoms B have $z(B) > 0$. (Non-root blossoms may have zero z-values.)*

(iii) **(Near Domination)** $yz(e) \geq w_i(e) - \delta_i$ *for all* $e \in E$.

(iv) **(Near Tightness)** *Call a matched or blossom edge* type j *if it was last made a matched or blossom edge in scale* $j \leq i$. *(That is, it entered the set* $M \cup \bigcup_{B \in \Omega} E_B$ *in scale* j *and has remained in that set, even as* M *and* Ω *change as augmenting paths are found and blossoms are created or destroyed.) If* e *is such a type* j *edge then* $yz(e) \leq w_i(e) + 2(\delta_j - \delta_i)$.

(v) **(Free Vertex Duals)** *The* y*-values of free vertices are equal and strictly less than the* y*-values of matched vertices.*

Lemma 2.4 allows us to measure the quality of a matching M, given duals y and z satisfying Property 2.3.

Lemma 2.4. *Let* M *be a matching satisfying Property 2.3 at scale* i *and let* M^* *be a maximum weight matching. Let* f *be the number of free vertices, each having* y*-value* ϕ, *and let* $\hat{\epsilon}$ *be such that* $yz(e) - w(e) \leq \hat{\epsilon} \cdot w(e)$ *for all* $e \in M$. *Then* $w(M) \geq (1 + \hat{\epsilon})^{-1}(w(M^*) - 2\delta_i|M^*| - f\phi)$. *If* $i = L$ *and* $\phi = 0$ *then* M *is a* $(1 - \epsilon' - \hat{\epsilon})$*-MWM.*

Proof. The claim follows from Property 2.3.

$$
\begin{aligned}
w(M) &= \sum_{e \in M} w(e) && \text{defn. of } w(M) \\
&\geq (1+\hat{\epsilon})^{-1} \sum_{e \in M} yz(e) && \text{near tightness, defn. of } \hat{\epsilon} \\
&= (1+\hat{\epsilon})^{-1} \left(\sum_{u \in V(M)} y(u) + \sum_{B \in \Omega} \frac{|B|-1}{2} \cdot z(B) \right) && \text{defn. of } yz \\
&\geq (1+\hat{\epsilon})^{-1} \left(\sum_{u \in V(M^*)} y(u) + \sum_{B \in \Omega} |E(B) \cap M^*| \cdot z(B) - f\phi \right) && (2.3) \\
&\geq (1+\hat{\epsilon})^{-1} \left(\sum_{e \in M^*} yz(e) - f\phi \right) && \text{defn. of } yz \\
&\geq (1+\hat{\epsilon})^{-1} \left(w_i(M^*) - f\phi - \delta_i \cdot |M^*| \right) && \text{near domination} \\
&> (1+\hat{\epsilon})^{-1} \left(w(M^*) - f\phi - 2\delta_i \cdot |M^*| \right) && \text{defn. of } w_i
\end{aligned}
$$

Inequality 2.3 follows from several facts: first, no matching can contain more than $(|B|-1)/2$ edges in B; second, $V(M^*) \backslash V(M)$ contains only free vertices (with respect to M), whose y-values are ϕ; and third, y- and z-values are nonnegative. Note that the last inequality is loose by $\delta_i |M^*|$ if $i = L$ since in that case $w_L = w$.

The integrality of edge weights implies that $w(M^*) \geq |M^*|$. If $i = L$ and $\phi = 0$ then $\delta_L = 2^{\lfloor \log(\epsilon' N) \rfloor - \lceil \log N \rceil} \leq \epsilon'$ and $w(M) \geq (1+\hat{\epsilon})^{-1}(w(M^*) - \delta_L |M^*|) \geq (1+\hat{\epsilon})^{-1}(1-\epsilon')w(M^*) > (1-\epsilon'-\hat{\epsilon})w(M^*)$, that is, M is a $(1-\epsilon'-\hat{\epsilon})$-MWM. □

2.4.1 The Scaling Algorithm

Initially $M = \emptyset, \Omega = \emptyset$, and $y(u) = N/2 - \delta_0/2$ for all $u \in V$, which clearly satisfies Property 2.3 for scale $i = 0$.

The algorithm repeatedly finds sets of augmenting paths of *eligible edges*, creates and destroys blossoms, and performs dual adjustments on y, z in order to maintain Property 2.3 and increase the number of eligible edges.

Definition 2.5. At scale i, an edge e is *eligible* if at least one of the following hold:

(i) $e \in E_B$ for some $B \in \Omega$.

(ii) $e \notin M$ and $yz(e) = w_i(e) - \delta_i$.

(iii) $e \in M$ and $yz(e) - w_i(e)$ is a nonnegative integer multiple of δ_i.

Let E_{elig} be the set of eligible edges and let $G_{elig} = (V, E_{elig})/\Omega$ be the unweighted graph obtained by discarding ineligible edges and contracting root blossoms.

Criterion (i) for eligibility simply ensures that an augmenting path in G_{elig} extends to an augmenting path of eligible edges in G. A key implication of Criteria (ii) and (iii) is that if P is an augmenting path in G_{elig}, every edge in P becomes ineligible in $(M/\Omega) \oplus P$. This follows from the fact that unmatched edges must have $yz(e) - w_i(e) < 0$ whereas matched edges must have $yz(e) - w_i(e) \geq 0$. Regarding Criterion (iii), note that Property 2.3 (granularity and near domination) implies that $w_i(e) - yz(e)$ is at least $-\delta_i$ and an integer multiple of $\delta_i/2$.

The algorithm contains $\lceil \log N \rceil + 1$ scales, and in each scale the step size δ_i of the dual adjustments shrinks by one half. In each scale, the following steps are repeated until the y-value of free vertices shrinks by about one half comparing to its value at the beginning of this scale. (The full description of this algorithm is shown in Figure 2.2.)

- First find a maximal set of augmenting paths in G_{elig}.
- Then find and shrink new blossoms. Update Ω and G_{elig}.
- Perform dual adjustments and dissolve root blossoms with zero z-values.

Initialization:

$M \leftarrow \emptyset$	no matched edges
$\Omega \leftarrow \emptyset$	no blossoms
$\delta_0 \leftarrow 2^{\lfloor \log(\epsilon' N) \rfloor}$	$\epsilon' = \Theta(\epsilon)$ a parameter
$y(u) \leftarrow \frac{N}{2} - \frac{\delta_0}{2}$, for all $u \in V(G)$	satisfies Property 2.3(iii)

Execute scales $i = 0 \ldots, L = \lceil \log N \rceil$ and return the matching M.

Scale i:

- Repeat the following steps until y-values of free vertices reach $N/2^{i+2} - \delta_i/2$, if $i \in [0, L)$, or until they reach zero, if $i = L$.

  * **Augmentation:**
    Find a maximal set Ψ of augmenting paths in G_{elig} and set $M \leftarrow M \oplus (\bigcup_{P \in \Psi} P)$. Update G_{elig}.

  * **Blossom Shrinking:**
    Let $V_{out} \subseteq V(G_{elig})$ be the vertices (that is, root blossoms) reachable from free vertices by even-length alternating paths; let Ω' be a maximal set of (nested) blossoms on V_{out}. (That is, if $(u, v) \in E(G_{elig}) \backslash M$ and $u, v \in V_{out}$, then u and v must be in a common blossom.) Let $V_{in} \subseteq V(G_{elig}) \backslash V_{out}$ be those vertices reachable from free vertices by odd-length alternating paths. Set $z(B) \leftarrow 0$ for $B \in \Omega'$ and set $\Omega \leftarrow \Omega \cup \Omega'$. Update G_{elig}.

  * **Dual Adjustment:**
    Let $\hat{V}_{in}, \hat{V}_{out} \subseteq V$ be original vertices represented by vertices in V_{in} and V_{out}. The y- and z-values for some vertices and root blossoms are adjusted:

    $y(u) \leftarrow y(u) - \delta_i/2$, for all $u \in \hat{V}_{out}$.
    $y(u) \leftarrow y(u) + \delta_i/2$, for all $u \in \hat{V}_{in}$.
    $z(B) \leftarrow z(B) + \delta_i$, if $B \in \Omega$ is a root blossom with $B \subseteq \hat{V}_{out}$.
    $z(B) \leftarrow z(B) - \delta_i$, if $B \in \Omega$ is a root blossom with $B \subseteq \hat{V}_{in}$.

    After dual adjustments some root blossoms may have zero z-values. Dissolve such blossoms (remove them from Ω) as long as they exist. Note that non-root blossoms are allowed to have zero z-values. Update G_{elig} by the new Ω.

- Prepare for the next scale, if $i \in [0, L)$:

  $\delta_{i+1} \leftarrow \delta_i/2$
  $y(u) \leftarrow y(u) + \delta_{i+1}$, for all $u \in V(G)$.

Figure 2.2: The Scaling Algorithm

2.4.2 Analysis and Correctness

Lemma 2.6. *After the Augmentation and Blossom Shrinking steps G_{elig} contains no augmenting path, nor is there a path from a free vertex to a blossom.*

Proof. Suppose there is an augmenting path P in G_{elig} after augmenting along paths in Ψ. Since Ψ is maximal, P must intersect some $P' \in \Psi$ at a vertex v. However, after the Augmentation step every edge in P' will become ineligible, so the matching edge $(v, v') \in M$ is no longer in G_{elig}, contradicting the fact that P consists of eligible edges. Since Ω' is maximal there can be no blossom reachable from a free vertex in G_{elig} after the Blossom Shrinking step. □

Lemma 2.7. *(Parity of y-values) Let $R \subseteq V(G_{elig})$ be the set of vertices reachable from free vertices by eligible alternating paths, at any point in scale i. Let $\hat{R} \subseteq V(G)$ be the set of original vertices represented by those in R. Then the y-values of $\hat{R}$-vertices have the same parity, as a multiple of $\delta_i/2$.*

Proof. Assume, inductively, that before the Blossom Shrinking step, all vertices in a common blossom have the same parity, as a multiple of $\delta_i/2$. Consider an eligible path $P = (B_0, B_1, \ldots, B_k)$ in G_{elig}, where the $\{B_j\}$ are either vertices or blossoms in Ω and B_0 is unmatched in G_{elig}. Let $(u_0, v_1), (u_1, v_2), \ldots, (u_{k-1}, v_k)$ be the G-edges corresponding to P, where $u_j, v_j \in B_j$. By the inductive hypothesis, u_j and v_j have the same parity, and whether (u_j, v_{j+1}) is matched or unmatched, Definition 2.5 implies that $yz(u_j, v_{j+1})/\delta_i$ is an integer, which implies $y(u_j)$ and $y(v_{j+1})$ have the same parity as a multiple of $\delta_i/2$. Thus, the y-values of all vertices in $B_0 \cup \cdots \cup B_k$ have the same parity as a free vertex in B_0, whose y-value is equal to every other free vertex, by Property 2.3(v). Since new blossoms are formed by eligible edges, the inductive hypothesis is maintained after the Blossom Shrinking step. It is also maintained after the Dual Adjustment step since the y-values of vertices in a common blossom are incremented or decremented together. This concludes the induction. □

Lemma 2.8. *The algorithm preserves Property 2.3.*

Proof. Property 2.3(v) (free vertex duals) is obviously maintained since only free vertices have their y-values decremented in each Dual Adjustment step. Property 2.3(ii) (active blossoms) is also maintained since all the new root blossoms found in the Blossom Shrinking step are contained in V_{out} and will have positive z-values after adjustment. Furthermore, each root blossom whose z-value drops to zero is dissolved, after Dual Adjustment. At the beginning of scale i all y- and z-values are integer multiples of $\delta_i/2$ and δ_i, respectively, satisfying Property 2.3(i) (granularity). This property is clearly maintained in each Dual Adjustment step.

It remains to show that the algorithm maintains Property 2.3(iii),(iv) (near domination, near tightness). Let $e = (u, v)$ be an arbitrary edge and i be the scale. First consider the dual adjustments made at the end of the scale; let yz and yz' be the function before and after adjustment. At the end of scale i we have $yz(e) \geq w_i(e) - \delta_i$. Each y-value is incremented by δ_{i+1} and $w_{i+1}(e) \leq w_i(e) + \delta_{i+1}$, hence $yz'(e) = yz(e) + 2\delta_{i+1} \geq w_i(e) \geq w_{i+1}(e) - \delta_{i+1}$, which preserves Property 2.3(iii). If $e \in M \cup \bigcup_{B\in\Omega} E_B$ is a type j edge, then at the end of the scale $yz(e) \leq w_i(e) + 2(\delta_j - \delta_i)$. By the same reasoning as above, $yz'(e) = yz(e) + 2\delta_{i+1} \leq w_i(e) + 2\delta_j - \delta_i \leq w_{i+1}(e) + 2(\delta_j - \delta_{i+1})$, preserving Property 2.3(iv).

If e is placed in M during an Augmentation step or it is a non-M edge placed in $\bigcup_{B\in\Omega} E_B$ during a Blossom Shrinking step then e has type i and $yz(e) = w_i(e) - \delta_i$, which satisfies Property 2.3(iv). Now consider a Dual Adjustment step. If neither u nor v is in $\hat{V}_{in} \cup \hat{V}_{out}$ or if u, v are in the same root blossom $B \in \Omega$, then $yz(e)$ is unchanged, preserving Property 2.3. The remaining cases depend on whether (u, v) is in M or not, whether (u, v) is eligible or not, and whether both $u, v \in \hat{V}_{in} \cup \hat{V}_{out}$ or not.

Case 1: $e \notin M$, $u, v \in \hat{V}_{in} \cup \hat{V}_{out}$ If e is ineligible then $yz(e) > w_i(e) - \delta_i$. However, by Lemma 2.7 (parity of y-values) we know $(yz(e) - w_i(e))/\delta_i$ is an integer, so $yz(e) \geq w_i(e)$ before adjustment and $yz(e) \geq w_i(e) - \delta_i$ after adjustment (if both $u, v \in \hat{V}_{out}$), which preserves Property 2.3(iii). If e is eligible then at least one of u, v is in $\hat{V}_{in}$, otherwise another blossom or augmenting path would have been formed, so $yz(e)$ cannot be reduced, which also preserves Property 2.3(iii).

Case 2: $e \in M$, $u, v \in \hat{V}_{in} \cup \hat{V}_{out}$ Since $u, v \in \hat{V}_{in} \cup \hat{V}_{out}$, Lemma 2.7 (parity of y-values) guarantees that $(yz(e) - w_i(e))/\delta_i$ is an integer. The only way e can be ineligible is if $yz(e) = w_i(e) - \delta_i$ and $u, v \in \hat{V}_{in}$, hence $yz(e) = w_i(e)$ after dual adjustment, which preserves Property 2.3(iii),(iv). On the other hand, if e is eligible then $u \in \hat{V}_{in}$ and $v \in \hat{V}_{out}$. It cannot be that $u, v \in \hat{V}_{out}$, otherwise e would have been included in an augmenting path or root blossom. In this case $yz(e)$ is unchanged, preserving Property 2.3(iii),(iv).

Case 3: $e \notin M$, $v \notin \hat{V}_{in} \cup \hat{V}_{out}$ If e is eligible then $u \in \hat{V}_{in}$ and $yz(e)$ will increase. If it is ineligible then $yz(e) \geq w_i(e) - \delta_i/2$ before adjustment and $yz(e) \geq w_i(e) - \delta_i$ after adjustment. In both cases Property 2.3(iii) is preserved.

Case 4: $e \in M$, $v \notin \hat{V}_{in} \cup \hat{V}_{out}$ It must be that e is ineligible, so $u \in \hat{V}_{in}$ and $yz(e) - w_i(e)$ is either negative or an odd multiple of $\delta_i/2$. If e is type j then, by Property 2.3(i),(iv) (granularity and near tightness), $yz(e) \leq w_i(e) + 2(\delta_j - \delta_i) - \delta_i/2$ before adjustment and $yz(e) \leq w_i(e) + 2(\delta_j - \delta_i)$ after adjustment, preserving Property 2.3(iv). □

Lemma 2.9. *Let $i \leq L$ be the scale index. Then*

(i) For $i < L$, all edges eligible at any time in scales 0 through i have weight at least $N/2^{i+1} + \delta_i$.

(ii) For any i, if $e \in M$ then $yz(e) \leq (1 + 4\epsilon')w(e)$.

Proof. **Part 1** The last search for augmenting paths in scale i begins when the y-values of free vertices are $N/2^{i+2}$, and strictly less than y-values of other vertices, by Property 2.3(v). An unmatched edge $e = (u, v)$ can only be eligible at this scale if $yz(e) = w_i(e) - \delta_i \leq w(e) - \delta_i$. Hence $w(e) \geq y(u) + y(v) + \delta_i \geq N/2^{i+1} + \delta_i$.

Part 2 Let e be a type j edge in M during scale i. Property 2.3(iv) states that $yz(e) - w_i(e) \leq 2(\delta_j - \delta_i)$. Since $w_i(e) \leq w(e)$ it also follows that $yz(e) - w(e) \leq 2\delta_j - 2\delta_i < 2^{\lfloor \log(\epsilon' N) \rfloor - j + 1} \leq \epsilon' N/2^{j-1}$. By part 1, a type j edge must have weight at least $N/2^{j+1} + \delta_j$, so $yz(e) - w(e) < 4\epsilon' \cdot w(e)$. □

Lemma 2.10. *After scale $L = \lceil \log N \rceil$, M is a $(1 - 5\epsilon')$-MWM.*

Proof. The final scale ends with free vertices having zero y-values. Property 2.3(iii) holds w.r.t. $\delta_L = \delta_0/2^L \leq \epsilon' N/2^L \leq \epsilon'$ and Lemma 2.9 states that $yz(e) \leq (1 + 4\epsilon')w(e)$. By Lemma 2.4 $w(M) \geq (1 - 5\epsilon')w(M^*)$. □

Theorem 2.11. *A $(1 - \epsilon)$-MWM can be computed in time $O(m\epsilon^{-1} \log N)$.*

Proof. Each Augmentation and Blossom Shrinking step takes $O(m)$ time [35, §8] using a modified depth-first search. (Finding a maximal set of augmenting paths is significantly simpler than finding a maximal set of minimum-length augmenting paths, as is done in [47, 66].) Each Dual Adjustment step clearly takes linear time. Scale $i < L = \lceil \log N \rceil$ begins with free vertices' y-values at $N/2^{i+1} - \delta_i$ and ends with them at $N/2^{i+2} - \delta_i$. Since y-values are decremented by $\delta_i/2$ in each Dual Adjustment step there are exactly $(N/2^{i+2})/(\delta_i/2) = N/(2\delta_0) < \epsilon'^{-1}$ such steps. The last inequality follows since $\delta_0 = 2^{\lfloor \log(\epsilon' N) \rfloor} > \epsilon' N/2$. The final scale begins with free vertices' y-values at $N/2^{L+1} - \delta_L$ and ends with them at zero, so there are fewer than $(N/2^{L+1})/(\delta_L/2) = (N/2^{L+1})/2^{\lfloor \log(\epsilon' N) \rfloor - (L+1)} = 2^{\log N - \lfloor \log(\epsilon' N) \rfloor} < 2\epsilon'^{-1}$ Dual Adjustment steps. Lemma 2.10 guarantees that the final matching is a $(1 - \epsilon)$-MWM for $\epsilon' = \epsilon/5$. Thus, the total running time is $O(m\epsilon^{-1} \log N)$. □

2.4.3 A Linear Time Algorithm

Our $O(m\epsilon^{-1}\log N)$-time algorithm requires few modifications to run in linear time, independent of N. In fact, the algorithm as it appears in Figure 2.2 requires no modifications at all: we only need to change the definition of *eligibility* and, in each scale, avoid scanning edges that cannot be eligible or part of augmenting paths or blossoms. From Lemma 2.9(i) it is helpful to index edges according to the first scale in which they may be eligible.

Definition 2.12. Define $\mu_i = N/2^{i+1} + \delta_i$, for $i < L$, and $\mu_L = 0$. For any edge e, define $\text{scale}(e) = i$ such that $w(e) \in [\mu_i, \mu_{i-1})$.

Definition 2.13 redefines eligibility. The differences with Definition 2.5 are underlined.

Definition 2.13. At scale i, an edge e is *eligible* if at least one of the following hold:

(i) $e \in E_B$ for some $B \in \Omega$.

(ii) $e \notin M$ and $yz(e) = w_i(e) - \delta_i$.

(iii) $e \in M$, $w_i(e) - yz(e)$ is a nonnegative integer multiple of δ_i, and $\underline{\text{scale}(e) \geq i - \gamma, \text{ where } \gamma \stackrel{\text{def}}{=} \lceil \log \epsilon'^{-1} \rceil.}$

Let E_{elig} be the set of eligible edges and let $G_{elig} = (V, E_{elig})/\Omega$ be the unweighted graph obtained by deleting ineligible edges and contracting root blossoms.

Lemma 2.14. *Using Definition 2.13 of eligibility rather than Definition 2.5, Property 2.3(i),(ii),(iii),(v) is maintained and Property 2.3(iv) (near tightness) holds in the following weaker form. Let $e \in M \cup \bigcup_{B\in\Omega} E_B$ be a type j edge with $\text{scale}(e) = i$. Then $yz(e) \leq w_k(e) + 2(\delta_j - \delta_k)$ at any scale $k \in [i, i+\gamma]$ and $yz(e) \leq w_k(e) + (3 + 3\epsilon'/2)\delta_i < (1 + 7\epsilon')w(e)$ for $k > i + \gamma$.*

Proof. In scales i through $i+\gamma$ Property 2.3(iv) is maintained as the two definitions of eligibility are the same. At the beginning of scale $i+\gamma+1$, e is no longer eligible and the y-values of free vertices are $N/2^{i+\gamma+2}-\delta_{i+\gamma+1}/2$. From this moment on, the y-values of free vertices are incremented by a total of $\sum_{l\geq i+\gamma+2}\delta_l$ (the dual adjustments following scales $i+\gamma+1$ through $\log N-1$) and decremented a total of $N/2^{i+\gamma+2}-\delta_{i+\gamma+1}/2+\sum_{l\geq i+\gamma+2}\delta_l$ (in the Dual Adjustment steps following searches for augmenting paths and blossoms). Each adjustment to a free y-value by some quantity Δ may cause $yz(e)$ to increase by 2Δ. This clearly occurs in the dual adjustments following each scale as $y(u)$ and $y(v)$ are incremented by Δ. Following a search for blossoms it may be that $u,v\in\hat{V}_{in}$, which would also cause $y(u)$ and $y(v)$ to each be incremented by Δ. Note that $y(u),y(v)$ cannot be decremented in scales $i+\gamma+1$ forward; if either were in $\hat{V}_{out}$ after a search for blossoms then e would have been eligible, which is a contradiction. Thus Property 2.3(iii) (near domination) is maintained for e. Putting this all together, it follows that from scale $k\geq i+\gamma+1$ forward,

$$
\begin{aligned}
yz(e) &\leq w_k(e)+2(\delta_j-\delta_k)+2\cdot\left(N/2^{i+\gamma+2}-\delta_{i+\gamma+1}/2+2\cdot\sum_{l\geq i+\gamma+2}\delta_l\right)\\
&< w_k(e)+2\delta_i+2\left(\epsilon' N/2^{i+2}+\tfrac{3}{2}\delta_{i+\gamma+1}\right) && j\geq i\text{, defn. of }\gamma\\
&< w_k(e)+2\delta_i+2\left(\delta_{i+1}+\tfrac{3\epsilon'}{2}\delta_{i+1}\right) && \epsilon' N/2^{i+2}<\delta_{i+1}\text{, defn. of }\gamma.\\
&= w_k(e)+(3+3\epsilon'/2)\delta_i\\
&\leq w_k(e)+(3+3\epsilon'/2)(\epsilon' N/2^i) && \delta_i\leq\epsilon' N/2^i\\
&< (1+7\epsilon')w(e) && w(e)\geq w_k(e)>N/2^{i+1},\ \epsilon'<1/3
\end{aligned}
$$

□

Lemma 2.15. *Let $e_1=(u,v)$ be an edge with* scale$(e_1)=i$ *and let $e_0=(u',u)$ and $e_2=(v,v')$ be the M-edges incident to u and v at some time after scale i. Then at least one of e_0 and e_2 exists, and its scale is at most $i+2$.*

Proof. Following the last Dual Adjustment step in scale i the y-values of free vertices are $N/2^{i+2} - \delta_i/2$. It cannot be that both u and v are free at this time, otherwise $yz(e_1) = y(u) + y(v) = N/2^{i+1} - \delta_i = \mu_i - 2\delta_i < w_i(e_1) - \delta_i$, violating Property 2.3(iii) (near domination). Thus, either u or v is matched for the remainder of the computation. If e_1 is matched the claim is trivial, so, assuming the claim is false, whenever e_0, e_2 exist we have $\text{scale}(e_0), \text{scale}(e_2) \geq i + 3$. That is, $w(e_0), w(e_2) < \mu_{i+2} = N/2^{i+3} + \delta_{i+2}$.

e_1 cannot be in a blossom without e_0 or e_2 also being in the blossom. Let $\mathcal{B}_l \subset \Omega$ be the blossoms containing e_l at a given time. The laminarity of blossoms ensures that either $\mathcal{B}_1 \subseteq \mathcal{B}_0$ or $\mathcal{B}_1 \subseteq \mathcal{B}_2$. Suppose it is the former, that is, e_0 exists and e_2 may or may not exist. Then, if the current scale is $k \geq i + 3$, by Property 2.3(iii) (near domination) $yz(e_1) = y(u) + y(v) + \sum_{B \in \mathcal{B}_1} z(B) \geq w_k(e_1) - \delta_k$. By Lemma 2.14 (near tightness) $y(u) + \sum_{B \in \mathcal{B}_1} < yz(e_0) \leq w_k(e_0) + (3 + 3\epsilon'/2)\delta_{i+3}$ and, if e_2 exists, $y(v) < yz(e_2) \leq w_k(e_2) + (3 + 3\epsilon'/2)\delta_{i+3}$. These inequalities follow from the definition of yz, the containment $\mathcal{B}_1 \subseteq \mathcal{B}_0$ and the fact that e_0 and e_2 can only be at scale $i + 3$ or higher. Without loss of generality we can assume $y(u) + \sum_{B \in \mathcal{B}_1} \geq y(v)$; note that if e_2 does not exist then $y(v) < y(u) + \sum_{B \in \mathcal{B}_1}$, by Property 2.3(v). Putting these inequalities together we have

$$
\begin{aligned}
w_k(e_1) &\leq y(u) + y(v) + \sum_{B \in \mathcal{B}_1} z(B) + \delta_k && \text{near domination} \\
&\leq 2\left(y(u) + \sum_{B \in \mathcal{B}_1} z(B)\right) + \delta_k \\
&< 2(w_k(e_0) + (3 + 3\epsilon'/2)\delta_{i+3}) + \delta_k && \text{near tightness} \\
&< 2w(e_0) + 8\delta_{i+3} && k \geq i + 3,\ \epsilon' < 1/3
\end{aligned}
$$

and therefore

$$\begin{aligned} w(e_0) &\geq \tfrac{1}{2}(w_k(e_1) - \delta_i) && 8\delta_{i+3} = \delta_i \\ &\geq N/2^{i+2} && \text{scale}(e_1) = i, w_k(e_1) \geq \mu_i = N/2^{i+1} + \delta_i \\ &> N/2^{i+3} + \delta_{i+2} = \mu_{i+2} \end{aligned}$$

This contradicts the fact that $\text{scale}(e_0) \geq i + 3$, since such edges have $w(e_0) < \mu_{i+2}$ by definition. □

Theorem 2.16. *A $(1-\epsilon)$-MWM can be computed in time $O(m\epsilon^{-1}\log\epsilon^{-1})$.*

Proof. We execute the algorithm from Figure 2.2 where G_{elig} refers to the eligible subgraph as defined in Definition 2.13. We need to prove several claims: (i) the algorithm does return a $(1-\epsilon)$-MWM for suitably chosen $\epsilon' = \Theta(\epsilon)$, (ii) the number of scales in which an edge could possibly participate in an augmenting path or blossom is $\log\epsilon^{-1} + O(1)$, and (iii) it is possible in linear time to compute the scales in which each edge must participate. Part (i) follows from Lemmas 2.4 and 2.14. Since $yz(e) \leq (1+7\epsilon')w(e)$ for any $e \in M$ (by Lemma 2.14) and $\delta_L \leq \epsilon'$, Lemma 2.4 implies that M is a $(1-\epsilon)$-MWM when $\epsilon' = \epsilon/8$.

Turning to part (ii), consider an edge e with $\text{scale}(e) = i$. By Lemma 2.9(i) e can be ignored in scales 0 through $i-1$. If $e = (u,v) \in M$, according to Definition 2.13, e will be ineligible in scales $i+\gamma+1$ through $\log N$. After scale $i+\gamma$ no augmenting path or blossom can contain e, so we can put it in the final matching and remove from consideration all edges incident to u or v. Now suppose that $e \notin M$ at the end of scale $i+\gamma+2$. Lemma 2.15 states that either u or v is incident to a matched edge e_0 with $\text{scale}(e_0) \leq i+2$, which by the argument above, will be put in the final matching. Therefore we can remove e from further consideration. Thus, to execute the algorithm we only need to consider e in scales $\text{scale}(e)$ through $\text{scale}(e)+\gamma+2$, that is, $\gamma + 3 = \lceil \log \epsilon'^{-1} \rceil + 3 \leq \log\epsilon^{-1} + 7$ scales in total.

We have narrowed our problem to that of computing scale(e) for all e. This is equivalent to computing the most significant bit (MSB(x) = $\lfloor \log_2 x \rfloor$) in the binary representation of $w(e)$. Once the MSB is known, scale(e) can be just one of two possible values. MSBs can be computed in a number of ways using standard instructions. It is trivial to extract MSB(x) after converting x to floating point representation. Fredman and Willard [31] gave an $O(1)$ time algorithm using unit time multiplication. However, we do not need to rely on floating point conversion or multiplication. In Section 2.2 we showed that without loss of generality $\log N \leq 2 \log n$. Using a negligible $O(n^\beta)$ space and preprocessing time we can tabulate the answers on $\beta \cdot \log n$-bit integers, where $\beta \leq 1$, then compute MSBs with $2\beta^{-1} = O(1)$ table lookups. □

2.4.4 Conclusion

We have given the first linear time $(1 - \epsilon)$-approximate MWM algorithm for arbitrarily small ϵ. Our result is a major improvement over the previous best linear time algorithm, which guaranteed only $(2/3 - \epsilon)$-approximations. [67, 51]. However, making our algorithm suitable for parallel computing is a major challenge. The best efficient parallel/distributed approximate MWM algorithm guarantees only 1/2-approximations. [44]. Improving the exact MWM algorithms is also a challenge for us.

CHAPTER III

Connectivity Oracle for Failure-Prone Graphs

The main result in this chapter is a new, space efficient data structure that can quickly answer connectivity queries after recovering from d *vertex failures*.[1] The recovery time is polynomial in d and $\log n$ but otherwise independent of the size of the graph. After processing the failed vertices, connectivity queries are answered in $O(d)$ time. The space used by the data structure is roughly mn^{ϵ}, for any fixed $\epsilon > 0$, where ϵ only affects the polynomial in the recovery time. The exact tradeoffs are given in Theorem 3.1. Our data structure is the first of its type. To achieve comparable query times using existing data structures we would need either $\Omega(n^d)$ space [19] or $\Omega(dn)$ recovery time [49].

It is easy to see that handling d vertex failures can be much harder than handling only d edge failures, since a vertex failure can cause the failure of as many as $n-1$ edges, which may have a large impact on the graph connectivity. First, we reduce the problem of d-edge failure recovery on a spanning forest of G to 2D range searching, that is, searching for edges reconnecting the split trees is equivalent to searching elements in rectangles in a 2D table. The time is quadratic of the number of deleted tree edges. Then we perform a "sparsification" on the spanning forest of G which restricts the degree bound of failed vertices in a set of forests when given any set of d failed

[1] This result appears in Duan and Pettie's paper "Connectivity Oracles for Failure Prone Graphs" [25] in STOC 2010.

vertices. In the complexities, there is a positive parameter c controlling the tradeoff between the space and the recovery time from vertex failures. When c becomes larger, the space becomes smaller but the recovery time gets larger. Theorem 3.1 gives a precise statement of the capabilities and time-space tradeoffs of our structure:

Theorem 3.1. *Let $G = (V, E)$ be a graph with m edges and n vertices and let $c \geq 1$ be an integer. A data structure with size $S = O(d^{1-2/c} m n^{1/c - 1/(c \log(2d))} \log^2 n)$ can be constructed in $\tilde{O}(S)$ time that supports the following operations. Given a set D of at most d failed vertices, D can be processed in $O(d^{2c+4} \log^2 n \log\log n)$ time so that connectivity queries w.r.t. the graph induced by $V \backslash D$ can be answered in $O(d)$ time.*

Overview. In Section 3.1 we present the *Euler Tour Structure*, which plays a key role in our vertex-failure oracle and can be used independently as an edge-failure oracle. In Sections 3.2 and 3.3 we define and analyze the redundant graph representation (called the *high degree hierarchy*) mentioned earlier. In Section 3.4 we provide algorithms to recover from vertex failures and answer connectivity queries.

3.1 The Euler Tour Structure

In this section we describe the *ET-structure* for handling connectivity queries avoiding multiple vertex and edge failures. When handling only d edge failures, the performance of the ET-structure is incomparable to that of Pătraşcu and Thorup [49] in nearly every respect.[2] The strength of the ET-structure is that if the graph can be covered by a low-degree tree T, the time to delete a *vertex* is a function of its degree

[2]The ET-structure is significantly faster in terms of construction time (near-linear vs. a large polynomial or exponential time) though it uses slightly more space: $O(m \log^{\epsilon} n)$ vs. $O(m)$. It handles d edge deletions exponentially faster for bounded d ($O(\log\log n)$ vs. $\Omega(\log^2 n \log\log n)$) but is slower as a function of d: $O(d^2 \log\log n)$ vs. $O(d \log^2 n \log\log n)$ time. The query time is the same for both structures, namely $O(\log\log n)$. Whereas the ET-structure naturally maintains a certificate of connectivity (a spanning tree), the Pătraşcu-Thorup structure requires modification and an additional logarithmic factor in the update time to maintain a spanning tree.

in T; incident edges not in T are deleted implicitly. We prove Theorem 3.2 in the remainder of this section.

Theorem 3.2. *Let $G = (V, E)$ be a graph, with $m = |E|$ and $n = |V|$, and let $\mathcal{F} = \{T_1, \ldots, T_t\}$ be a set of vertex disjoint trees in G. (The T_i's do not necessarily span a connected component of G.) There is a data structure $\mathbf{ET}(G, \mathcal{F})$ occupying space $O(m \log^\epsilon n)$ (for any fixed $\epsilon > 0$) that supports the following operations. Suppose D is a set of failed edges, of which d are tree edges in $\mathcal{F}$ and d' are non-tree edges. Deleting D splits some subset of the trees in $\mathcal{F}$ into at most $2d$ trees $\mathcal{F}' = \{T'_1, \ldots, T'_{2d}\}$. In $O(d^2 \log\log n + d')$ time we can report which pairs of trees in $\mathcal{F}'$ are connected by an edge in $E \backslash D$. In $O(\min\{\log\log n, \log d\})$ time we can determine which tree in $\mathcal{F}'$ contains a given vertex.*

Our data structure uses as a subroutine Alstrup et al.'s data structure [2] for range reporting on the integer grid $[U] \times [U]$. They showed that given a set of N points, there is a data structure with size $O(N \log^\epsilon N)$, where $\epsilon > 0$ is fixed, such that given $x, y, w, z \in [U]$, the set of points in $[x, y] \times [w, z]$ can be reported in $O(\log\log U + k)$ time, where k is the number of reported points. Moreover, the structure can be built in $O(N \log N)$ time.

For a tree T, let $L(T)$ be a list of its vertices encountered during an Euler tour of T (an undirected edge is treated as two directed edges), where we only keep the *first* occurrence of each vertex. One may easily verify that removing f edges from T partitions it into $f + 1$ connected subtrees and splits $L(T)$ into at most $2f + 1$ intervals, where the vertices of a connected subtree are the union of some subset of the intervals. To build $\mathbf{ET}(G = (V, E), \mathcal{F})$ we build the following structure for each pair of trees $(T_1, T_2) \in \mathcal{F} \times \mathcal{F}$; note that T_1 and T_2 may be the same. Let m' be the number of edges connecting T_1 and T_2. Let $L(T_1) = (u_1, \ldots, u_{|T_1|})$, $L(T_2) = (v_1, \ldots, v_{|T_2|})$, and let $U = \max\{|T_1|, |T_2|\}$. We define the point set $P \subseteq [U] \times [U]$ to be $P = \{(i, j) \mid (u_i, v_j) \in E\}$. Suppose D is a set of edge failures including

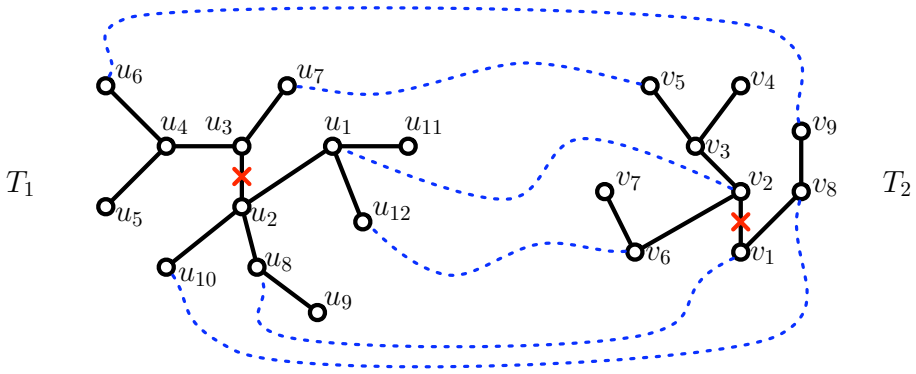
d_1 edges in T_1, d_2 in T_2, and d' non-tree edges. Removing D splits T_1 and T_2 into d_1+d_2+2 connected subtrees and partitions $L(T_1)$ into a set $I_1 = \{[x_i, y_i]\}_i$ of $2d_1+1$ intervals and $L(T_2)$ into a set $I_2 = \{[w_i, z_i]\}_i$ of $2d_2+1$ intervals. For each pair i, j we query the 2D range reporting data structure for points in $[x_i, y_i] \times [w_j, z_j] \cap P$. However, we stop the query the moment it reports some point corresponding to a non-failed edge, i.e., one in $E \backslash D$. Since there are $(2d_1+1) \times (2d_2+1)$ queries and each failed edge in D can only be reported in *one* such query, the total query time is $O(d_1 d_2 \log\log U + |D|) = O(d_1 d_2 \log\log n + d')$. See Figure 3.1 for an illustration.

The space for the data structure (restricted to T_1 and T_2) is $O(|T_1| + |T_2| + m' \log^{\epsilon} n)$. We can assume without loss of generality[3] that $|T_1| + |T_2| < 4m'$, so the space for the ET-structure on T_1 and T_2 is $O(m' \log^{\epsilon} n)$. Since each non-tree edge only appears in one such structure the overall space for $\mathbf{ET}(G, \mathcal{F})$ is $O(m \log^{\epsilon} n)$. For the last claim of the Theorem, observe that if a vertex u lies in an original tree $T_1 \in \mathcal{F}$, we can determine which tree in $\mathcal{F}'$ contains it by performing a predecessor search over the left endpoints of intervals in I_1. This can be accomplished in $O(\min\{\log\log n, \log d_1\})$ query time using a van Emde Boas tree [62] or sorted list, whichever is faster.

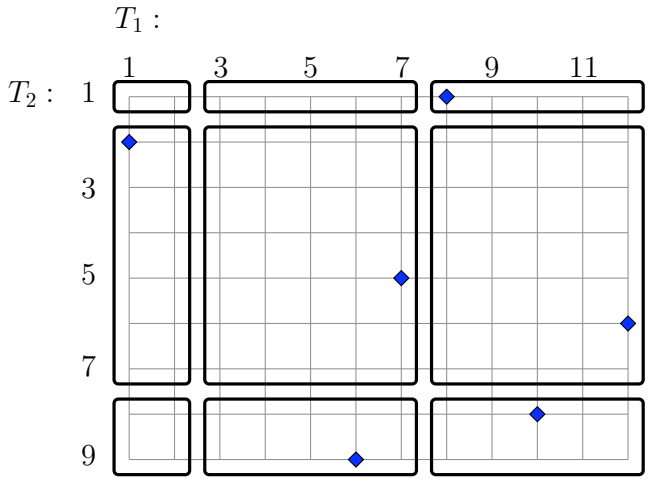
Corollary 3.3 demonstrates how $\mathbf{ET}(G, \cdot)$ can be used to answer connectivity queries avoiding edge and vertex failures.

Corollary 3.3. *The data structure* $\mathbf{ET}(G = (V, E), \{T\})$*, where* T *is a spanning tree of* G*, supports the following operations. Given a set* $D \subset E$ *of edge failures,* D *can be processed in* $O(|D|^2 \log\log n)$ *time so that connectivity queries in the graph* $(V, E \backslash D)$

[3]The idea is to remove irrelevant vertices and contract long paths of degree-2 vertices. More formally: let $V_1 \subseteq V(T_1)$ be those vertices incident to one of the m' non-tree edges. We can replace T_1 by an equivalent tree $\tilde{T}_1$ with less than $2m'$ vertices via the following steps: (1) Let T_1' be the minimal subtree of T_1 in which V_1 remains connected, then (2) Let $\tilde{V_1}$ be the union of V_1 and all branching vertices, i.e., those with degree at least 3, in T_1' (note $|\tilde{V_1}| < 2|V_1|$), then (3) Let $\tilde{T_1} = (\tilde{V_1}, \tilde{E_1})$, where $(u, v) \in \tilde{E_1}$ if there is a path $(u, \ldots, v)$ in T_1', none of whose interior vertices are in $\tilde{V_1}$. The removal of an edge from T_1 can clearly be simulated by removing an edge from $\tilde{T_1}$. To determine *which* edge in $\tilde{T_1}$ we only need to perform a predecessor search over $\tilde{V_1}$. Using a van Emde Boas tree, such queries can be answered in $O(\log\log|T_1|) = O(\log\log n)$ time. We only need to perform $d_1 + d_2$ such queries, the cost of which is dominated by the $\Omega(d_1 d_2 \log\log n)$ time for 2D range reporting.



(A)



(B)

Figure 3.1: **(A)** Here T_1 and T_2 are two trees and $L(T_1) = (u_1, \ldots, u_{12})$ and $L(T_2) = (v_1, \ldots, v_9)$ are their vertices, listed by their first appearance in some Euler tours of T_1 and T_2. (It does not matter which Euler tour we pick.) There are six non-tree edges connecting T_1 and T_2, marked by dashed curves. If the edges (u_2, u_3) and (v_1, v_2) are removed, T_1 and T_2 are split into four subtrees, say T_1', T_2', T_3', T_4', and both $L(T_1)$ and $L(T_2)$ are split into three intervals, namely $X_1 = (u_1, u_2), X_2 = (u_3, \ldots, u_7), X_3 = (u_8, \ldots, u_{12}), Y_1 = (v_1), Y_2 = (v_2, \ldots, v_7)$, and $Y_3 = (v_8, v_9)$. Each tree T_i' is identified with some subset of the intervals: $T_1', \ldots, T_4'$ are identified with $\{X_1, X_3\}, \{X_2\}, \{Y_1, Y_3\}$, and $\{Y_2\}$. **(B)** The point (i, j) (marked by a diamond) is in our point set if (v_i, u_j) is a non-tree edge. To determine if, for example, T_1' and T_4' are connected by an edge, we perform two 2D range queries, $X_1 \times Y_2$ and $X_3 \times Y_2$, and keep at most one point (i.e., a non-tree edge) for each query. In general, removing d_1 edges from T_1 and d_2 edges from T_2 necessitates $(2d_1 + 1)(2d_2 + 1)$ 2D range queries to determine incidences between all pairs of subtrees. In this example we require nine 2D range queries, indicated by boxes in the point set diagram.

can be answered in $O(\min\{\log\log n, \log|D|\})$ *time. If* $D \subset V$ *is a set of vertex failures, the update time is* $O((\sum_{v\in D}\deg_T(v))^2 \log\log n)$ *(note, this is independent of* $\sum_{v\in D}\deg(v)$*) and the query time is* $O(\min\{\log\log n, \log(\sum_{v\in D}\deg_T(v))\})$.

Proof. Let d be the number of failed edges in T (or edges in T incident to failed vertices). Using $\mathbf{ET}(G, \{T\})$ we split T into $d+1$ subtrees and $L(T)$ into a set I of $2d+1$ connected intervals, in which each connected subtree is made up of some subset of the intervals. Using $O(d^2)$ 2D range queries, in $O(d^2 \log\log n + |D|)$ time we find at most one edge connecting each pair in $I \times I$. In $O(d^2)$ time we find the connected components[4] of $V\backslash D$ and store with each interval a representative vertex from its component. To answer a query (u, v) we only need to determine which subtree u and v are in, which involves two predecessor queries over the left endpoints of intervals in I. This takes $O(\min\{\log\log n, \log d\})$ time. □

3.2 Constructing the High-Degree Hierarchy

Theorem 3.2 and Corollary 3.3 demonstrate that given a spanning tree T with maximum degree t, we can processes d vertex failures in time roughly $(dt)^2$. However, there is no way to bound t as a function of d. Our solution is to build a *high-degree hierarchy* that represents the graph in a redundant fashion so that given d vertex failures, in some representation of the graph all failed vertices have low (relevant) degree.

3.2.1 Definitions

Let $\deg_H(v)$ be the degree of v in the graph H and let $\text{High}(H) = \{v \in V(H) \mid \deg_H(v) > s\}$ be the set of *high degree* vertices in H, where $s = \Omega(d^2)$ is a fixed parameter of the construction and d is an *upper bound* on the number of vertex

[4]This involves performing a depth first search of the graph whose vertices correspond to intervals in I.

failures. Increasing s will increase the update time and decrease the space.

We assign arbitrary distinct weights to the edges of the input graph $G = (V, E)$, which guarantees that every subgraph has a unique minimum spanning forest. Let X and Y be arbitrary subsets of vertices. We define F_X to be the minimum spanning forest of the graph $G \backslash X$. (The notation $G \backslash X$ is short for "the graph induced by $V \backslash X$.") Let $F_X(Y)$ to be the subforest of F_X that preserves the connectivity of $Y \backslash X$, i.e., an edge appears in $F_X(Y)$ if it is on the path in F_X between two vertices in $Y \backslash X$. If X is omitted it is $\emptyset$. Note that $F_X(Y)$ may contain branching vertices (having degree greater than 2) that are not in $Y \backslash X$.

Lemma 3.4. *For any vertex sets X, Y, $|\text{High}(F_X(Y))| \leq \lfloor \frac{|Y \backslash X| - 2}{s-1} \rfloor$.*

Proof. Note that all leaves of $F_X(Y)$ belong to $Y \backslash X$. We prove by induction that the maximum number of vertices with degree at least $s+1$ (the threshold for being high degree) in a tree with l leaves is precisely $\lfloor (l-2)/(s-1) \rfloor$. This upper bound holds whenever there is one internal vertex, and is clearly tight when $l \leq s+1$. Given a tree with $l > s+1$ leaves and at least two internal vertices, select an internal vertex v adjacent to exactly one internal vertex and a maximum number of leaves. If v is incident to fewer than s leaves it can be spliced out without decreasing the number of high-degree vertices, so assume the number of incident leaves is at least s. Trimming the adjacent leaves of v leaves a tree with a net loss of $s-1$ leaves and 1 high degree vertex. The claim then follows from the inductive hypothesis. □

3.2.2 The Hierarchy Tree and Its Properties

Definition 3.5 describes the hierarchy tree, and in fact shows that it is constructible in roughly linear time per hierarchy node. See Figure 3.2 for an explanatory diagram.

Definition 3.5. The *hierarchy tree* is a rooted tree that is uniquely determined by the graph $G = (V, E)$, its artificial edge weights, and the parameters d and s. Nodes

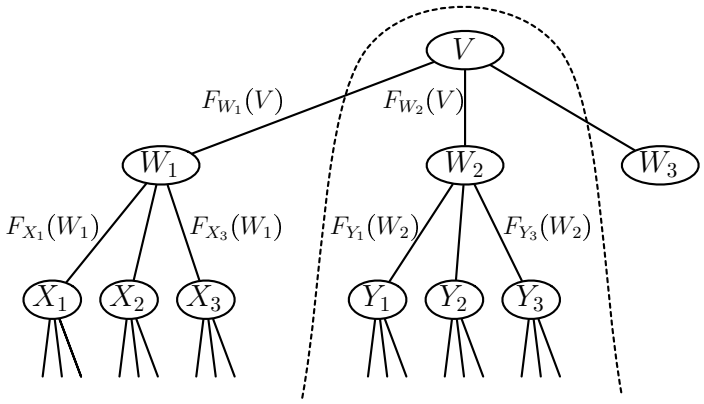


Figure 3.2: After W_1, W_2 and all their descendants have been constructed we construct W_3 as follows. First, include all members of W_2 in W_3. Second, look at *all* hierarchy edges (X', U') where X' is in W_2's subtree and U' is the parent of X' (i.e., all edges under the dashed curve), and include all the high degree vertices in $F_{X'}(U')$ in W_3. In this example W_3 includes $\text{High}(F_{W_2}(V)), \text{High}(F_{Y_1}(W_2)), \text{High}(F_{Y_2}(W_2)), \text{High}(F_{Y_3}(W_2))$, and so on.

in the tree are identified with subsets of V. The root is V and every internal node has precisely d children. A (not necessarily spanning) forest of G is associated with each node and each edge in the hierarchy tree. The tree is constructed as follows:

(i) Let W be a node with parent U. We associate the forest $F(U)$ with U and $F_W(U)$ with the edge (U, W).

(ii) If $F(U)$ has no high degree vertices then U is a leaf; otherwise it has children $W_1, \ldots, W_d$ defined as follows. (Subtree(X) is the set of descendants of X in the hierarchy, including X.)

$$
\begin{aligned}
W_1 &= \text{High}(F(U)) \\
W_i &= W_{i-1} \cup \left[U \cap \left(\bigcup_{\substack{W' \in \text{ Subtree}(W_{i-1}) \\ U' = \text{ parent of } W'}} \text{High}(F_{W'}(U')) \right) \right]
\end{aligned}
$$

In other words, W_i inherits all the vertices from W_{i-1} and adds all vertices that are both in U and high-degree in some forest associated with an edge (W', U'),

where W' is a descendant of W_{i-1}. Note that this includes the forest $F_{W_{i-1}}(U)$.

It is regretful that Definition 3.5((ii)) is so stubbornly unintuitive. We do not have a clean justification for it, except that it guarantees all the properties we require of the hierarchy: that it is small, shallow, and effectively represents the graph in many ways so that given d vertex failures, failed vertices have low degree in some graph representation. After establishing Lemmas 3.6–3.8, Definition 3.5((ii)) does not play any further role in the data structure whatsoever. Proofs of Lemmas 3.6 and 3.7 appear in the appendix.

Lemma 3.6. (Containment of Hierarchy Nodes) *Let U be a node in the hierarchy tree with children $W_1, \ldots, W_d$. Then* $\text{High}(F(U)) \subseteq U$ *and* $W_1 \subseteq \cdots \subseteq W_d \subseteq U$.

Proof. The second claim will be established in the course of proving the first claim. We prove the first claim by induction on the preorder (depth first search traversal) of the hierarchy tree. For the root node V, $\text{High}(V)$ is trivially a subset of V. Let W_i be a node, U be its parent, and W_1 be U's first child, which may be the same as W_i. Suppose the claim is true for all nodes preceding W_i. If it is the case that $W_i = W_1$, we have that $W_1 = \text{High}(F(U))$ (by Definition 3.5((ii))) and $\text{High}(F(U)) \subseteq U$ (by the inductive hypothesis). Since $F(W_1)$ is a subforest of $F(U)$ (this follows from the fact that for a vertex set Y we select $F(Y)$ to be the *minimum* forest spanning Y), every high degree node in $F(W_1)$ also has high degree in $F(U)$, i.e., $\text{High}(F(W_1)) \subseteq \text{High}(F(U)) = W_1$, which establishes the claim when $W_i = W_1$. Once we know that $W_1 \subseteq U$ it follows from Definition 3.5((ii)) that $W_1 \subseteq \cdots \subseteq W_d \subseteq U$. By the same reasoning as above, when $W_i \neq W_1$, we have that $W_i \subseteq U$, implying that $F(W_i)$ is a subforest of $F(U)$, which implies that $\text{High}(F(W_i)) \subseteq \text{High}(F(U)) = W_1 \subseteq W_i$. □

Lemma 3.7. (Hierarchy Size and Depth) *Consider the hierarchy tree constructed with high-degree threshold $s = (2d)^{c+1} + 1$, for some integer $c \geq 1$. Then:*

(i) The depth of the hierarchy is at most $k = \lceil \log_{(s-1)/2d} n \rceil \leq \lceil (\log n)/(c \log(2d)) \rceil$.

(ii) The number of nodes in the hierarchy is on the order of $d^{-2/c}n^{1/c-1/(c\log 2d)}$.

Proof. We prove Parts (1) and (2) by induction over the postorder of the hierarchy tree. In the base case U is a leaf, (1) is vacuous and (2) is trivial, since there is one summand, namely $|\text{High}(F_U(p(U)))|$, which is at most $(|p(U)|-2)/(s-1)$ by Lemma 3.4. For Part (1), in the base case $|W_1| < |U|/(s-1)$. For $i \in [2,d]$ we have:

$$\begin{aligned}
|W_i| &\le |W_{i-1}| + \sum_{X \in \text{Subtree}(W_{i-1})} |\text{High}(F_X(p(X)))| \\
&\le \frac{2(i-1)|U|}{s-1} + \frac{2|U|}{s-1} \ \{\text{Ind. hyp. (1) and (2)}\} \\
&= \frac{2i|U|}{s-1}
\end{aligned}$$

For Part (2) we have:

$$\begin{aligned}
&\sum_{X \in \text{Subtree}(U)} |\text{High}(F_X(p(X)))| \\
&= |\text{High}(F_U(p(U)))| + \sum_{i=1}^{d} \sum_{X \in \text{Subtree}(W_i)} |\text{High}(F_X(p(X)))| \quad \{\text{Defn. of Subtree}\} \\
&< \frac{|p(U)|}{s-1} + \frac{2d|U|}{s-1} \ \{\text{Lemma 3.4, Ind. hyp. (2)}\} \\
&\le \frac{|p(U)|}{s-1} + \frac{2d[2d|p(U)|/(s-1)]}{s-1} \ \{\text{Ind. hyp. (1)}\} \\
&\le \frac{2|p(U)|}{s-1} \ \{s \ge 4d^2+1\}
\end{aligned}$$

We prove Part (3) for a slight modification of the hierarchy tree in which U is forced to be a leaf if $|U| \le 2ds$. This change has no effect on the running time of the algorithm.[5] Consider the set of intervals $\{B_j\}$ where $B_j = [(2d)^j, (2d)^{j+1})$, and let l_j be the maximum number of leaf descendants of a node U for which $|U| \in B_j$. If $|U| \le (2d)s$ then U is a leaf, i.e., $l_j = 1$ for $j \le c+1$. Part (1) implies that if $|U|$ lies in

[5] We only require that in a leaf node U, any set of d failed vertices are incident to a total of $O(ds)$ tree edges from $F(U)$, i.e., that the average degree in $F(U)$ is $O(s)$. We do not require that *every* failed vertex be low degree in $F(U)$.

B_j then each child lies in either B_{j-c-1} or B_{j-c}. Hence, $l_j \leq d \cdot l_{j-c}$, and, by induction, $l_j \leq d^{\lfloor (j-2)/c \rfloor}$. Now suppose that n lies in the interval $[(2d)^{cx+2}, (2d)^{(c+1)x+2}) = B_{cx+2} \cup \cdots \cup B_{(c+1)x+1}$. Then the number of leaf descendants of V, the hierarchy tree root, is at most $d^c < n^{1/c} 2^{-x} d^{-2/c} \leq n^{1/c - 1/(c \log(2d))} d^{-2/c}$. □

For the remainder of the chapter the variable k is fixed, as defined above. Aside from bounds on its size and depth, the only other property we require from the hierarchy tree is that, for any set of d vertex failures, all failures have low degree in forests along *some* path in the hierarchy. More formally:

Lemma 3.8. (The Hierarchy's Low-Degree Property) *For any set D of at most d failed vertices, there exists a path $V = U_0, U_1, ..., U_p$ in the hierarchy tree such that all vertices in D have low degree in the forests $F_{U_1}(U_0), \ldots, F_{U_p}(U_{p-1}), F(U_p)$. Furthermore, this path can be found in $O(d(p+1)) = O(dk)$ time.*

Proof. We construct the path $V = U_0, U_1, \ldots$ one node at a time using the following procedure.

1. $U_0 \leftarrow V$
2. For i from 1 to ∞ : {
3. If U_{i-1} is a leaf set $p \leftarrow i-1$ and HALT. (I.e., $U_{i-1} = U_p$ is the last node on the path.)
4. Let $W_1, \ldots, W_d$ be the children of U_{i-1} and artificially define $W_0 = \emptyset$ and $W_{d+1} = W_d$.
5. Let $j \in [0, d]$ be minimal such that $D \cap (W_{j+1} \backslash W_j) = \emptyset$.
6. If $j = 0$ set $p \leftarrow i-1$ and HALT. (I.e., $U_{i-1} = U_p$ is the last node on the path.)
7. Otherwise $U_i \leftarrow W_j$
8. }

First let us note that in Line 5 there always exists such a j, since we defined the artificial set $W_{d+1} = W_d$, and that this procedure eventually halts since the hierarchy tree is finite. If, during the construction of the hierarchy, we record for each $v \in U_{i-1}$

the *first* child of U_{i-1} in which v appears, Line 5 can easily be implemented in $O(d)$ time, for a total of $O((p+1)d) = O(dk)$ time.

Define $D_i = D \cap U_i$. It follows from Lemma 3.6 that $U_0 \supseteq \cdots \supseteq U_p$ and therefore that $D = D_0 \supseteq \cdots \supseteq D_p$. In the remainder of the proof we will show that:

(A) When the procedure halts, in Line 3 or 6, D is disjoint from $\text{High}(F(U_p))$.

(B) For each $i \in [1, p]$, $D_{i-1} \backslash D_i$ is disjoint from $\text{High}(F_{U_{i'}}(U_{i'-1}))$, for $i' \in [i, p]$.

Regarding (B), notice that for $i' \in [1, i)$, $D_{i-1} \backslash D_i$ is trivially disjoint from $\text{High}(F_{U_{i'}}(U_{i'-1}))$ because vertices in $D_{i-1} \backslash D_i \subseteq U_{i-1} \subseteq U_{i'}$ are specifically excluded from $F_{U_{i'}}(U_{i'-1})$. Thus, the lemma will follow directly from (A) and (B).

Proof of (A) Suppose the procedure halts at Line 3, i.e., $U_{i-1} = U_p$ is a leaf. By Definition 3.5((ii)), $\text{High}(F(U_p)) = \emptyset$ and is trivially disjoint from D. The procedure would halt at Line 6 if $j = 0$, meaning $W_1 \backslash W_0 = W_1$ is disjoint from D, where W_1 is the first child of $U_{i-1} = U_p$. This implies $\text{High}(F(U_p))$ is also disjoint from D since $W_1 = \text{High}(F(U_p))$ *by definition.*

Proof of (B) Fix an $i \in [1, p]$ and let $W_j = U_i$ be the child of U_{i-1} selected in Line 5. We first argue that if $j = d$ there is nothing to prove, then deal with the case $j \in [1, d-1]$. If $j = d$ that means the d disjoint sets $W_1, W_2 \backslash W_1, \ldots, W_d \backslash W_{d-1}$ each intersect D, implying that $U_i = W_d \supseteq D$ and therefore $D_i = D$. Thus $D_{i-1} \backslash D_i = \emptyset$ is disjoint from any set. Consider now the case when $j < d$, i.e., the node W_{j+1} exists and $W_{j+1} \backslash W_j$ is disjoint from D. By Definition 3.5((ii)) and the fact that $U_i, \ldots, U_p$ are descendants of $W_j = U_i$, we know that W_{j+1} includes all the high-degree vertices in $F_{U_i}(U_{i-1}), \ldots, F_{U_p}(U_{p-1})$ that are also in U_{i-1}. By definition, $D_{i-1} \backslash D_i$ is contained in U_{i-1} and disjoint from $U_i, \ldots, U_p$, implying that no vertex in $D_{i-1} \backslash D_i$ has high-degree in $F_{U_i}(U_{i-1}), \ldots, F_{U_p}(U_{p-1})$. If one did, it would have been put in W_{j+1} (as dictated by Definition 3.5((ii))) and $W_{j+1} \backslash W_j$ would not have been disjoint from D, contradicting the choice of j. □

3.3 Inside the Hierarchy Tree

Lemma 3.8 guarantees that for any set D of d vertex failures, there exists a path of hierarchy nodes $V = U_0, \ldots, U_p$ such that all failures have low degree in the forests $F_{U_1}(U_0), \ldots, F_{U_p}(U_{p-1}), F(U_p)$. Using the ET-structure from Section 3.1 we can delete the failed vertices and reconnect the disconnected trees in $O(d^2 s^2 \log\log n)$ time for each of the $p+1$ levels of forests. This will allow us to quickly answer connectivity queries *within one level*, i.e., whether two vertices are connected in the subgraph induced by $V(F_{U_{i+1}}(U_i)) \backslash D$. However, to correctly answer connectivity queries we must consider paths that traverse many levels.

Our solution, following an idea of Chan et al. [10], is to augment the graph with *artificial* edges that capture the fact that vertices at one level (say in $U_i \backslash U_{i+1}$) are connected by a path whose intermediate vertices come from lower levels, in $V \backslash U_i$. We do not want to add too many artificial edges, for two reasons. First, they take up space, which we want to conserve, and second, after deleting vertices from the graph some artificial edges may become invalid and must be removed, which increases the recovery time. (In other words, an artificial edge (u, v) between $u, v \in U_i \backslash U_{i+1}$ indicates a u-to-v path via $V \backslash U_i$. If $V \backslash U_i$ suffers vertex failures then this path may no longer exist and the edge (u, v) is presumed invalid.) We add artificial edges so that after d vertex failures, we only need to remove a polynomial (in d, s, and $\log n$) number of artificial edges.

3.3.1 Stocking the Hierarchy Tree with ET-Structures

The data structure described in this section (as well as all notation) are for a *fixed* path $V = U_0, \ldots, U_p$ in the hierarchy tree. In other words, for *each* path from the root to a descendant in the hierarchy we build a *completely distinct* data structure. In order to have a uniform notation for the forests at each level we artificially define $U_{p+1} = \emptyset$, so $F(U_p) = F_{U_{p+1}}(U_p)$. For $i > j$ we say vertices in $U_i \backslash U_{i+1}$ are at a *higher*

level than those in $U_j \backslash U_{j+1}$ and say the trees in the forest $F_{U_{i+1}}(U_i)$ are at a higher level than those in $F_{U_{j+1}}(U_j)$. Remember that $F_{U_{i+1}}(U_i)$ is the minimum spanning forest connecting $U_i \setminus U_{i+1}$ in the graph $G \backslash U_{i+1}$ and may contain vertices at lower levels. (See Fig. 3.3) We distinguish these two types of vertices:

Definition 3.9. (Major Vertices) The *major vertices* in a tree T in $F_{U_{i+1}}(U_i)$ are those that are also in $U_i \setminus U_{i+1}$. Let $T(u)$ be the *unique* tree in $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ in which u is a major vertex.

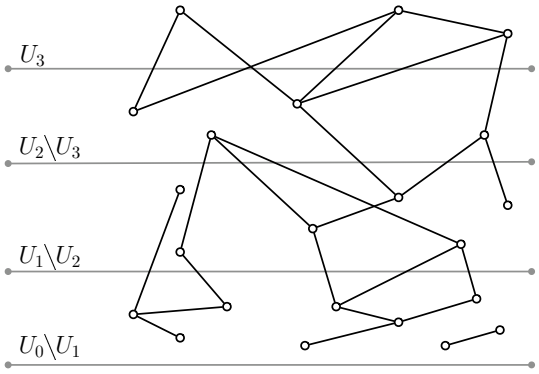
It is not clear, a priori, that the trees in $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ have any coherent organization. Lemma 3.11 shows that they naturally form a hierarchy, with trees in $F_{U_{p+1}}(U_p)$ on top. Below we give the definition of *ancestry* between trees and show each tree has exactly one ancestor at each higher level. See Figure 3.3 for an illustration of Definitions 3.9 and 3.10.

Definition 3.10. (Ancestry Between Trees) Let $0 \leq j \leq i \leq p$ and let T and T' be trees in $F_{U_{j+1}}(U_j)$ and $F_{U_{i+1}}(U_i)$, respectively. Call T' an *ancestor* of T (and T a descendant of T') if T and T' are in the same connected component in the graph $G \backslash U_{i+1}$. Notice that T is both an ancestor and descendant of itself.

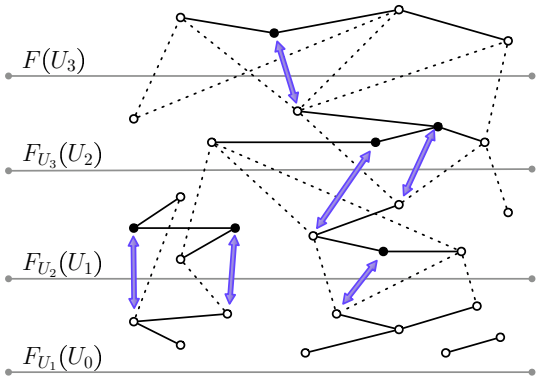
Lemma 3.11. (Unique Ances.) *Each tree T in $F_{U_{j+1}}(U_j)$ has at most one ancestor in $F_{U_{i+1}}(U_i)$, for $j \leq i \leq p$.*

Proof. Suppose T has two ancestors T_1 and T_2 in $F_{U_{i+1}}(U_i)$, i.e., T_1 and T_2 span connected components in $G \backslash U_{i+1}$. Since they are *both* connected to T in $G \backslash U_{i+1}$ (which contains T since $U_{i+1} \subseteq U_{j+1}$), T_1 and T_2 are connected in $G \backslash U_{i+1}$ and cannot be distinct trees in $F_{U_{i+1}}(U_i)$. □

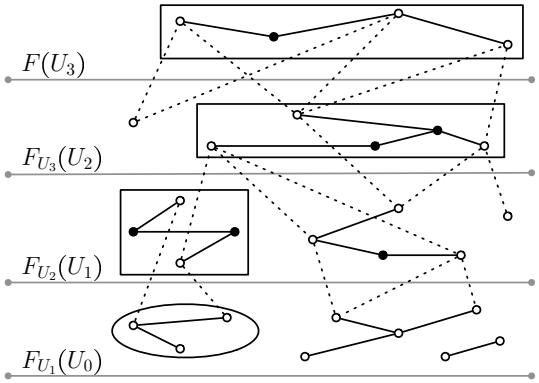
Observe that the ancestry relation between trees T in $F_{U_{j+1}}(U_j)$ and T' in $F_{U_{i+1}}(U_i)$ is the *reverse* of the ancestry relation between the nodes U_j and U_i in the hierarchy tree! That is, if $j < i$, T' is an ancestor of T but U_j is an ancestor of U_i in the hierarchy tree.



(A)



(B)



(C)

Figure 3.3: **(A)** A path $U_0, \ldots, U_3$ in the hierarchy tree (where $V = U_0$ is the root) naturally partitions the vertices into four levels $U_0 \backslash U_1, U_1 \backslash U_2, U_2 \backslash U_3$, and U_3. **(B)** The forest $F_{U_{i+1}}(U_i)$ may contain "copies" of vertices from lower levels. (Hollow vertices are *major* vertices at their level; solid ones are copies from a lower level. Thick arrows associate a copy with its original major vertex.) **(C)** A tree T in $F_{U_{j+1}}(U_j)$ is a descendant of T' in $F_{U_{i+1}}(U_i)$ (where $j \leq i$) if T and T' are connected in $G \backslash U_{i+1}$. The tree inscribed in the oval is a descendant of those trees inscribed in rectangles.

Definition 3.12. (Descendant Sets) Let $\Delta(T) = \{v \mid T(v) \text{ is a descendant of } T\}$ be the *descendent set* of a tree T. Equivalently, if T is in, say, $F_{U_{i+1}}(U_i)$, then $\Delta(T)$ is the set of vertices in the connected component of $G \backslash U_{i+1}$ containing T.

Lemma 3.13 is a simple consequence of the definitions of ancestry and descendant set, and one that will justify the way we augment the graph with artificial edges.

Lemma 3.13. (Paths and Unique Descendant Sets) *Consider a path between two vertices u and v and let w be an intermediate vertex (i.e., not u or v) with highest level. Then all intermediate vertices are in $\Delta(T(w))$ and each of $T(u)$ and $T(v)$ is either an ancestor or descendant of $T(w)$.*

Proof. This follows immediately from the definition of $\Delta(\cdot)$. □

Now that we have notions of ancestry and descendent sets, we are almost ready to describe exactly how we generate artificial edges. Recall that we are dealing with a fixed path $V = U_0, \ldots, U_p$ in the hierarchy tree. We construct two graphs H_1 and H_2 on the forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$, where the forests are regarded as having *disjoint* vertex sets. In other words, each vertex from the original graph could have $p + 1$ copies in H_1 and H_2, but only one copy is a major vertex in its forest. The graph H_1 includes the forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ and all the original graph edges. More precisely:

Definition 3.14. (The Graph H_1) The vertex set of H_1 is the union of the (disjoint) vertex sets of $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$. The edge set of H_1 consists of the tree edges in $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ and, for each edge (u, v) in the original graph, an edge connecting the major copies of u and v.

Before defining H_2 we need to introduce some additional concepts. A d-adjacency list is essentially a path that is augmented to be resilient (in terms of connectivity) to up to d vertex failures.

Definition 3.15. (d-Adjacency List) Let $L = (v_1, v_2, \ldots, v_r)$ be a list of vertices and $d \geq 1$ be an integer. The *d-adjacency edges* $\Lambda_d(L)$ connect all vertices at distance at most $d+1$ in the list L:

$$\Lambda_d(L) = \{(v_i, v_j) \mid 1 \leq i < j \leq r \text{ and } j - i \leq d+1\}$$

Before proceeding we state some simple properties of d-adjacency lists.

Lemma 3.16. (Properties of d-Adjacency Lists) *The following properties hold for any vertex list L:*

(i) *$\Lambda_d(L)$ contains fewer than $(d+1)|L|$ edges.*

(ii) *If a set D of at most d vertices are removed from L then the subgraph of $\Lambda_d(L)$ induced by $L \backslash D$ remains connected.*

(iii) *If L is split into lists L_1 and L_2, then we must remove $O(d^2)$ edges from $\Lambda_d(L)$ to obtain $\Lambda_d(L_1)$ and $\Lambda_d(L_2)$.*

Proof. Part (1) is trivial, as is (2), since each pair of consecutive undeleted vertices is at distance at most $d+1$, and therefore adjacent. Part (3) is also trivial: the number edges connecting any prefix and suffix of L is at most $(d+1)(d+2)/2$. □

Aside from the forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$, the edge set of H_2 includes a set of edges $C(T)$ (for each tree T in the forests) that represents connectivity between major vertices in ancestors of T via paths through descendants of T, i.e., via vertices in $\Delta(T)$.

Definition 3.17. (The Graph H_2) The graph H_2 is on the same vertex set as H_1. The edge set of H_2 includes the forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ and $\bigcup_T C(T)$, where the union is over all trees T in the forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$, and $C(T)$ is constructed as follows:

- Let the strict ancestors of T be $T_1, T_2, \ldots, T_q$.
- For $1 \leq i \leq q$, let $A(T, T_i)$ be a list of the major vertices in T_i that are incident to some vertex in $\Delta(T)$, ordered according to an Euler tour of T_i. (This is done in exactly as in Section 3.1.) Let $A(T)$ be the concatenation of $A(T, T_1), \ldots, A(T, T_q)$.
- Define $C(T)$ to be the edge set $\Lambda_d(A(T))$.

See Figure 3.4 for an illustration of how $C(T)$ is constructed. Lemma 3.18 exhibits the two salient properties of H_2: that it encodes useful connectivity information and that it is economical to effectively destroy $C(T)$ when it is no longer valid, often in time sublinear in $|C(T)|$.

Lemma 3.18. (Disconnecting $C(T)$) *Consider a $C(T) \subseteq E(H_2)$, where T is a tree in $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$.*

(i) Suppose d vertices fail, none of which are in $\Delta(T)$, and let u and v be major vertices in ancestors of T that are adjacent to at least one vertex in $\Delta(T)$. Then u and v remain connected in the original graph and remain connected in H_2.

(ii) Suppose the proper ancestors of T are $T_1, \ldots, T_q$ and a total of f edges are removed from these trees, breaking them into subtrees $T'_1, \ldots, T'_{q+f}$. Then at most $O(d^2(q+f))$ edges must be removed from $C(T)$ such that no remaining edge in $C(T)$ connects distinct trees T'_i and T'_j.

Proof. For Part (1), the vertices u and v are connected in the original graph because they are each adjacent to vertices in $\Delta(T)$ and, absent any failures, all vertices in $\Delta(T)$ are connected, by definition. By Definition 3.17, u and v appear in $C(T)$ and, by Lemma 3.16, $C(T)$ remains connected after the removal of any d vertices. Turning to Part (2), recall from Definition 3.17 that $A(T)$ was the concatenation of $A(T, T_1), \ldots, A(T, T_q)$ and each $A(T, T_i)$ was ordered according to an Euler tour of T_i.

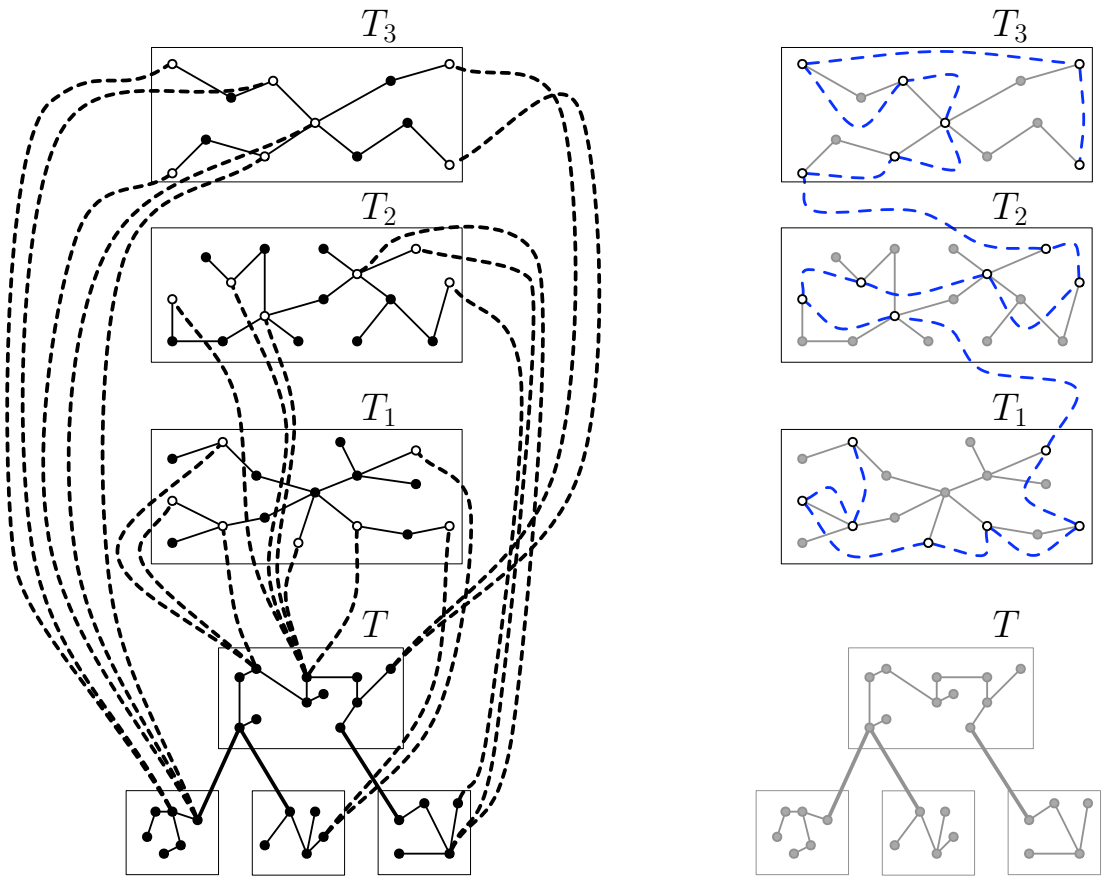


Figure 3.4: **Left:** T is a tree in some forest among $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ having three strict descendants and three ancestors T_1, T_2, T_3. Dashed curves indicate edges connecting vertices from $\Delta(T)$ (all vertices in descendants of T) to major vertices in strict ancestors of T, which are drawn as hollow. **Right:** The set $C(T)$ consists of, first, linking up all hollow vertices in a list that is consistent with Euler tours of T_1, T_2, T_3 (indicated by dashed curves), and second, adding edges between all hollow vertices at distance at most $d + 1$ in the list.

Removing f edges from $T_1, \ldots, T_q$ separates their Euler tours (and, hence, the lists $\{A(T, T_i)\}_i$) into at most $2f + q$ intervals. (This is exactly the same reasoning used in Section 3.1.) By Lemma 3.16 we need to remove at most $(2f + q - 1) \cdot O(d^2)$ edges from $C(T)$ to guarantee that all remaining edges are internal to one such interval, and therefore internal to one of the trees $T'_1, \ldots, T'_{q+f}$. Note that $C(T)$ is now "logically" deleted since remaining edges internal to some T'_i do not add any connectivity. □

Finally, we generate ET-structures for graphs H_1 and H_2, as defined in Section 3.1. Specifically, let $\mathcal{F}$ be the set of all trees in $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$. We associate with the path $U_0, \ldots, U_p$ the two ET-structures $\mathbf{ET}(H_1, \mathcal{F})$ and $\mathbf{ET}(H_2, \mathcal{F})$. Lemma 3.19 bounds the space for the overall data structure.

Lemma 3.19. (Space Bounds) *Given a graph G with m edges, n vertices, and parameters d and $s = (2d)^{c+1} + 1$, where $c \geq 1$, the space for a d-failure connectivity oracle is $O(d^{1-2/c} m n^{1/c - 1/(c \log(2d))} \log^2 n)$.*

Proof. Recall that $k = \log_{(s-1)/2d} n < \log n$ is the height of the hierarchy. Each of H_1 and H_2 has at most $(p+1)n \leq kn$ vertices. Clearly H_1 has less than $kn + m$ edges and we claim that H_2 has less than $kn + (d+1)km$ edges. Each edge (u, v) in the original graph causes v to make an appearance in the list $A(T, T(v))$, whenever $u \in \Delta(T)$, and there are at most k such lists; moreover, v's appearance in $A(T, T(v))$ (and hence $A(T)$) contributes at most $d + 1$ edges to $C(T) = \Lambda_d(A(T))$. By Theorem 3.2, each edge in H_1 or H_2 contributes $O(\log n)$ space in the ET-tree structure in which it appears, for a total of $O((dkm + kn) \log n) = O(dm \log^2 n)$ space for *one* hierarchy node. By Lemma 3.7 there are $d^{-2/c} m n^{1/c - 1/(c \log(2d))}$ hierarchy tree nodes, which gives the claimed bound. □

3.4 Recovery From Failures

In this section we describe how, given up to d failed vertices, the data structure can be updated in time $O((dsk)^2 \log\log n)$ such that connectivity queries can be answered in $O(d)$ time. Section 3.4.1 gives the algorithm to delete failed vertices and Section 3.4.2 gives the query algorithm.

3.4.1 Deleting Failed Vertices

Step 1. Given the set D of at most d failed vertices, we begin by identifying a path $V = U_0, \ldots, U_p$ in the hierarchy in which D have low degree in the $p+1$ levels of forests $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$. By Lemma 3.8 this takes $O(d \log n)$ time.

In subsequent steps we delete all failed vertices in each of their appearances in the forests, i.e., up to $p+1 \leq k$ copies for each failed vertex. Edges remaining in H_1 (between vertices not in D) represent original graph edges and are obviously valid. However, an edge in H_2, say one in $C(T)$, represents connectivity via a path whose intermediate vertices are in the descendant set $\Delta(T)$. If $\Delta(T)$ contains failed vertices then that path may no longer exist, so *all edges* in $C(T)$ become suspect, and are presumed invalid. Although $C(T)$ may contain many edges, Lemma 3.18(2) will imply that $C(T)$ can be *logically* destroyed in time polynomial in d and s. Before describing the next steps in detail we need to distinguish affected from unaffected trees.

Definition 3.20. (Affected Trees) If a tree T in $F_{U_1}(U_0), \ldots, F_{U_{p+1}}(U_p)$ intersects the set of failed vertices D, T and all ancestors of T are *affected*. Equivalently, T is affected if $\Delta(T)$ contains a failed vertex. If T is affected, the connected subtrees of T induced by $V(T) \backslash D$ (i.e., the subtrees remaining after vertices in D fail) are called *affected subtrees*.

Lemma 3.21. (The Number of Affected Trees) *The number of affected trees is at most kd. The number of affected subtrees is at most $kd(s+1)$.*

Proof. If u is a major vertex in T, u can only appear in ancestors of T. Thus, when u fails it can cause at most k trees to become affected. Since, by choice of U_p, all failed vertices have low degree in the trees in which they appear, at most kds tree edges are deleted, yielding $kd(s+1)$ affected subtrees. □

Step 2. We identify the affected trees in $O(kd)$ time and mark as deleted the tree edges incident to failed vertices in $O(kds)$ time. Deleting $O(kds)$ tree edges effectively splits the Euler tours of the affected trees into $O(kds)$ intervals, where each affected subtree is the union of some subset of the intervals.

Step 3. Recall from the discussion above that if T is an affected tree then $\Delta(T)$ contains failed vertices and the connectivity provided by $C(T)$ is presumed invalid. By Lemma 3.18 we can logically delete $C(T)$ by removing $O(d^2)$ edges for *each* edge removed from an ancestor tree of T i.e., $O(d^2 \cdot kds)$ edges need to be removed to destroy $C(T)$. (All remaining edges from $C(T)$ are internal to some affected subtree and can therefore be ignored; they do not provide additional connectivity.) There are at most dk affected trees T, so at most $O(k^2d^4s)$ edges need to be removed from H_2. Let H_2' be H_2 with these edges removed.

Step 4. We now attempt to reconnect all affected subtrees using valid edges, i.e., those not deleted in Step 3. Let R be a graph whose vertices $V(R)$ represent the $O(kds)$ affected subtrees such that $(t_1, t_2) \in E(R)$ if t_1 and t_2 are connected by an edge from either H_1 or H_2'. Using the structures $\mathbf{ET}(H_1, \mathcal{F})$ and $\mathbf{ET}(H_2, \mathcal{F})$ (see Section 3.1, Theorem 3.2), we populate the edge set in time $O(|V(R)|^2 \log\log n + k^2d^4s)$, which is $O((dsk)^2 \log\log n)$ since $s > d^2$. In $O(|E(R)|) = O((dsk)^2)$ time we determine the connected components of R and store with each affected subtree a representative vertex of its component.

This concludes the deletion algorithm. The running time is dominated by Step 4.

3.4.2 Answering a Connectivity Query

The deletion algorithm has already identified the path $U_0, \ldots, U_p$. To answer a connectivity query between u and v we first check to see if there is a path between them that avoids affected trees, then consider paths that intersect one or more affected trees.

Step 1. We find $T(u)$ and $T(v)$ in $O(1)$ time; recall that these are trees in which u and v are major vertices. If $T(u)$ is unaffected, let T_1 be the most ancestral unaffected ancestor of $T(u)$, and let T_2 be defined in the same way for $T(v)$. If $T_1 = T_2$ then $\Delta(T_1)$ contains u and v but no failed vertices; if this is the case we declare u and v *connected* and stop. We can find T_1 and T_2 in $O(\log k) = O(\log \log n)$ time using a binary search over the ancestors of $T(u)$ and $T(v)$, or in $O(\log d)$ time by the complicated least common ancestor data structure by Bender and Farach-Colton [3], in which the least common ancestor can be found in constant time.

Step 2. We now try to find vertices u' and v' in affected subtrees that are connected to u and v respectively. If $T(u)$ is affected then $u' = u$ clearly suffices, so we only need to consider the case when $T(u)$ is unaffected and T_1 exists. Recall from Definition 3.17 that $A(T_1)$ is the list of major vertices in proper ancestors of T_1 that are adjacent to some vertex in $\Delta(T_1)$. We scan $A(T_1)$ looking for *any* non-failed vertex u' adjacent to $\Delta(T_1)$. Since $\Delta(T_1)$ is unaffected, u is connected to u', and since T_1's parent is affected u' must be in an affected subtree. Since there are at most d failed vertices we must inspect at most $d + 1$ elements of $A(T_1)$. This takes $O(d)$ time to find u' and v', if they exist. If one or both of u' and v' does not exist we declare u and v *disconnected* and stop.

Step 3. Given u' and v', in $O(\min\{\log \log n, \log d\})$ time we find the affected subtrees t_1 and t_2 containing u' and v', respectively. Note that t_1 and t_2 are vertices in

R, from Step 4 of the deletion algorithm. We declare u and v to be connected if and only if t_1 and t_2 are in the same connected component of R. This takes $O(1)$ time.

We now turn to the correctness of the query algorithm. If the algorithm replies *connected* in Step 1 or *disconnected* in Step 2 it is clearly correct. (This follows directly from the definitions of $\Delta(T_i)$ and $A(T_i)$, for $i \in \{1, 2\}$.) If u' and v' are discovered then u and v are clearly connected to u' and v', again, by definition of $\Delta(T_i)$ and $A(T_i)$. Thus, we may assume without loss of generality that the query vertices $u = u'$ and $v = v'$ lie in affected subtrees. The correctness of the procedure therefore hinges on whether the graph R correctly represents connectivity between affected subtrees.

Lemma 3.22. (Query Algorithm Correctness) *Let u and v be vertices in affected subtrees t_u and t_v. Then there is a path from u to v avoiding failed vertices if and only if t_u and t_v are connected in R.*

Proof. Edges in R represent either original graph edges (not incident to failed vertices) or paths whose intermediate vertices lie in some $\Delta(T)$, for an unaffected T. Thus, if there is a path in R from t_u to t_v then there is also a path from u to v avoiding failed vertices. For the reverse direction, let P be a path from u to v in the original graph avoiding failed vertices. If all intermediate vertices in P are from affected subtrees then P clearly corresponds to a path in R, since all inter-affected-tree edges in P are included in H_1 and eligible to appear in R. For the last case, let $P = (u, \ldots, x, x', \ldots, y', y, \ldots, v)$, where x' is the *first* vertex not in an affected tree and y is the first vertex following x' in an affected tree. That is, the subpath $(x', \ldots, y')$ lies entirely in $\Delta(T)$ for some unaffected tree T, which implies that x and y appear in $A(T)$. By Lemma 3.18, x and y remain connected in $C(T)$ even if d vertices are removed, implying that x and y remain connected in H_2'. Since all edges from H_2' are eligible to appear in R, t_x and t_y must be connected in R. Thus, u lies in t_u, which is connected to t_x in R, which is connected to t_y in R. The claim then follows by induction on the (shorter) path from y to v. □

3.5 Conclusion

We have given the first space/time-efficient data structure for one of most natural fundamental graph problems: given that a set of vertices has *failed*, is there still a path from point A to point B avoiding all failures? Our connectivity oracle recovers from d vertex failures in time polynomial in d and answers connectivity queries in time linear in d. However, the exponential of d in the update time is large. How to improve this update time and the space to almost linear without making the structure more complex is a major challenge.

In addition to our vertex-failure oracle we presented a new edge-failure oracle that is incomparable to a previous structure of Pătraşcu and Thorup [49] in many ways.[6] We note that it excels when the number of failures is small; for $d = O(1)$ the oracle recovers from failures in $O(\log \log n)$ time and answers connectivity queries in $O(1)$ time. It would be very interesting if lower bounds on predecessor search [48] could be strengthened to give non-trivial lower bounds on vertex- or edge-failure oracles. These questions are still quite difficult even when d is assumed to be a (possibly large) constant.

[6]The recovery time and query time in ours is $O(d^2 \log \log n)$ and $O(\min\{\log \log n, \log d\})$, versus $O(d \log^2 n \log \log n)$ and $O(\log \log n)$ for the version of [49] constructible in exponential time.

CHAPTER IV

All-Pair Bottleneck Paths and Bottleneck Shortest Paths

In this chapter we consider the all-pair bottleneck paths (APBP) problem and all-pair bottleneck shortest path (APBSP) problem. In APBP, for all pairs of vertices s and t, we want to find the path with maximum flow that can be routed from s to t, that is, to maximize the smallest weights of edges in the path. In [69, 65] they show that finding APBP in edge capacitated graphs is equivalent to computing the $(\max, \min)$-product of two real valued matrices, which is defined by $(A \otimes B)[i, j] = \max_k \min\{A[i, k], B[k, j]\}$. (See Definition 4.2.) In this Chapter we give a $(\max, \min)$-matrix product algorithm running in time $O(n^{(3+\omega)/2}) \leq O(n^{2.688})$, where $\omega = 2.376$ is the exponent of binary matrix multiplication. Our algorithm improves on a recent $O(n^{2+\omega/3}) \leq O(n^{2.792})$-time algorithm of Vassilevska, Williams, and Yuster [65].

In APBSP, which asks for the maximum flow that can be routed along a *shortest* path, we give an algorithm for edge-capacitated graphs running in $O(n^{(3+\omega)/2})$ time and a slightly faster $O(n^{2.657})$-time algorithm for vertex-capacitated graphs. The second algorithm significantly improves on an $O(n^{2.859})$-time APBSP algorithm of Shapira, Yuster, and Zwick. [57] [1]

[1] These results appear in Duan and Pettie's paper "Fast Algorithms for (Max, Min)-Matrix Multiplication and Bottleneck Shortest Paths" [23] in SODA 2009.

In Section 4.2 we present our new algorithms for sparse dominance products and $(\max, \min)$-products, which leads directly to a faster APBP algorithm. In Section 4.3 we define new products called dominance-distance and distance-max-min, both of which operate on pairs of matrices. In Sections 4.3.3 and 4.3.4 we show how to compute APBSP in edge- and vertex-capacitated graphs using the distance-max-min product.

4.1 Definitions

In this chapter, we assume w.l.o.g. that the capacities for edges or vertices are real numbers with the additional minimum and maximum elements $-\infty$ and ∞.

4.1.1 Row-Balancing and Column-Balancing

Most algorithms in this chapter will use the concept of row-balancing (and column-balancing) for sparse matrices, in which we partition the dense rows into parts and reposition each part in a distinct row.

Definition 4.1. Let A be an $n \times p$ matrix with m finite elements. Depending on context, the other elements will either all be ∞ or all be $-\infty$. We assume the former below. The *row-balancing* of A, or $\mathbf{rb}(A)$, is a pair (A', A'') of $n \times p$ matrices, each with at most $k = \lceil m/n \rceil$ elements in each row. The row-balancing is obtained by the following procedure: First, sort all the finite elements in the ith row of A in increasing order, and divide this list into several parts $T_i^1, T_i^2, ...T_i^{a_i}$ such that all parts except the last one contain k elements and the last part ($T_i^{a_i}$) contains at most k elements. Let A' be the submatrix of A containing the last parts:

$$A'[i,j] = \begin{cases} A[i,j] & \text{if } A[i,j] \in T^{a_i} \\ \infty & \text{otherwise} \end{cases}$$

Since the remaining parts have exactly k elements, there can be at most $m/k \leq n$ of them. We assign each part to a distinct row in A'', i.e., we choose an arbitrary mapping $\rho : [n] \times [p/k] \to [n]$ such that $\rho(i,q) = i'$ if T_i^q is assigned row i'; it is undefined if T_i^q doesn't exist. Let A'' be defined as:

$$A''[i',j] = \begin{cases} A[i,j] & \text{if } \rho^{-1}(i') = (i,q) \text{ and } (i,j) \in T_i^q \\ \infty & \text{otherwise} \end{cases}$$

Thus, every finite $A[i,j]$ in A has a corresponding element in either A' or A'', which is also in the jth column. The *column-balancing* of A, or $\mathbf{cb}(A)$, is similarly defined as (A'^T, A''^T), where $(A', A'') = \mathbf{rb}(A^T)$.

4.1.2 Matrix Products

We use $\cdot$ to denote the standard $(+,\cdot)$-product on matrices and let $\circledcirc, \circledvee$, and $\star$ be the *dominance*, *max-min*, and *distance* products.

Definition 4.2. (Various Products) Let A and B be real-valued matrices. The products $\cdot, \circledcirc, \circledvee$, and $\star$ are defined as

$$\begin{aligned} (A \cdot B)[i,j] &= \sum_k (A[i,k] \cdot B[k,j]) \\ (A \circledcirc B)[i,j] &= |\{k \mid A[i,k] \leq B[k,j]\}| \\ (A \circledvee B)[i,j] &= \max_k \min\{A[i,k], B[k,j]\} \\ (A \star B)[i,j] &= \min_k \{A[i,k] + B[k,j]\} \end{aligned}$$

In Section 4.3.2 we introduce hybrids of these called the dominance-distance and distance-max-min products.

4.2 Dominance and APBP

Matoušek [45] showed that the dominance product of two $n \times n$ matrices can be computed in $O(n^{(3+\omega)/2}) = O(n^{2.688})$ time. Recently Yuster [68] has slightly improved this to $O(n^{2.684})$ by the rectangular matrix multiplication. However, in our algorithms we need the dominance product only for relatively sparse matrices. Theorem 4.3 shows that $A \circledcirc B$ can be computed in $O(n^\omega)$ time when the number of finite elements is $O(n^{(\omega+1)/2})$. The algorithm behind this theorem is used directly in our APBP and APBSP algorithms. Using Theorem 4.3 as a subroutine we give a faster dominance product algorithm for somewhat denser matrices; however, these improvements have no implications for APBP or related problems. Theorem 4.4 was originally claimed by Vassilevska et al. [65]. Their algorithm, which does not appear in [65], is a bit more involved.

Theorem 4.3. (Sparse Dominance Product) *Let A and B be two $n \times n$ matrices where the number of non-(∞) values in A is m_1 and the number of non-$(-\infty)$ values in B is m_2. Then $A \circledcirc B$ can be computed in time $O(m_1 m_2/n + n^\omega)$.*

Proof. Let $(A', A'') = \mathbf{cb}(A)$ be the column-balancing of A. We build two Boolean matrices $\hat{A}$ and $\hat{B}$ and compute $\hat{A} \cdot \hat{B}$ in $O(n^\omega)$ time.

$$\hat{A}[i,k] = 1 \qquad \text{if } A''[i,k] \neq \infty$$
$$\hat{B}[k,j] = 1 \qquad \text{if } B[k',j] \geq \max T_{k'}^{q'},\ (k',q') = \rho^{-1}(k)$$

One may verify that $\hat{A}[i,k] \cdot \hat{B}[k,j] = 1$ if and only if $B[k',j]$ is greater or equal to *all* the elements in the kth column of A'', which is the q'th part in k'th column of A, where $q' < a_{k'}$ is not the last part of column k'. What $(\hat{A} \cdot \hat{B})[i,j]$ does not count are dominances $A[i,k] \leq B[k,j]$, where either $A[i,k] \in T_k^q$ but $B[k,j]$ dominates some but not all elements in T_k^q, or $A[i,k] \in T_k^{a_k}$ (the last part of column k) and $B[k,j]$

does dominate all of $T_k^{a_k}$. We check these possibilities in $O(m_1 m_2/n)$ time. Each of the m_2 elements in B is compared against at most $\lceil m_1/n \rceil$ elements from A. $\square$

Using the procedure from Theorem 4.3 as a subroutine, we can compute $A \circledcirc B$ faster for denser matrices. The resulting algorithm is somewhat simpler than that of Vassilevska et al. [65].

Theorem 4.4. (Dense Dominance Product) *Let A and B be two $n \times n$ matrices where m_1 is the number of non-(∞) elements in A and m_2 the number of non-$(-\infty)$ elements in B, where $m_1 m_2 \geq n^{1+\omega}$. Then $A \circledcirc B$ can be computed in time $O(\sqrt{m_1 m_2} n^{(\omega-1)/2})$.*

Proof. Let L be the sorted list of all the finite elements in A. We divide L into t parts $L_1, L_2, ..., L_t$, for a t to be determined, so each part has at most $\lceil m_1/t \rceil$ elements. Then we build Boolean matrices $\hat{A}_p, \hat{B}_p, A_p$, and B_p, for $1 \leq p \leq t$ as follows:

$$\hat{A}_p[i,k] = 1 \qquad \text{if } A[i,k] \in L_p$$
$$\hat{B}_p[k,j] = 1 \qquad \text{if } B[k,j] \geq \max L_p$$

$$A_p[i,k] = \begin{cases} A[i,k] & \text{if } A[i,k] \in L_p \\ \infty & \text{otherwise} \end{cases}$$
$$B_p[k,j] = \begin{cases} B[k,j] & \text{if } \min L_p \leq B[k,j] < \max L_p \\ -\infty & \text{otherwise} \end{cases}$$

Notice that every finite element of B is in at most one B_p. One may verify that

$$A \circledcirc B = \sum_{p=1}^{t} (\hat{A}_p \cdot \hat{B}_p + A_p \circledcirc B_p)$$

From Theorem 4.3, the computation of $A_p \circledcirc B_p$ takes time $O((m_1/t)|B_p|/n + n^\omega)$,

where $|B_p|$ is the number of finite elements in B_p. Thus, the total time to compute $A \oslash B$ is $O(m_1 m_2/tn + tn^\omega)$. The theorem follows by setting $t = \sqrt{m_1 m_2}/n^{(1+\omega)/2}$. $\square$

4.2.1 Max-Min Product

In this section we give an efficient algorithm for solving the max-min product of two matrices that uses the sparse dominance product as a key subroutine. One corollary is that all-pairs bottleneck capacities can be found in the same time bound [1]. By incurring an additional $\log n$ factor, we can find all-pairs bottleneck *paths* using existing techniques [69, 65]; see Appendix 4.2.2 for a review.

Theorem 4.5. (Max-Min Product) *Given two real $n \times n$ matrices A and B, $A \otimes\!\!\!\!\bigcirc\, B$ can be computed in $O(n^{(3+\omega)/2}) \leq O(n^{2.688})$ time.*

Proof. It suffices to compute matrices C and C':

$$\begin{aligned} C[i,j] &= \max_k \{A[i,k] \mid A[i,k] \leq B[k,j]\} \\ C'[i,j] &= \max_k \{B[k,j] \mid A[i,k] \geq B[k,j]\} \end{aligned}$$

since $(A \otimes\!\!\!\!\bigcirc\, B)[i,j] = \max\{C[i,j], C'[i,j]\}$. Below we compute C; the procedure for C' is obviously symmetric.

Let L be the sorted list (in increasing order) of all the elements in A and B. We evenly divide L into t parts $L_1, L_2, ..., L_t$, so each part has at most $\lceil 2n^2/t \rceil$ elements. Let A_r and B_r be the submatrices of A and B containing L_r:

$$A_r[i,j] = \begin{cases} A[i,j] & \text{if } A[i,j] \in L_r \\ \infty & \text{otherwise} \end{cases}$$

$$B_r[i,j] = \begin{cases} B[i,j] & \text{if } B[i,j] \in L_r \\ -\infty & \text{otherwise} \end{cases}$$

Let $(A'_r, A''_r) = \mathbf{rb}(A_r)$ be the row-balancing of A_r. After we compute $A_r \circledcirc B$, $A'_r \circledcirc B$, and $A''_r \circledcirc B$, for all r, we may determine $C[i,j]$ as follows:

(i) Find the largest r such that $(A_r \circledcirc B)[i,j] > 0$. Thus, $C[i,j]$ must be in A_r.

(ii) Check whether $(A'_r \circledcirc B)[i,j] > 0$. If it is, since A'_r contains the largest part of each row in A_r, $C[i,j]$ must be in the ith row of A'_r. It follows that $C[i,j] = \max_k\{A'_r[i,k] \mid A'_r[i,k] \leq B[k,j]\}$.

(iii) If $(A'_r \circledcirc B)[i,j] = 0$, find the largest q such that $(A''_r \circledcirc B)[\rho(i,q),j] > 0$. It follows that $C[i,j] \in T_i^q$. We determine $C[i,j]$ be checking each element of T_i^q one by one.

Steps 1–3 take $O(n/t)$ time per element, for a total of $O(n^3/t)$ time. To compute $A_r \circledcirc B$ we begin by building two Boolean matrices $\hat{A}_r$ and $\hat{B}_r$ for all r such that:

$$\hat{A}_r[i,k] = 1 \qquad \text{if } A[i,k] \in L_r$$
$$\hat{B}_r[k,j] = 1 \qquad \text{if } B[k,j] \in L_{r+1} \cup \cdots \cup L_t$$

It is straightforward to see that $A_r \circledcirc B = A_r \circledcirc B_r + \hat{A}_r \cdot \hat{B}_r$: the inter-part comparisons are covered in $\hat{A}_r \cdot \hat{B}_r$ and the intra-part comparisons in $A_r \circledcirc B_r$. The products $A'_r \circledcirc B$ and $A''_r \circledcirc B$ can be computed in a similar fashion.

By Theorem 4.3 the time to compute $A_r \circledcirc B, A'_r \circledcirc B$, and $A''_r \circledcirc B$, for all r, is $t \cdot O(n^3/t^2 + n^\omega)$. In total the running time is $O(n^3/t + tn^\omega)$. The theorem follows by setting $t = n^{(3-\omega)/2}$. □

Theorem 4.5 leads immediately to an algorithm computing all-pairs bottleneck *capacities* in $O(n^{(3+\omega)/2})$ time. We review Section 4.2.2 an existing algorithm [69, 65] for finding explicit bottleneck *paths*.

Corollary 4.6. *APBP can be computed in $O(n^{(3+\omega)/2})$ time.*

4.2.2 Explicit Maximum Bottleneck Paths

The algorithm from Theorem 4.6 calculates the capacities of all bottleneck paths but does not return the paths as such. In this section we review some well known algorithms for actually generating the paths.

Let A_0 be the original capacity matrix of the graph (with ∞ along the diagonal) and let $A_q = A_{q-1} \otimes\!\!\!\!\bigcirc\, A_{q-1}$. Thus, $A_{\lceil \log n \rceil}[i,j]$ is the capacity of the bottleneck path between vertices i and j. Let W_q be the witness matrix for the qth iteration, i.e.:

$$W_q[i,j] = k \text{ s.t. } A_q[i,j] = \min\{A_{q-1}[i,k], A_{q-1}[k,j]\}$$

It is very simple to have our algorithms return the witness matrix. Let $I[i,j] = \min\{q \mid A_q[i,j] = A_{\lceil \log n \rceil}[i,j]\}$ be the iteration that establishes the bottleneck capacity between i and j. If the bottleneck path from i to j is composed of l edges we can return the path in $O(l)$ time as follows. If $I[i,j] = 0$ return the edge (i,j); otherwise, concatenate the paths from i to $W_{I[i,j]}[i,j]$ and from $W_{I[i,j]}[i,j]$ to j. The procedure above gives each edge in *amortized* constant time. Zwick [69] and Vassilevska et al. [65] gave simple procedures for finding the successor matrix S, given W, I, which allows us to generate the bottleneck path in $O(1)$ worst case time per edge. Let $S[i,j] = k$ if (i,k) is the first edge on the path from i to j.

It is straightforward to show that the **witness-to-successor** algorithm is correct and runs in $O(n^2)$ time; see [69, 65].

witness-to-successor(W, I)

$S \leftarrow 0$

For q from 0 to $\log n$

$I_q \leftarrow \{(i,j) \mid I[i,j] = q\}$

For every $(i,j) \in I_0$

$S[i,j] \leftarrow j$

For q from 1 to $\log n$

  For each $(i, j) \in I_q$

    $k \leftarrow W_q[i, j]$

    While $S[i, j] = 0$

      $S[i, j] \leftarrow S[i, k]$

      $i \leftarrow S[i, k]$

Return S

The procedure above can easily be adapted to work with our APBSP algorithms.

4.3 Bottleneck Shortest Paths

In this section, we consider the All-Pairs Bottleneck Shortest Paths problem (APBSP) in both edge- and vertex-capacitated graphs. Let $D(u, v)$ be the unweighted distance from u to v and let $sc(u, v)$ be the maximum capacity path from u to v with length $D(u, v)$.

When the graph is *edge*-capacitated we give an APBSP algorithm running in $\tilde{O}(n^{(3+\omega)/2})$ time, matching the running time of our APBP algorithm. This is the first published subcubic APBSP algorithm for edge-capacitated graphs. It improves on an unpublished algorithm of Vassilevska [63], which runs in $O(n^{(15+\omega)/6}) = O(n^{2.896})$ time. For vertex-capacitated graphs our algorithm runs slightly faster, in $O(n^{2.657})$ time; this improves on a recent algorithm of Shapira et al. [57] running in $O(n^{(8+\mu)/3}) = O(n^{2.859})$ time.

In Section 4.3.1 we review some facts about rectangular matrix multiplication. In Section 4.3.2 we present fast algorithms for certain *hybrid* products based on dominance, distance, and max-min products. In Sections 4.3.3 and 4.3.4 we present our APBSP algorithms for edge- and vertex-capacitated graphs.

4.3.1 Rectangular Matrix Multiplication

In our algorithms we often use fast rectangular matrix multiplication algorithms [11, 40]. Let $\omega(r, s, t)$ to be the constant such that multiplying $n^r \times n^s$ and $n^s \times n^t$ matrices takes $O(n^{\omega(r,s,t)})$ time. We use the standard definitions of the constants α, β, and μ.

Definition 4.7. Let α be the maximum value satisfying $\omega(1, \alpha, 1) = 2$ and let $\beta = \frac{\omega-2}{1-\alpha}$. Define μ to be the constant satisfying $\omega(1, \mu, 1) = 1 + 2\mu$. If $\omega = 2$ then $\alpha = 1, \beta = 0$, and $\mu = 1/2$.

Then following bounds on α, β, and μ can be found in [11, 40]:

Lemma 4.8. *$\alpha > 0.294$, $\beta > 0.533$, and $\mu < 0.575$. For $s \geq \alpha$, $\omega(1, s, 1) \leq 2 + \beta(s - \alpha)$.*

4.3.2 Hybrid Products

Our all-pairs bottleneck shortest path algorithms use products that are hybrids of dominance, distance, and max-min products.

Definition 4.9. (Dominance-Distance) Let $(A, \tilde{A})$ and $(B, \tilde{B})$ be pairs of real matrices. Their *dominance-distance* product is written $C = (A, \tilde{A}) \overset{\leqslant}{\star} (B, \tilde{B})$, where

$$C[i, j] = \min_{\substack{k: \\ \tilde{A}[i,k] \leq \tilde{B}[k,j]}} (A[i, k] + B[k, j])$$

In a similar fastion we define the distance-max-min product as a hybrid of distance and max-min.

Definition 4.10. (Distance-Max-Min) Let $(A, \tilde{A})$ and $(B, \tilde{B})$ be pairs of real

matrices. Their *distance-max-min* product is defined as:

$$
\begin{aligned}
(C, \tilde{C}) &= (A, \tilde{A}) \overset{\star}{\circledcirc} (B, \tilde{B}) \\
&\text{where} \\
C &= A \star B \\
\tilde{C}[i,j] &= \max_{\substack{k: \\ A[i,k]+B[k,j]=C[i,j]}} \min\{\tilde{A}[i,k], \tilde{B}[k,j]\}
\end{aligned}
$$

Our algorithms make use of Zwick's algorithm [69] for distance products in integer-weighted matrices.

Theorem 4.11. (Distance Product) *[69] Let A and B be $n \times n^s$ and $n^s \times n$ matrices, respectively, whose elements are in $\{1, \ldots, M\}$. Then $A \star B$ can be computed in $O(\min\{n^{2+s}, Mn^{\omega(1,s,1)}\})$ time.*

Theorem 4.12. (Dominance-Distance Product) *Let $A, \tilde{A}, B, \tilde{B}$ be matrices such that:*

$$
\begin{aligned}
A &\in \{1, \ldots, M, \infty\}^{n \times n^s} & \tilde{A} &\in (\mathbb{R} \cup \{\infty\})^{n \times n^s} \\
B &\in \{1, \ldots, M, \infty\}^{n^s \times n} & \tilde{B} &\in (\mathbb{R} \cup \{-\infty\})^{n^s \times n}
\end{aligned}
$$

where M is an integer and $s \leq 1$. If the number of finite elements in $\tilde{A}$ and $\tilde{B}$ are m_1 and m_2, resp., then $(A, \tilde{A}) \overset{\circledcirc}{\star} (B, \tilde{B})$ can be computed in $O(m_1 m_2 / n^s + Mn^{\omega(1,s,1)})$ time.

Proof. Let $(\tilde{A}', \tilde{A}'') = \mathbf{cb}(\tilde{A})$ be the column-balancing of $\tilde{A}$. We build two matrices

$\hat{A}$ and $\hat{B}$, defined below. Here $(k', q') = \rho^{-1}(k)$.

$$\hat{A}[i,k] = \begin{cases} A[i,k'] & \text{if } \tilde{A}''[i,k] \neq \infty \\ \infty & \text{otherwise} \end{cases}$$

$$\hat{B}[k,j] = \begin{cases} B[k',j] & \text{if } \tilde{B}[k',j] \geq \max T_{k'}^{q'} \\ \infty & \text{otherwise} \end{cases}$$

In other words, $(\hat{A} \star \hat{B})[i,j]$ is the minimum $A[i,k'] + B[k',j]$ such that $\tilde{B}[k',j]$ dominates *all* of $T_{k'}^{q'}$, the part containing $A[i,k']$. Furthermore, $q' < a_{k'}$, i.e., $T_{k'}^{q'}$ is not the last part that appears in $\tilde{A}'$. What we must consider now are sums $A[i,k]+B[k,j]$ which could be smaller than $(\hat{A}\star\hat{B})[i,j]$. If $\tilde{A}[i,k] \in T_k^q$ then we must examine $\tilde{B}[k,j]$ if it dominates some, but not all, elements of T_k^q, or if $q = a_k$ and $\tilde{B}[k,j]$ dominates all of $T_k^{a_k}$. Each of the m_2 elements of $\tilde{B}$ participates in $\lceil m_1/n^s \rceil$ such sums, requiring $O(m_1 m_2/n^s)$ time. The product $\hat{A} \star \hat{B}$ is computed in $O(Mn^{\omega(1,s,1)})$ time. □

Just as the max-min product may be applied directly to compute APBP, the distance-max-min product will be useful in computing APBSP on both edge- and vertex-capacitated graphs.

Theorem 4.13. (Distance-Max-Min Product) *Let $A, \tilde{A}, B, \tilde{B}$ be matrices such that:*

$$A \in \{1, \ldots, M, \infty\}^{n \times n^s} \qquad \tilde{A} \in \mathbb{R}^{n \times n^s}$$

$$B \in \{1, \ldots, M, \infty\}^{n^s \times n} \qquad \tilde{B} \in \mathbb{R}^{n^s \times n}$$

where M is an integer and $s \leq 1$. Then $(A, \tilde{A}) \overset{\star}{\otimes} (B, \tilde{B})$ can be computed in $O(\min\{n^{2+s}, M^{1/2} \cdot n^{1+s/2+\omega(1,s,1)/2}\})$ time.

Proof. Recall that $(C, \tilde{C}) = (A, \tilde{A}) \overset{\star}{\otimes} (B, \tilde{B})$, where $C = A \star B$ and $\tilde{C}[i,j] = \max_k \min\{\tilde{A}[i,k], \tilde{B}[k,j]\}$ such that $A[i,k] + B[k,j] = C[i,j]$. (Note that we could

always compute C and $\tilde{C}$ by the trivial algorithm in $O(n^{2+s})$ time.) We begin by computing C in $O(Mn^{\omega(1,s,1)})$ time with Zwick's algorithm [69], then compute matrices $\tilde{C}_1, \tilde{C}_2$:

$$\tilde{C}_1[i,j] = \max_k\{\tilde{A}[i,k] \mid \tilde{A}[i,k] \leq \tilde{B}[k,j] \text{ and } A[i,k] + B[k,j] = C[i,j]\}$$
$$\tilde{C}_2[i,j] = \max_k\{\tilde{B}[k,j] \mid \tilde{A}[i,k] \geq \tilde{B}[k,j] \text{ and } A[i,k] + B[k,j] = C[i,j]\}$$

One can verify that $\tilde{C}[i,j] = \max\{\tilde{C}_1[i,j], \tilde{C}_2[i,j]\}$. Below we describe how to compute $\tilde{C}_1$; computing $\tilde{C}_2$ is symmetric.

Let L be the sorted list of all the elements in $\tilde{A}$ and $\tilde{B}$. We divide L into t parts, $L_1, L_2, ..., L_t$, so each part has $2n^{1+s}/t$ elements. Define the matrices A_r and B_r, for $1 \leq r \leq t$, as:

$$\tilde{A}_r[i,j] = \begin{cases} \tilde{A}[i,j] & \text{if } \tilde{A}[i,j] \in L_r \\ \infty & \text{otherwise} \end{cases}$$
$$\tilde{B}_r[i,j] = \begin{cases} \tilde{B}[i,j] & \text{if } \tilde{B}[i,j] \in L_r \\ -\infty & \text{otherwise} \end{cases}$$

Let $(\tilde{A}'_r, \tilde{A}''_r) = \mathbf{rb}(\tilde{A}_r)$ be the row-balancing of $\tilde{A}_r$. We compute the dominance-distance products G_r, G'_r, and G''_r, for $1 \leq r \leq t$, defined as:

$$G_r = (A, \tilde{A}_r) \overset{\Theta}{\star} (B, \tilde{B})$$
$$G'_r = (A, \tilde{A}'_r) \overset{\Theta}{\star} (B, \tilde{B})$$
$$G''_r = (A, \tilde{A}''_r) \overset{\Theta}{\star} (B, \tilde{B})$$

For every pair i, j we determine $\tilde{C}_1[i,j]$ as follows:

(i) Find the largest r such that $G_r[i,j] = C[i,j]$, then $\tilde{C}_1[i,j]$ must be in $\tilde{A}_r$

(ii) Check whether $G'_r[i,j] = C[i,j]$. If it is, $\tilde{C}_1[i,j]$ must be in the ith row of $\tilde{A}'_r$. Check all the finite elements in that row one by one.

(iii) If $G'_r[i,j] \neq C[i,j]$, find the largest q such that $G''_r[\rho(i,q),j] = C[i,j]$. Thus, $\tilde{C}_1[i,j]$ must be in T_i^q, the qth part of the ith row of $\tilde{A}$. Check the elements in T_i^q one by one.

Steps 1–3 take $O(n^s/t)$ time per pair, that is, $O(n^{2+s}/t)$ time in total. What remains is to show that we can compute G_r, G'_r, G''_r in the stated bounds. To find G_r, we begin by constructing two matrices $\hat{A}_r$ and $\hat{B}_r$ such that:

$$\hat{A}_r[i,k] = \begin{cases} A[i,k] & \text{if } \tilde{A}[i,k] \in L_r \\ \infty & \text{otherwise} \end{cases}$$

$$\hat{B}_r[k,j] = \begin{cases} B[k,j] & \text{if } \tilde{B}[k,j] \in L_{r+1} \cup \cdots \cup L_t \\ \infty & \text{otherwise} \end{cases}$$

We compute $\hat{G}_r = \hat{A}_r \star \hat{B}_r$ using Zwick's algorithm [69] and $\tilde{G}_r = (A, \tilde{A}_r) \overset{\circledcirc}{\star} (B, \tilde{B}_r)$ using the algorithm from Theorem 4.12. One may verify that:

$$G_r[i,j] = \min\{\hat{G}_r[i,j],\ \tilde{G}_r[i,j]\}$$

If $G_r[i,j] = A[i,k] + B[k,j]$, the $\hat{G}_r$ matrix covers the case where $A[i,k]$ and $B[k,j]$ come from different parts and $\tilde{G}_r$ covers the case where they are both in part L_r. The matrices G'_r and G''_r are computed in a similar fashion.

In total, the time required to find G_r, G'_r, and G''_r, for $1 \leq r \leq t$, is $t \cdot O(Mn^{\omega(1,s,1)} + n^{2+s}/t^2)$, where the first term comes from [69] and the second from Theorem 4.12.

(Recall that $\tilde{A}_r$ and $\tilde{B}_r$ have at most $2n^{1+s}/t$ finite elements.) We choose t to be $n^{1+s/2-\omega(1,s,1)/2}M^{-1/2}$, which makes the overall running time $O(M^{1/2}\cdot n^{1+s/2+\omega(1,s,1)/2})$.

□

4.3.3 APBSP with Edge Capacities

For the APBSP problem, we use the "bridging sets" technique; see Zwick [69] and Shapira et al. [57]. A standard probabilistic argument shows that a small set of randomly selected vertices will cover a set of relatively long paths.

Lemma 4.14. *[69] Let S be a set of paths between distinct pairs of vertices, each of length at least t, in a graph with n vertices. A set of $O(t^{-1}n\log n)$ vertices selected uniformly at random contains, with probability $1-n^{-\Omega(1)}$, at least one vertex from each path in S. Such a set is called a t-bridging set. A t-bridging set can be found deterministically in $O(tn^2)$-time.*

Theorem 4.15. *Given a real edge-capacitated graph on n vertices, APBSP can be computed in $O(n^{(3+\omega)/2}) = O(n^{2.688})$ time.*

Proof. We begin by computing unweighted distances in $O(n^{2+\mu}) = O(n^{2.575})$ time [69]. Let D and C be the distance and edge capacity matrices, respectively. For vertices at distance 1 or 2 it follows that:

$$sc(u,v) = \begin{cases} C[u,v] & \text{if } D[u,v]=1 \\ (C \circledcirc C)[u,v] & \text{if } D[u,v]=2 \end{cases}$$

In general, once $sc(u,v)$ is computed for u,v with $D[u,v]\le t$, it can be computed for all u,v with $D[u,v]\le 3t/2$ as follows. Let B be a bridging set for the set of bottleneck shortest paths with length $t/2$. We compute such a set if $t\le\sqrt{n}$ and, if not, use the last bridging set when t was at most $\sqrt{n}$. Thus, $|B| = \tilde{O}(\max\{n/t,\sqrt{n}\})$. If $D[u,v]$ is between t and $3t/2$ there must be some vertex $b\in B$ that lies on the middle third of

the bottleneck shortest path from u to v and, therefore, satisfies $D[u,b], D[b,v] \leq t$. In other words, $sc(u,v)$ can be derived from $sc(u,b)$ and $sc(b,v)$, both of which have already been computed. We have:

$$sc(u,v) = \max_{\substack{b \in B \\ D[u,b]+D[b,v]=D[u,v]}} \min\{sc(u,b), sc(b,v)\}$$

This is clearly an instance of the distance-max-min product of $n \times |B|$ and $|B| \times n$ matrices. If $B = \tilde{O}(\sqrt{n})$ we use the trivial $O(n^{2.5})$-time algorithm. Otherwise, let $n^s = |B| = \tilde{O}(n/t)$. By Theorem 4.13 this product can be computed in time:

$$\begin{aligned}
&O(t^{\frac{1}{2}} \cdot n^{1+\frac{s+\omega(1,s,1)}{2}}) && \\
&= O(n^{\frac{3+\omega(1,s,1)}{2}}) && t = n^{1-s} \\
&= O(n^{\frac{5+\beta(s-\alpha)}{2}} && \omega(1,s,1) \leq 2 + \beta(s-\alpha) \\
&= O(n^{(3+\omega)/2}) && s \leq 1, \beta(1-\alpha) = \omega - 2
\end{aligned}$$

By Lemma 4.14 B can be computed in $O(tn^2) = O(n^{5/2}) = O(n^{(3+\omega)/2})$ time. The procedure above is obviously repeated just $\log_{3/2} n$ times, for a total running time of $O(n^{(3+\omega)/2})$. □

4.3.4 APBSP with Vertex Capacities

In this section, we consider the APBSP problem for vertex-capacitated graphs. There are two variants of the problem: closed-APBSP, where the endpoints of a path are taken into account, and open-APBSP, where they are not. However, Shapira et al. [57] showed that open-APBSP is reducible to closed-APBSP in $O(n^2)$ time. Thus we only consider the closed-APBSP problem in this chapter. The Shapira et al. algorithm runs in time $O(n^{(8+\mu)/3}) \leq O(n^{2.859})$. Here we improve their result by

the techniques introduced earlier.

Lemma 4.16 shows how bottleneck shortest paths can be found quickly for relatively close pairs of vertices. The proof borrows extensively from [57].

Lemma 4.16. *Given a vertex-capacitated graph on n vertices, the bottleneck shortest paths can be computed for all pairs at distance at most n^t, in time $O(n^{(3+\omega+t-3\beta)/(2-\beta)})$.*

Proof. Number the vertices $V = \{v_1, v_2, ..., v_n\}$ in increasing order of capacity. We begin by computing the distance matrix D in $O(n^{2+\mu})$ time [69]. For each $s = 0, ..., n^t$, we compute two $n \times n$ Boolean matrices P_s and Q_s, where $P_s[i,j] = 1$ if and only if there is a path from v_i to v_j, of length at most s, in which v_j has minimum capacity, $Q_s[i,j] = 1$ if and only if there is a path from v_i to v_j, of length at most s, in which v_i has minimum capacity. From [57], the computation will take $O(n^{t+\omega})$ time, as follows. Let E be the adjacency matrix of G and F be the Boolean matrix satisfying $F[i,j] = 1$ iff $i \geq j$. Then $P_0 = Q_0 = I$, $P_s = EP_{s-1} \wedge F$ and $Q_s = Q_{s-1}E \wedge F^T$.

We define two $n \times n$ matrices A and B:

$$A[i,j] = \begin{cases} D[i,j] & \text{if } P_{D[i,j]}[i,j] = 1 \\ \infty & \text{otherwise} \end{cases}$$

$$B[i,j] = \begin{cases} D[i,j] & \text{if } Q_{D[i,j]}[i,j] = 1 \\ \infty & \text{otherwise} \end{cases}$$

Then we just need to compute the bottleneck capacity matrix C in which:

$$C[i,j] = \min\{k \mid A[i,k] + B[k,j] = D[i,j]\}$$

By the definition of A and B, $A[i,k] = D[i,k]$, $B[k,j] = D[k,j]$, and v_k has the minimum capacity in both paths. Thus $sc(v_i, v_j)$ is just the capacity of $v_{C[i,j]}$.

To compute C, as in [14], partition A into $n \times n^r$ sub-matrices A_p and B into $n^r \times n$ sub-matrices B_p where A_p covers columns $(p-1)n^r + 1$ through pn^r and

B_p covers the rows $(p-1)n^r+1$ through pn^r. Then, for every p, we compute the distance product $C_p = A_p \star B_p$, which will take $O(n^{1-r} \cdot n^{\omega(1,r,1)+t})$ time. For $r > \alpha$ we have $\omega(1,r,1) \leq 2+\beta(r-\alpha) = \omega-(1-r)\beta$. Thus, the time for this phase is $O(n^{\omega+t+1+\beta(r-1)-r})$

For every i,j, find the smallest p such that $C_p[i,j] = D[i,j]$, i.e., $C[i,j]$ will be in the range $[(p-1)n^r+1, pn^r]$. We check all possibilities one by one. This will take $O(n^{2+r})$ time. To balance the two bounds we choose $r = (\omega+t-1-\beta)/(2-\beta)$, making the total running time $O(n^{\frac{\omega+3+t-3\beta}{2-\beta}})$. □

Theorem 4.17. *Given a vertex-capacitated graph with n vertices, APBSP can be computed in $O(n^{\frac{3+\omega}{2}-\frac{\beta^2(3-\omega)}{4+2\beta(2-\beta)}}) = O(n^{2.657})$ time.*

Proof. This algorithm has two phases. In the first phase we use Lemma 4.16 to compute the bottleneck shortest paths for vertices at distance is at most n^t, for some properly selected t. In the second phase, we convert the vertex-capacitated graph to an edge-capacitated graph by giving each edge the minimum capacity of its endpoints. The algorithm from Theorem 4.15 will compute bottleneck shortest paths for the remaining vertex pairs in $O(n^{(3+\omega-\beta t)/2})$ time. To balance the two phases we choose $t = \beta(3-\omega)/(2+\beta(2-\beta))$, making the total running time: $O(n^{\frac{3+\omega}{2}-\frac{\beta^2(3-\omega)}{4+2\beta(2-\beta)}}) = O(n^{2.657})$. □

CHAPTER V

Dual-Failure Distance Oracle

5.1 Introduction

This chapter considers the problem of answering distance queries between any two vertices in a graph with the presence of several failed vertices. More precisely, given source and target vertices x, y and a set $F \subset V$, the problem is to report $\delta_{G-F}(x, y)$, where $\delta_{G'}$ is the distance function w.r.t. the subgraph G'.[1] Demetrescu et al. [16] consider this type of structure for single-failure distance queries (either a vertex or an edge), which can be answered in constant time by an oracle occupying $O(n^2 \log n)$ space. We extend this to two vertex failures with only more $\log n$ factors on space and query time. The main result can be summarized by the following theorem: [2]

Theorem 5.1. *Given a weighted directed graph $G = (V, E, \ell)$, where $\ell : E \to \mathbb{R}$ assigns arbitrary real lengths, a data structure with size $O(n^2 \log^3 n)$ can be constructed in polynomial time such that given vertices $x, y \in V$ and two failed vertices or edges $u, v \in V \cup E$, $\delta_{G-\{u,v\}}(x, y)$ can be reported in $O(\log n)$ time. Furthermore, a path with this length can be returned in $O(\log n)$ time per edge.*

[1] The notation $G - z$ refers to the graph G after removing z, where z is a vertex, edge, or set of vertices or edges.

[2] This result appears in Duan and Pettie's paper "Dual-Failure Distance and Connectivity Oracles" [22] in SODA 2009.

As a special case, Theorem 5.1 allows one to answer connectivity queries in $O(\log n)$ time. We only prove Theorem 5.1 for two vertex failures. There is a simple reduction from an f-edge failure distance query to $O(1)$ f-vertex failure queries, for any fixed f. We can use Theorem 5.1 to answer distance queries involving an arbitrary number of failures. For $f \geq 2$ we can build an $\tilde{O}(n^f)$-space data structure answering f-failure distance queries in $\tilde{O}(1)$ time. This compares favorably with the trivial $O(n^{f+2})$ space bound and the $\tilde{O}(n^{f+1})$ bound implied by [16, 5].

Our data structure of dual-failure distance queries is much more complicated than those of [16, 5], which can be seen from the number of cases caused by the second failed vertex/edge. If p is a shortest path from x to y and u a failed vertex, the shortest path avoiding u consists of a prefix of the original path p followed by a "detour" avoiding p (and u), followed by a suffix of p. In the presence of 2 failures (assumed to be u and v) it is no longer possible to create such a clean partition. The shortest path avoiding two failed vertices on p may depart from and return to p many times, because p can be directed and the detour can travel back on p. When we first find the detour only avoiding u, then v may be on this detour, but if we further find the detour avoiding v, the new detour may still pass through u. So we will need much more complex structures and query algorithm to deal with this. With 3 (or more) failures the possible cases of the optimal detours becomes even more complicated. The conclusion we draw from our results is that handling dual-failure distance queries is possible but extending our structure to handle 3 or more failures is practically infeasible.

Organization. In Section 5.2 we summarize the notation used throughout this chapter. In Section 5.3, we review the one-failure distance structure similar to [16], which gives us the basic idea of constructing oracles for more failures. In Sections 5.4–5.6 we describe how the query algorithm works in part of Case I– III.

5.2 Notations:

In this section we summarize the notation and conventions used throughout the paper.

- The query asks for the shortest path from x to y avoiding vertices u and v. We assume that at least one failed node, u, lies on the shortest path from x to y.

- We use $p_H(x, y)$ to denote the shortest path from x to y in the subgraph H and use xy as shorthand for $p_G(x, y)$, where G is the whole graph. The length and number of edges in a path p are denoted as $\|p\|$ and $|p|$, respectively. The concatenation of two paths p and p' is $p \cdot p'$. We use min to select the path with minimum length, i.e., $\min\{p_1, \ldots, p_k\}$ refers to the minimum length path among $\{p_1, \ldots, p_k\}$.

- We define the function $\rho_s(p_H(s,t))$ to be the vertex $c \in p_H(s,t)$ such that $|p_H(s,c)| = 2^{\lfloor \log |p_H(s,t)| \rfloor}$, i.e., $\rho_s(p_H(s,t))$ is the farthest vertex from s in the path $p_H(s,t)$, to whom the *unweighted* distance from s in H is a power of 2. Symmetrically, the function $\rho_t(p_H(s,t))$ is the vertex $c \in p_H(s,t)$ such that $|p_H(c,t)| = 2^{\lfloor \log |p_H(s,t)| \rfloor}$. It is easy to see $\rho_t(p_H(s,t))$ is *before* $\rho_s(p_H(s,t))$ in $p_H(s,t)$.

- Let $p_H(x,y) \diamond A$ be short for $p_{H-A}(x,y)$. For example, our query is to determine $\|xy \diamond \{u,v\}\|$: the length of the shortest x-y path avoiding u and v. Here A can be a range of vertices if the range is clear from context. For example, if we have established that s and t appear in $xy \diamond u$ then $(xy \diamond u) \diamond [s,t]$ refers to the shortest path from x to y avoiding u *and* the subpath from s to t within $xy \diamond u$.

- We let $s \oplus i$ and $s \ominus i$ be the ith vertex *after* s and *before* s on some path known from context. Typically the path we are considering is from x to y. For brevity we use $\oplus i$ and $\ominus i$ as short for $x \oplus i$ and $y \ominus i$, respectively. For example,

$(xy \diamond u) \diamond (\ominus i)$ is the shortest path from x to y avoiding u and avoiding the ith vertex *before* y on the path $xy \diamond u$.

The following vertices are all *with respect to* some path $p_H(x, y)$ known from context, e.g., $p_H(x, y)$ may be $xy \diamond u$ (which is $P_{G-u}(x, y)$).

- Δ, ∇: The vertex at which $p_H(x, y)$ and xy first diverge is called the *divergence point*, denoted by Δ, and, symmetrically, the vertex where they converge is called the *convergence point*, denoted by ∇.

- w, w': Let $w \in p_H(x, y)$ be the first vertex such that $wy = p_H(w, y)$; i.e., wy is a subpath of $p_H(x, y)$. So for *every* vertex *before* w, its shortest path to y in G goes through some vertex in $G \backslash H$. Symmetrically, $w' \in p_H(x, y)$ is the last vertex such that $xw' = p_H(x, w')$.

Throughout the paper we use the term *detour* to mean the shortest path avoiding some set of vertices.

5.3 Review of the One-Failure Distance Oracle

As in [16], throughout the paper we assume that all shortest paths in any subgraph of G are unique. Thus, we can determine if u is on *the* shortest path xy by checking whether $\|xu\| + \|uy\| = \|xy\|$.

Before delving into the description of our two-failure distance oracle, we first give a simplified version of the one-failure oracle [5, 16] that uses a log-factor more space: $O(n^2 \log^2 n)$.

5.3.1 Structure

- B_0: For every pair of vertices x and y, $B_0(x, y)$ stores $\|xy\|$, $|xy|$, i.e., the length and the number of vertices of xy. We also preprocess the shortest path trees [3]

so that, given x_1, x_2, y, the first common vertex of x_1y and x_2y can be answered in constant time.

- B_1: For every pair of vertices x and y, $B_1(x, y)$ stores the length and number of vertices of the paths:

$$\begin{aligned} xy \diamond \{\oplus 2^i\} \quad &, \quad \forall i < \lfloor \log |xy| \rfloor \\ xy \diamond \{\ominus 2^i\} \quad &, \quad \forall i < \lfloor \log |xy| \rfloor \\ xy \diamond [\oplus 2^i, \ominus 2^j] \quad &, \quad \forall i, j < \lfloor \log |xy| \rfloor \end{aligned}$$

5.3.2 Query Algorithm

Let the only failed vertex on the path xy be u. If $|xu|$ or $|uy|$ is an integer power of 2, then $xy \diamond u$ will be stored in $B_1(x, y)$, so we can get the distance avoiding u immediately. Otherwise we will find $u_l = \rho_u(xu)$ and $u_r = \rho_u(uy)$ on xy, so $|u_l u|$ and $|uu_r|$ are powers of 2. There are 3 possible types of detours:

(i) The detour that reaches some point in $[u_l, u)$.
(ii) The detour that reaches some point in $(u, u_r]$.
(iii) The detour that avoids the range $[u_l, u_r]$ in xy.

For the first and second types, the path will go through u_l or u_r. Since u is not on xu_l and $|u_l u|$ is a power of 2, $\|xu_l\|$ is stored in $B_0(x, u_l)$ and $\|u_l y \diamond u\|$ is stored in $B_1(u_l, y)$. Then $xu_l \cdot u_l y \diamond u$ is just the shortest path of the first type. In this situation, we say that the path $xu_l \cdot u_l y \diamond u$ **covers** this type. Symmetrically, $xu_r \diamond u \cdot u_r y$ can cover the second type, where $\|xu_r \diamond u\|$ is in $B_1(x, u_r)$ and $\|u_r y\|$ is in $B_0(u_r, y)$. When we deal with the third type, we let $x' = \rho_x(xu)$ and $y' = \rho_y(uy)$, so $|xx'|$ and $|y'y|$ are integer powers of two and $xy \diamond [x', y']$ is stored in $B_1(x, y)$. Since x' is *after* u_l and y' is *before* u_r on xy, the detour avoiding $[u_l, u_r]$ must also avoid $[x', y']$, so $xy \diamond [x', y']$

covers the third type. (See Figure 5.1.) Thus, the single failure distance can be found in constant time.

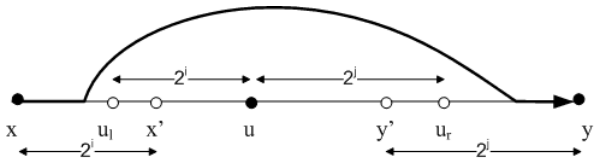


Figure 5.1: One-failure case, where the thick line denotes a detour of the third type.

In the following parts of this paper, we will consider the dual failure data structure in three cases. In Section 5.4, we will dicuss Case I, in which only u is on xy and $|xu|$ or $|uy|$ is a power of 2. In Section 5.5, the general case of only $u \in xy$ will be discussed, we call it Case II. In Section 5.6, we will talk about Case III, where both u and v are on xy.

5.4 Case I

The first case we consider is when only one of the failed vertices u is on the original shortest path from x to y and $|xu|$ (or, symmetrically, $|uy|$) is a power of 2.

In Section 5.4.1 we present the data structures used in Case I. In Section 5.4.2 we present the query algorithm and dispense with several relatively easy subcases. Sections 5.4.2.1 and 5.4.2.2 cover the more complicated subcases of Case I.

5.4.1 Structures

First we will introduce the data structures used in Case I, which are

5.4.1.1 Common Structures

B_0, B_1: As described in the one-failure case.

B_2: For every detour $p_H(x, y) \in B_1(x, y)$ and every $\hat{x} \in \{x, \Delta, w\}$, $\hat{y} \in \{y, \nabla, w'\}$, when $\hat{x}$ is *before* $\hat{y}$, $B_2(x, y)$ stores the length and number of vertices of the paths:

(The definitions of Δ, ∇, w, w' have been stated in Section 5.2.)

$$
\begin{aligned}
p_H(x,y) \diamond \{\hat{x} \oplus 2^i\}, &\quad \forall i < \lfloor \log |p_H(xy)| \rfloor \\
p_H(x,y) \diamond \{\hat{y} \ominus 2^i\}, &\quad \forall i < \lfloor \log |p_H(xy)| \rfloor \\
p_H(x,y) \diamond [\hat{x} \oplus 2^i, \hat{x} \oplus 2^{i+1}], &\quad \forall i < \lfloor \log |p_H(xy)| - 1 \rfloor \\
p_H(x,y) \diamond [\hat{y} \ominus 2^{j+1}, \hat{y} \ominus 2^j], &\quad \forall j < \lfloor \log |p_H(xy)| - 1 \rfloor
\end{aligned}
$$

We also store the length of $p_H(x, \hat{x} \oplus 2^i)$ and $p_H(\hat{y} \ominus 2^j, y)$ for every "exponential of two" points on $p_H(x,y)$. One can see that the structures B_0, B_1, B_2 occupy $O(n^2 \log^3 n)$ space.

In this paper, u_l usually means the vertex from which the number of vertices on the shortest path to u is a power of 2, and u_r usually means the vertex to which the number of vertices on the shortest path from u is a power of 2. Similar as v_l and v_r.

5.4.1.2 The Tree Structure

In this section we introduce a specialized but useful data structure whose purpose will only become clear once it is seen in action, in Section 5.4.2. For every pair of vertices (u,y), define the sets $S(u,y)$ and $\hat{S}(u,y)$ as:

$$
\begin{aligned}
S(u,y) &= \{x \mid u \in xy \text{ and } |xu| \text{ is a power of 2}\} \\
\hat{S}(u,y) &= S(u,y) \cup \{z \mid \exists x_1, x_2 \in S(u,y) \\
&\qquad s.t.\ z \text{ is the first common vertex of } x_1 y \diamond u \text{ and } x_2 y \diamond u\}
\end{aligned}
$$

In the tree formed by the shortest paths from the vertex set of $S(u,y)$ to y in the subgraph $G - \{u\}$, $\hat{S}(u,y)$ is the set of all leaves and branch vertices in the tree, so $|\hat{S}(u,y)| \leq 2|S(u,y)|$. Given a vertex y, every other vertex x can only be in at

most $\log n$ different $S(u,y)$ since u must be on xy and $|xu|$ is a power of 2. Thus $\sum_u |S(u,y)| = n \log n$, and $\sum_{u,y} |S(u,y)| = n^2 \log n$.

For every pair of vertices (u,y) we store the following tree structure $T(u,y)$. For a given $x \in \hat{S}(u,y)$, let $z(i)$ be the 2^ith vertex of $\hat{S}(u,y)$ on the path $xy \diamond u$, i.e., $|(xz(i) \diamond u) \cap \hat{S}(u,y)| = 2^i$. For each $x \in \hat{S}(u,y)$ and i,j, we store $\|(xy \diamond u) \diamond [z(i), \ominus 2^j]\|$ in $T(u,y)$, where $\ominus 2^j$ is w.r.t. $xy \diamond u$. We also preprocess $T(u,y)$ to answer level ancestor and least common ancestor queries [3, 4] in the tree induced by $\hat{S}(u,y)$ in constant time; this allows us to identify $z(i)$ and other vertices in $O(1)$ time. Obviously the size of $T(u,y)$ is $O(|S(u,y)| \log^2 n)$, and the total space for the T structure is $O(n^2 \log^3 n)$.

Lemma 5.2. *Given $x_1, x_2 \in S(u,y)$ and an integer j, let z be the first common vertex of $x_1 y \diamond u$ and $x_2 y \diamond u$. Using the tree structure $T(u,y)$ we can find $\|(x_1 y \diamond u) \diamond [z, \ominus 2^j]\|$ in constant time.*

Proof. The vertex z can be identified in $O(1)$ time with a least common ancestor query. Let $i = \left\lfloor \log |(x_1 z \diamond u) \cap \hat{S}(u,y)| \right\rfloor$ be the log of the number of $\hat{S}(u,y)$-vertices on the path $x_1 z \diamond u$. Using two level ancestor queries we can identify z_l and x_1' in $T(u,y)$ where $|(z_l z \diamond u) \cap \hat{S}(u,y)| = 2^i$ and $|(x_1 x_1' \diamond u) \cap \hat{S}(u,y)| = 2^i$. The shortest detour $(x_1 y \diamond u) \diamond [z, \ominus 2^j]$ must be one of the following.

(i) The detour that avoids the range $[z_l, \ominus 2^j]$ in $xy \diamond u$.

(ii) The detour that reaches some point in $[z_l, z)$.

The lengths of both of the paths $\|(x_1 y \diamond u) \diamond [x_1', \ominus 2^j]\|$ and $\|x_1 z_l \cdot (z_l y \diamond u) \diamond [z, \ominus 2^j]\|$ can be retrieved in $O(1)$ time from $T(u,y)$ and $B_0(x_1, z_l)$. These two paths cover both of the possibilities for $(x_1 y \diamond u) \diamond [z, \ominus 2^j]$. See Figure 5.2. □

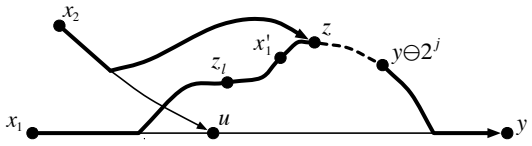


Figure 5.2: An illustration of the query $(x_1 y \diamond u) \diamond [z, \ominus 2^j]$, where we are given j, x_1, x_2, and y, but not z.

5.4.2 The detour from x to y avoiding u

We begin with a simple observation:

Lemma 5.3. *Suppose for some distinct vertices u, v, x and y, v is on the detour $xy \diamond u$. Then at least one of u and v is on xy.*

Proof. Suppose u is not on xy, then $xy \diamond u = xy$, so $v \in xy$. □

Since $|xu|$ is a power of 2, $xy \diamond u$ is stored in $B_1(x, y)$. We determine whether $v \in xy \diamond u$ by checking whether $\|xv \diamond u\| + \|vy \diamond u\| = \|xy \diamond u\|$ in constant time using the one-failure oracle. If $v \notin xy \diamond u$, the optimal detour is just $xy \diamond u$. If $v \in xy \diamond u$ and $|xv \diamond u|$ or $|vy \diamond u|$ is a power of 2, then we can return $(xy \diamond u) \diamond v$, which is stored in $B_2(x, y)$. Otherwise, we proceed to find $v_l = \rho_v(xv \diamond u)$ and $v_r = \rho_v(vy \diamond u)$ in $O(\log n)$ time as follows:

Since $|xu|$ is a power of 2, if $u \in xv$ then $xv \diamond u \in B_1(x, v)$, otherwise $xv \diamond u = xv \in B_0(x, v)$. Thus the vertex v_l whose unweighted distance to v in $xv \diamond u$ is a power of 2 can be found in $B_2(x, v)$ or $B_1(x, v)$ in constant time.

However, since $|uy|$ is not necessarily a power of 2, v_r is not symmetrical to v_l. To locate v_r, we analyze how the path $vy \diamond u$ was constructed in the one-failure query algorithm. The only non-trivial case is when $vy \diamond u$ was composed of two parts (the first or second types, from Section 5.3.2), i.e., it was of the form $vu_l' \cdot u_l' y \diamond u$ or $vu_r' \diamond u \cdot u_r' y$, where $|uu_r'|$ and $|u_l' u|$ are powers of 2. We find some vertex v' that, depending on the form of $vy \diamond u$, is a maximal power of 2 from v, u_l', or u_r' (in unweighted distance) but

before v_r. We then continue to search for v_r on $v'y \diamond u$. Since $|v'y \diamond u| < |vy \diamond u|/2$ this procedure terminates after $O(\log n)$ steps. If v_r lies in vu'_l or $u'_r y$ then v' may be retrieved from B_1; if it lies in $vu'_r \diamond u$ or $u'_l y \diamond u$ then v' is stored in B_2.

The optimal detour avoiding u and v will belong to one of the following types:

(i) The detour that avoids u and the range $[v_l, v_r]$ in $xy \diamond u$.

The shortest detour of this kind must be no shorter than $(xy \diamond u) \diamond [\oplus 2^j, \oplus 2^{j+1}] \in B_2(x, y)$ or $(xy \diamond u) \diamond [\ominus 2^{j'+1}, \ominus 2^{j'}] \in B_2(x, y)$ where $j = \lfloor \log |xv \diamond u| \rfloor$ and $j' = \lfloor \log |vy \diamond u| \rfloor$. To see this, without loss of generality, assume $j < j'$. Then v_l is *before* $x \oplus 2^j$ in $xy \diamond u$ and $|vv_r| = 2^{j'} > 2^j$, so v_r is *after* $x \oplus 2^{j+1}$ in $xy \diamond u$. Therefore, any detour avoiding $[v_l, v_r]$ belongs to the set of detours avoiding $[\oplus 2^j, \oplus 2^{j+1}]$ when $j < j'$. (This is the same argument used in [16].)

(ii) The detour that reaches some points in $(v, v_r]$.

In this case, the detour must go through v_r, so we need to find the path $xv_r \diamond \{u, v\} \cdot v_r y \diamond \{u, v\}$. Since $v \notin v_r y \diamond u$, we only consider the path $xv_r \diamond \{u, v\}$. When $u \in xv_r$, since $|xu|$ and $|vv_r \diamond u|$ are powers of 2, we can immediately return $(xv_r \diamond u) \diamond v \in B_2(x, v_r)$. When $u \notin xv_r$ ($xv_r \diamond u = xv_r$), since $v \in xv_r \diamond u$, by Lemma 5.3 only v is on xv_r and $|vv_r|$ is a power of 2. Thus $xv_r \diamond v \in B_1(x, v_r)$. If $u \notin xv_r \diamond v$ (which can be checked with the one-failure oracle) we are done. If $u \in xv_r \diamond v$, since $|xu \diamond v| = |xu|$ is a power of 2, we just return $(xv_r \diamond v) \diamond u$, which is stored in $B_2(x, v_r)$.

(iii) The detour that reaches some point in $[v_l, v)$, but does not reach $(v, v_r]$.

So now we only have to consider the last type of detour, which must go through v_l but not $(v, v_r]$. So far we have only ascertained that $v \in v_l y \diamond u$ and $|v_l v \diamond u|$ is a power of 2. From Lemma 5.3, at least one of u and v is on $v_l y$. We break the analysis into two main cases depending on whether v is in $v_l y$ (Case I.1) or not (Case II.2).

In both cases, we begin by locating the u_l and u_r relative to u on the path $v_l y \diamond v$. We consider further subcases depending on whether $u_l \in xy \diamond u$:

- **I.1.a:** $v \in v_l y$ and $u_l \in xy \diamond u$
- **I.1.b:** $v \in v_l y$ and $u_l \notin xy \diamond u$
- **I.2.a:** $v \notin v_l y$ and $u_l \notin xy \diamond u$
- **I.2.b:** $v \notin v_l y$ and $u_l \in xy \diamond u$

5.4.2.1 Case I.1: v is on $v_l y$

Since v is not on uy, u cannot be *before* v on $v_l y$. So $u \notin v_l v$ and $|v_l v|$ is a power of 2. We check whether u is in $v_l y \diamond v$; if not, we are done. If $u \in v_l y \diamond v$, define $\overline{w}$ as the point w of the detour $v_l y \diamond v$, i.e., the first vertex in $v_l y \diamond v$ which satisfies $v \notin \overline{w} y$. Since $v \notin uy$, u must be equal to or *after* $\overline{w}$ on $v_l y \diamond v$. If $u = \overline{w}$, then $(v_l y \diamond v) \diamond \overline{w}$ is in $B_2(v_l, y)$. Otherwise we can find $u_l = \rho_u(\overline{w} u \diamond v)$ and $u_r = \rho_u(uy \diamond v)$ in $O(\log n)$ time as in Section 5.4.2. So u_l is *after* $\overline{w}$ on $v_l y \diamond v$, and $v \notin u_l y$. The possible types of detours from v_l to y are:

(i) The detour that avoids v and the range $[u_l, u_r]$ in $v_l y \diamond v$.

(ii) The detour that reaches some point in $(u, u_r]$.

(iii) The detour that reaches some point in $[u_l, u)$, but does not reach $(u, u_r]$.

Similar to the discussion in Section 5.4.2, the first case can be covered by the paths $(v_l y \diamond v) \diamond [\overline{w} \oplus 2^k, \overline{w} \oplus 2^{k+1}], (v_l y \diamond v) \diamond [\ominus 2^{k'+1}, \ominus 2^{k'}] \in B_2(v_l, y)$ for some k, k', and the second case can also be handled in the same way. Thus we only have to consider the third case, that is, the path from u_l to y avoiding u and $(v, v_r]$. Since $v \notin u_l y$ and $|u_l u|$ is a power of 2, $u_l y \diamond u$ is in $B_1(u_l, y)$. We check whether u_l is on $xy \diamond u$ in constant time and have the following subcases:

Case I.1.a When $u_l \in xy \diamond u$, we know that $v_l \in xy \diamond u$ and $u \notin v_l v$. Furthermore, u_l is not in the range $[v_l, v)$ on $xy \diamond u$. To see this, assume u_l is on $v_l v \diamond u = v_l v$

(the range $[v_l, v)$ on $xy \diamond u$). Since $v \in v_l y$, u_l is *before* v on $v_l y$ and $v \in u_l y$, which contradicts the fact that $v \notin u_l y$.

If u_l is *after* v on $xy \diamond u$, then $v \notin u_l y \diamond u$, so we just return $u_l y \diamond u$. We do not need to consider the case when u_l is *before* v_l on $xy \diamond u$ since any detour that goes through v_l then u_l contains a cycle.

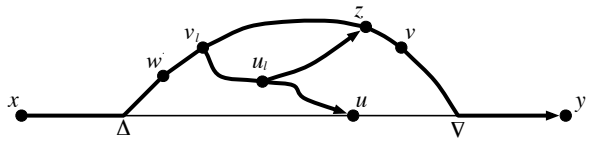


Figure 5.3: The usage of tree structure in Case I.1.b.

Case I.1.b When $u_l \notin xy \diamond u$, since $u \in xy, u \in u_l y$ and both $|xu|$ and $|u_l u|$ are powers of 2, x and u_l are both in $S(u, y)$. From Lemma 5.2, we can find the least common ancestor z of x and u_l in $T(u, y)$ in constant time [56], i.e., z is the first common vertex of the shortest paths $xy \diamond u$ and $u_l y \diamond u$. (See Figure 5.3.) If $v \notin u_l y \diamond u$, just return $u_l y \diamond u$. If $v \in u_l y \diamond u$, v must be *after* z in the path $u_l y \diamond u$ because $v \in xy \diamond u$.

Assume the shortest detour reaches u_l then reaches some point in the common range $[z, v)$ of $xy \diamond u$ and $u_l y \diamond u$. Since $u_l \notin xy \diamond u$, the path from x through $xy \diamond u$ to $[z, v)$ must be shorter than that detour. Thus we do not need to consider the detours that pass through u_l then to some vertices in $[z, v)$ of $u_l y \diamond u$.

Since we are in the third type of Section 5.4.2, in which the range $(v, v_r]$ is avoided, we just have to find $(u_l y \diamond u) \diamond [z, v_r]$, which can be covered by $(u_l y \diamond u) \diamond [z, \ominus 2^{j'}]$ (where $j' = \lfloor \log |vy \diamond u| \rfloor$) since $v_r = v \oplus 2^{j'}$ is *after* $y \ominus 2^{j'}$ on $xy \diamond u$. By Lemma 5.2, this can be achieved in constant time using the $T(u, y)$ structure.

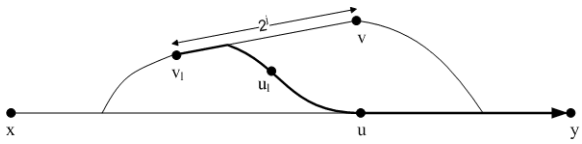


Figure 5.4: Illustration of Case I.2.a

5.4.2.2 Case I.2: v is not on $v_l y$

Since v is on $v_l y \diamond u$, by Lemma 5.3, u must be on $v_l y$. We find the vertices $u_l = \rho_u(v_l u)$ and $u_r = \rho_u(uy)$. There are two further possible cases:

Case I.2.a If u_l is not on $xy \diamond u$, this case is very similar to Case I.1.b. The three possible types of detours are:

(i) The detour that avoids the range $[u_l, u_r]$ in $v_l y$ and the vertex v.

(ii) The detour that reaches some point in $(u, u_r]$.

(iii) The detour that reaches some point in $[u_l, u)$, but does not reach $(u, u_r]$.

The first type is clearly in $B_2(v_l, y)$, since $(v_l y \diamond [u_l, u_r]) \diamond v$ can be covered by $v_l y \diamond [\oplus 2^j, \ominus 2^{j'}] \diamond v$ ($j = \lfloor \log |v_l u| \rfloor$ and $j' = \lfloor \log |uy| \rfloor$), and the number of vertices between v_l and v in that path is a power of 2 (see Figure 5.4). The second type is also similar to Section 5.4.2. But for the third type, the detour must reach u_l, so we have to find the path $u_l y$ avoiding u and v. Since u_l and x are both in $S(u, y)$, by the same argument of Case I.1.b, we only need to find the path $(u_l y \diamond u) \diamond [z, v_r]$, where z is the first common vertex of $xy \diamond u$ and $u_l y \diamond u$. By utilizing the tree structure $T(u, y)$, the path $(u_l y \diamond u) \diamond [z, \ominus 2^{j'}]$ where $j' = \lfloor \log |vy \diamond u| \rfloor$ can be answered in constant time.

Case I.2.b If u_l is on $xy \diamond u$, there are two kinds of detours since our overall goal is to find $(xy \diamond u) \diamond (v, v_r]$:

(i) The detour $(xy \diamond u) \diamond [u_l, v_r]$

(ii) The detour that reaches some point in $[u_l, v)$

For the first kind, both x and u_l are in $S(u, y)$, so they are in the tree structure $T(u, y)$. We can find the detour $(xy \diamond u) \diamond [u_l, \ominus 2^{j'}]$ where $j' = \lfloor \log |vy \diamond u| \rfloor$ in constant time, which will cover the first kind.

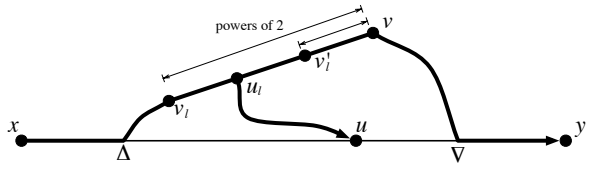


Figure 5.5: The illustration of Case I.2.a(3), where v_l' is the corresponding v_l for the path $u_l y \diamond u$.

For the second kind, the detour will reach u_l through $xu_l \diamond u$. Since only u is on $u_l y$ and $|u_l u|$ is a power of 2, $u_l y \diamond \{u, v\}$ itself is in Case I, and we can deal with it recursively by the procedure of Case I. When we try to find the detour from u_l to y avoiding u and v by the procedure in Section 5.4.2, the position of v_r has not changed, so we do not need another $O(\log n)$ time to locate it. Furthermore the new v_l' found must be such that $|v_l' v \diamond u|$ is a *smaller* power of 2 than $|v_l v \diamond u|$; see Figure 5.5. Thus, the number of recursive invocations of Case I is $O(\log n)$.

5.5 Case II: One failed vertex on xy

In Case II we deal with the situation where only one failed vertex is on xy. Our strategy is to systematically reduce such a query to several Case I queries. The case that $|xu|$ or $|uy|$ is a power of 2 has already been studied in Section 5.4.

For the general case, as in the one-failure algorithm, we find $u_l = \rho_u(xu)$ and $u_r = \rho_u(uy)$ on xy. The 3 possible type of detours are:

(i) The detour that reaches some point in $(u, u_r]$.

(ii) The detour that reaches some point in $[u_l, u)$.

(iii) The detour that avoids the range $[u_l, u_r]$ in xy.

For the first and second types, the path will go through u_r or u_l. Since $|u_l u|$ and $|uu_r|$ are powers of 2, these types are reducible to Case I. When we deal with the third type, we can see $xy \diamond [x', y'] \in B_1(xy)$, where $x' = \rho_x(xu)$ and $y' = \rho_y(uy)$.

The first thing we will face is checking whether v is in $xy \diamond [x', y']$. We have to consider two possibilities. First, if $xy \diamond [x', y']$ is $xy \diamond u$, it is easy to check whether v is in $xy \diamond [x', y']$. If $xy \diamond [x', y']$ is not $xy \diamond u$, the difficulty arises from the fact that the subpath of $xy \diamond [x', y']$ from x to v could be different from $xv \diamond u$, since xv could only go through one part of $[x', y']$ and the detour avoiding this part can also go through the other part of $[x', y']$. However, we only need to consider whether v is in the union of $xy \diamond [x', y']$ and $xy \diamond u$, since it is trivial if v is not in $xy \diamond u$.

This will need some extra data structures and different ideas from the previous case. First we will introduce the data structures only used in Case II.

5.5.1 Data Structures

When a path $xy \diamond [x', y']$ is known from context, where x', y' are two points on xy, we define the following c-vertices. (Here the subscripts and superscripts are mnemonics, where l,r,b,F,L stand for left, right, both, first, last, respectively.)

- c_l : Define c_l to be the first vertex in the range (Δ, ∇) *of the path* $xy \diamond [x', y']$ satisfying:

$$\exists u' \in [x', y'], \text{ such that } x', c_l \in xy \diamond u', \text{ and } y' \notin xy \diamond u'$$

  and symmetrically, let c_r be the last vertex in the range (Δ, ∇) of the path $xy \diamond [x', y']$ satisfying:

$$\exists u' \in [x', y'], \text{ such that } y', c_r \in xy \diamond u' \text{ and } x' \notin xy \diamond u'$$

In this structure we also store the u' with c_l or c_r.

- Let c_{bl} be the first vertex in the range (Δ, ∇) of the path $xy \diamond [x', y']$ satisfying:

$$\exists u' \in [x', y'] \text{ such that } x', y', c_{bl} \in xy \diamond u'$$

- Denote the set Ψ_l to be $(xy \diamond [x', y']) \cap (x'y' \diamond u')$, in which $x'y' \diamond u'$ must be a subpath of $xy \diamond u'$. So c_{bl} is the first vertex on $xy \diamond [x', y']$ that is in Ψ_l. We also define the following vertices in Ψ_l:
  - Let c_{bl}^F be the first vertex of $x'y' \diamond u'$ on Ψ_l.
  - Let c_{bl}^L be the last vertex of $x'y' \diamond u'$ on Ψ_l.
  - Let c_{bl}^r be the last vertex on $xy \diamond [x', y']$ that is in Ψ_l.

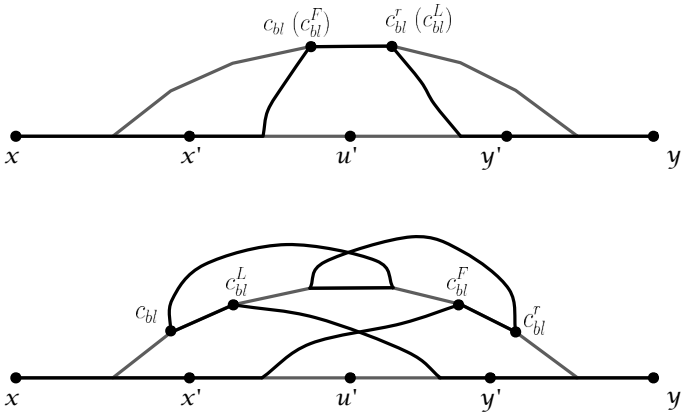


Figure 5.6: The illustration of the position of u' and c_{bl}, etc. There are two possibilities. The grey line is the path $xy \diamond [x', y']$, and the black line are $xy \diamond u'$.

On $xy \diamond [x', y']$, we have c_{bl} is before c_{bl}^L and c_{bl}^F is before c_{bl}^r. Since the range $[c_{bl}^F, c_{bl}^L]$ on $xy \diamond u'$ is disjoint from $x'y'$, the ranges $[c_{bl}, c_{bl}^L]$ and $[c_{bl}^F, c_{bl}^r]$ on the path $xy \diamond [x', y']$ are also on the path $xy \diamond u'$, thus they are in Ψ_l. See Figure 5.6.

- In a symmetric fashion, define c_{br} to be the last vertex on the range (Δ, ∇) in the path $xy \diamond [x', y']$ satisfying:

$$\exists u'' \in [x', y'] \text{ such that } x', y', c_{br} \in xy \diamond u''$$

- Also denote the set Ψ_r to be $xy \diamond [x', y'] \cap x'y' \diamond u''$, in which $x'y' \diamond u''$ must be a subpath of $xy \diamond u''$. So c_{br} is the last vertex on $xy \diamond [x', y']$ that is in Ψ_r. We also define the following vertices in Ψ_r:

  - Let c_{br}^F be the first vertex of $x'y' \diamond u''$ on Ψ_r.
  - Let c_{br}^L be the last vertex of $x'y' \diamond u''$ on Ψ_r.
  - Let c_{br}^l be the first vertex on $xy \diamond [x', y']$ that is in Ψ_r.

  On $xy \diamond [x', y']$, we have c_{br}^l is before c_{br}^L and also c_{br}^F is before c_{br}. Thus the ranges $[c_{br}^l, c_{br}^L]$ and $[c_{br}^F, c_{br}]$ on the path $xy \diamond [x', y']$ are also on the path $xy \diamond u''$.

In order to simplify the description of the data structure from Section 5.4.1.1 we left out some pieces that are only used in Case II. The structure B_2 contains more paths than previously stated (the difference is that $\hat{x}$ and $\hat{y}$ can also be in $\{c_l, c_{bl}, c_{bl}^F, c_{br}^F, c_{br}^l\}$ and $\{c_r, c_{br}, c_{br}^L, c_{bl}^L, c_{bl}^r\}$, resp.) and we also use make use of a new structure $\overline{B}_2$. They are defined as follows:

- B_2: For every detour $p_H(x, y) \in B_1(x, y)$ and every $\hat{x} \in \{x, \Delta, w, c_l, c_{bl}, c_{bl}^f, c_{br}^f, c_{br}^L\}$ and $\hat{y} \in \{y, \nabla, w', c_r, c_{br}, c_{br}^l, c_{bl}^l, c_{bl}^R\}$ ($\hat{x}$ is before $\hat{y}$), $B_2(x, y)$ contains the distance and the number of vertices of the paths:

$$\begin{aligned}
p_H(x, y) \diamond \{\hat{x} \oplus 2^i\} \quad &, \quad \forall i < \lfloor \log |p_H(x, y)| \rfloor \\
p_H(x, y) \diamond \{\hat{y} \ominus 2^i\} \quad &, \quad \forall i < \lfloor \log |p_H(x, y)| \rfloor \\
p_H(x, y) \diamond [\hat{x} \oplus 2^i, \hat{x} \oplus 2^{i+1}] \quad &, \quad \forall i < \lfloor \log |p_H(x, y)| - 1 \rfloor \\
p_H(x, y) \diamond [\hat{y} \ominus 2^{j+1}, \hat{y} \ominus 2^j] \quad &, \quad \forall j < \lfloor \log |p_H(x, y)| - 1 \rfloor
\end{aligned}$$

- For every vertex a on a path $p_H(x,y)$ in $B_1(x,y)$, if ay is not a subpath of $p_H(x,y)$, define $F(a)$ to be the first vertex at which ay and $p_H(x,y)$ diverge. Symmetrically if xa is not a subpath of $p_H(x,y)$, define $F'(a)$ to be the first vertex at which xa and $p_H(x,y)$ converge. See Figure 5.7. We clearly have the following property:

- $\overline{B}_2$: In $\overline{B}_2(x,y)$, for a path of the form $p_H(x,y) \diamond [a,b]$ in $B_2(x,y)$ ($p_H(x,y) \in B_1(x,y)$), we store $p_H(x,y) \diamond [F(a), b]$, $p_H(x,y) \diamond [a, F'(b)]$ and $p_H(x,y) \diamond [F(a), F'(b)]$.

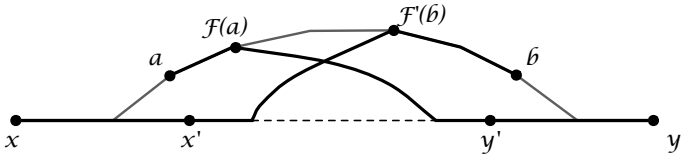


Figure 5.7: The a and b represent vertices on the path $xy \diamond [x', y']$, and the black line denotes the path ay and xb. So the vertices $F(a)$ and $F'(b)$ are determined.

5.5.2 Query Algorithm

Theorem 5.4. *Suppose we have known that $v \in xy \diamond [x', y']$. For any $\hat{x} \in \{x, \Delta, w, c_l, c_{bl}, c_{bl}^f, c_{br}^f, c_{br}^L\}$ and $\hat{y} \in \{y, \nabla, w', c_r, c_{br}, c_{br}^l, c_{bl}^l, c_{bl}^R\}$ on the path $xy \diamond [x', y']$, if $\hat{x}v \diamond u$ and $v\hat{y} \diamond u$ are both subpaths of $xy \diamond [x', y']$, then we can find $(xy \diamond [x', y']) \diamond v$ in $O(\log n)$ time.*

Proof. First we find $v_l = \rho_v(\hat{x}v \diamond u)$ and $v_r = \rho_v(v\hat{y} \diamond u)$ in $O(\log n)$ time by the same procedure in Section 5.4.2. By the condition of the theorem, v_l and v_r are both in $xy \diamond [x', y']$. Then there are three possibilities:

(i) The detour that reaches some point in $[v_l, v)$.

(ii) The detour that reaches some point in $(v, v_r]$.

(iii) The detour that avoids the range $[v_l, v_r]$ in $xy \diamond [x', y']$.

Since $xy \diamond [x', y']$ is in $B_1(xy)$, as before, let $j = \lfloor \log |xv \diamond u| \rfloor$, $j' = \lfloor \log |vy \diamond u| \rfloor$ and find the following vertices on $xy \diamond u = xy \diamond [x', y']$:

$$a_1 = x \oplus 2^j \tag{5.1}$$

$$b_1 = x \oplus 2^{j+1} \tag{5.2}$$

$$a_2 = y \ominus 2^{j'+1} \tag{5.3}$$

$$b_2 = y \ominus 2^{j'} \tag{5.4}$$

We can see $(xy \diamond [x', y']) \diamond [a_1, b_1], (xy \diamond [x', y']) \diamond [a_2, b_2] \in B_2(x, y)$, which can covered the third type. However, due to the need of the analysis of the first type, we further use $F(a_i)$ to replace a_i $(i = 1, 2)$ if v is not on $a_i F(a_i)$, which is a subpath of $xy \diamond [x', y']$. Similarly, we replace b_i with $F'(b_i)$ if v is not on $F'(b_i)b_i$. Then the path $xy \diamond [x', y']) \diamond [a_i(F(a_i)), b_i(F(b_i))]$ will be stored in $\overline{B}_2$. For example, if v is not on $a_1 F(a_1)$ and $F'(b_1)b_1$, then we use the path $(xy \diamond [x', y']) \diamond [F(a_1), F'(b_1)]$ which is in $\overline{B}_2$. Clearly, they can also cover the third type since $F(a_i)$ is equal to or after a_i and $F(b_i)$ is equal to or before b_i. The importance of the use of $F(a_i)$ and $F'(b_i)$ is shown in the subcase 2 of the first type discussed below.

We will only consider the path from v_l to y to cover the first type as the path from x to v_r is symmetric to it.

When $u \notin v_l y$, if $v_l y$ also does not go through v, it is trivial. If it goes through v, $|v_l v| = |v_l v \diamond u|$ is a power of 2 from the definition of v_l, so this case is reducible to Case I.

When $u, v \in v_l y$, since $v \notin uy$, u is after v on $v_l y$. So $|v_l v| = |v_l v \diamond u|$ is a power of 2, and this case is reducible to Case I, see Section 5.6.1.

When only $u \in v_l y$, then we find $u_l'' = \rho_u(v_l u)$. There are 2 types of detours needed to be considered: (The one that reaches $(u, u_r]$ have already been covered $(xu_r \diamond u) \diamond v$, which is in the Case I.)

(i) The detour from v_l to y that avoids $[u_l'', u_r]$ and v.

(ii) The detour from v_l to y that reaches some point in $[u_l'', u)$.

For the second type, if we start at u_l'', since $|u_l''u|$ is a power of 2, it is reducible to Case I. But for the first type, there are two subcases. See Figure 5.8.

Subcase 1 $u_l'' \notin v_l v \diamond u$. Let $v_l' = \rho_{v_l}(v_l u)$, so v_l' is after u_l'' on $v_l y$ and it is not on $v_l v \diamond u$, so $v_l' \notin v_l v \diamond u$. Suppose v is on $v_l y \diamond [v_l', y']$, because $v_l' y'$ is disjoint with the path $v_l v \diamond u$, the subpath of $v_l y \diamond [v_l', y']$ from v_l to v must be the same as $v_l v \diamond u$, which is a subpath of $xy \diamond [x', y']$, so v must be a "power of 2" vertex on the path $v_l y \diamond [v_l', y']$.

Thus, in Subcase 1, first we check whether v is a "power of 2" point of $v_l y \diamond [v_l', y']$ in $B_2(v_l, y)$. If it is, $(v_l y \diamond [v_l', y']) \diamond v$ can be covered by $B_2(v_l, y)$. If it is not, we can conclude that v is not on $v_l y \diamond [v_l', y']$, so just return $v_l y \diamond [v_l', y']$ for the first type above.

Subcase 2 $u_l'' \in v_l v \diamond u$. So u_l'' is also on $xy \diamond [x', y']$. Since u_l'' is in $v_l u$, u is not in $v_l u_l''$, so $v_l u_l'' = v_l u_l'' \diamond u$ is a subpath of $xy \diamond [x', y']$. Thus u_l'' is before or equal to $F(v_l)$ on $xy \diamond [x', y']$, and $F(v_l)$ is before v since $v \notin v_l y$ in this case. We will see the importance of $\overline{B}_2$ here. Consider the a_1 and a_2 defined above, we also consider two possibilities:

- If u_l'' is before or equal to a_i $(i = 1, 2)$, then the detour $(xy \diamond [x', y']) \diamond [a_i, b]$ (Here b can be b_i or $F'(b_i)$) in $\overline{B}_2$ has already cover the first type, since the path reaches a_i will also reach u_l'', and $u_l'' y$ is reducible to Case I.

- If u_l'' is after a_i, then a_i is also on $v_l y$, so $F(a_i) = F(v_l)$, which is after or equal to u_l''. Also $F(a_i)$ is before v, so the detour $(xy \diamond [x', y']) \diamond [F(a_i), b]$ (Here b can be b_i or $F'(b_i)$) in $\overline{B}_2$ has already cover the first type, since the path reaches $F(a_i)$ will also reach u_l'', and $u_l'' y$ is reducible to Case I.

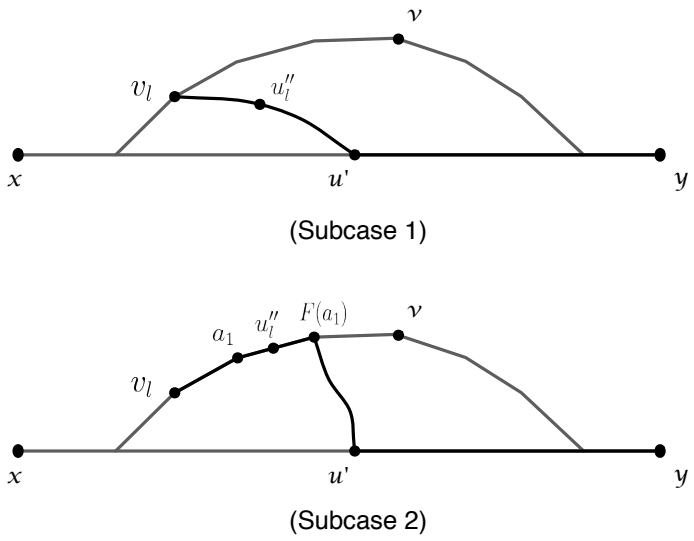


Figure 5.8: The illustration of subcases 1 and 2. The black line is $v_l y$.

□

Corollary 5.5. *Suppose we have known that* $v \in xy \diamond [x', y']$. *For any* $\hat{x} \in \{x, \Delta, w, c_l, c_{bl}, c_{bl}^f, c_{br}^f, c_{br}^L\}$ *and* $\hat{y} \in \{y, \nabla, w', c_r, c_{br}, c_{br}^l, c_{bl}^l, c_{bl}^R\}$ *on the path* $xy \diamond [x', y']$, *if* $\hat{x}v \diamond u$ *is a subpath of* $xy \diamond [x', y']$, *then we can find* $(xy \diamond [x', y']) \diamond [v, \hat{y}]$ *in* $O(\log n)$ *time. Symmetrically, if* $v\hat{y} \diamond u$ *is a subpath of* $xy \diamond [x', y']$, *then we can find* $(xy \diamond [x', y']) \diamond [\hat{x}, v]$ *in* $O(\log n)$ *time.*

We only need to consider one of v_l or v_r and replace the detour $(xy \diamond [x', y']) \diamond [a_i(F(a_i)), b_i(F(b_i))]$ with $(xy \diamond [x', y']) \diamond [a_i(F(a_i)), \hat{y}]$ or $(xy \diamond [x', y']) \diamond [\hat{x}, b_i(F(b_i))]$.

Now we consider the detour $(xy \diamond [x', y']) \diamond v$ by different case. Recall that we can find $u_l = \rho_u(xu)$ and $u_r = \rho_u(uy)$ on xy. The 3 possible type of detours are:

(i) The detour that reaches some point in $(u, u_r]$.

(ii) The detour that reaches some point in $[u_l, u)$.

(iii) The detour that avoids the range $[u_l, u_r]$ in xy.

The first and second types are reducible to Case I. For the third type, we consider

the different cases on whether the path $xy \diamond u$ goes through x' or y'. Recall that $x' = \rho_x(xu)$, $y' = \rho_y(uy)$. There are 4 possibilities of their locations:

- **Case II.1** If $xy \diamond u$ does not go through x' or y',
- **Case II.2** If $xy \diamond u$ goes through x' but not y',
- **Case II.3** If $xy \diamond u$ goes through y' but not x',
- **Case II.4** If $xy \diamond u$ goes through both x' and y',

5.5.2.1 Case II.1

If $xy \diamond u$ does not go through x' or y', which means that Δ of $xy \diamond u$ is before x' and ∇ of $xy \diamond u$ is after y'. so $xy \diamond u = xy \diamond [x', y']$. We can easily check whether v is in $xy \diamond [x', y']$ by checking whether $v \in xy \diamond u$. If $v \in xy \diamond [x', y']$, just call the procedure of Theorem 5.4 with $\hat{x} = \Delta$ and $\hat{y} = \nabla$.

5.5.2.2 Case II.2

If $xy \diamond u$ goes through x' but not y', we make use of the point c_l. Of course, if $v \notin xy \diamond u$, it is trivial. In the case that $v \in xy \diamond u$, recall that c_l is the first vertex on the range (Δ, ∇) of the path $xy \diamond [x', y']$ satisfying:

$$\exists u' \in [x', y'], \text{ such that } x', c_l \in xy \diamond u', \text{ and } y' \notin xy \diamond u'$$

From the definition of c_l, v cannot be in the range $[x, c_l)$ in $xy \diamond [x', y']$. Since $y' \notin xy \diamond u'$, $xy \diamond u'$ does not go through any vertex in $u'y'$, so $c_l y \diamond u'$ is the subpath of $xy \diamond [x', y']$ from c_l to y.

We check whether $v \in xy \diamond u'$. Note that in the structure u' is stored with c_l. There are three possibilities:

(i) If $v \notin xy \diamond u'$, then v is not in the range $[c_l, y]$ in $xy \diamond [x', y']$. Since v is also not in the range $[x, c_l)$, it follows that $v \notin xy \diamond [x', y']$. This case is trivial.

(ii) If $v \in xy \diamond u'$ and v is before c_l on that path, then $v \notin c_l y \diamond u'$, so v is also not in $xy \diamond [x', y']$.

(iii) If $v \in c_l y \diamond u'$ and $\|c_l v \diamond u\| = \|c_l v \diamond u'\|$, so v is on $xy \diamond [x', y']$ and it is equal to or after c_l. Thus $c_l v \diamond u$ is a subpath of $xy \diamond [x', y']$. Since v is on $xy \diamond u$ and y' is not on $xy \diamond u$, $vy \diamond u$ is also a subpath of $xy \diamond [x', y']$. So we can the procedure of Theorem 5.4 with $\hat{x} = c_l$ and $\hat{y} = \nabla$.

(iv) If $v \in c_l y \diamond u'$ and $\|c_l v \diamond u\| \neq \|c_l v \diamond u'\|$, then v is on $xy \diamond [x', y']$ and $u \neq u'$. Since $c_l v \diamond u'$ is a subpath of $xy \diamond [x', y']$, $u \notin c_l v \diamond u'$, so $c_l v \diamond u$ must go through u'. We now consider the relativeposition of u and u':

If u is before u' in xy, the path $(c_l u' \diamond u) \cdot u' y$ must be shorter than $(c_l v \diamond u) \cdot (vy \diamond u)$, since $c_l u' \diamond u$ is a subpath of $c_l v \diamond u$. Also $c_l v \diamond u$ is shorter than the subpath from c_l to v on $xy \diamond [x', y']$. So the path from x to c_l through $xy \diamond [x', y']$ concatenating the path $(c_l u' \diamond u) \cdot u' y$ will be shorter than $xy \diamond [x', y']$, which does not go through v and can be covered by the path from x to y' in Case I. See Figure 5.9.

If u is after u' in xy, it is easy to see that u is not on xc_l which goes through x'. So the shortest path from x to c_l avoiding u will go through x', which can be covered by the path from u_l which is reducible to Case I. Thus we only do not need to consider the detour $(xy \diamond [x', y']) \diamond [c_l, v]$ by the Corollary 5.5 with $\hat{x} = c_l$ and $\hat{y} = \nabla$.

Case II.3 is symmetric to Case II.2.

5.5.2.3 Case II.4

In this case $xy \diamond u$ goes through both x' and y'. If $v \in xy \diamond u$, recall that c_{bl} is the first vertex on the range $[\Delta, \nabla]$ in the path $xy \diamond [x', y']$ satisfying:

$$\exists u' \in [x', y'], \text{ such that } x', y', c_{bl} \in xy \diamond u'$$

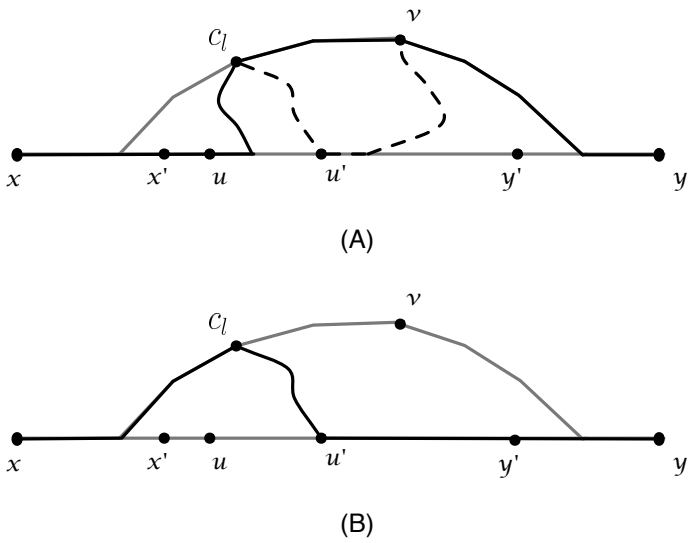


Figure 5.9: When u is before u' and $c_l v \diamond u$ goes through u' in Case II.2, as shown in the dash line of (A), we can see $c_l v \diamond u$ is shorter than the subpath from c_l to v in $xy \diamond [x', y']$. So the black line in (B) is shorter than $xy \diamond [x', y']$, and it goes through y' so can be obtained by Case I.

From this definition, since $x', y', v \in xy \diamond u$, v is not in the range $[x, c_{bl})$ on $xy \diamond [x', y']$. Then we check the relative position of v in the path $xy \diamond u'$. See Figure 5.6.

- Suppose that v is in the range (Δ, c^F_{bl}) or the range (c^L_{bl}, ∇) in the path $xy \diamond u'$, where Δ and ∇ are w.r.t. the detour $xy \diamond u'$. Since these ranges are disjoint with $xy \diamond [x', y']$, we can guarantee that $v \notin xy \diamond [x', y']$.

- If v is in the range $[c^F_{bl}, c^r_{bl}]$ and $\|c^F_{bl} c^r_{bl} \diamond u\| = \|c^F_{bl} c^r_{bl} \diamond u'\|$, then v is on $xy \diamond [x', y']$ and $c^F_{bl} c^r_{bl} \diamond u$ is a subpath of $xy \diamond [x', y']$. Thus we can call the procedure of Theorem 5.4 with $\hat{x} = c^F_{bl}$ and $\hat{y} = c^r_{bl}$.

- If v is in the range $[c^F_{bl}, c^r_{bl}]$ and $\|c^F_{bl} c^r_{bl} \diamond u\| \neq \|c^F_{bl} c^r_{bl} \diamond u'\|$, we can see $u' \in c^F_{bl} c^r_{bl} \diamond u$. If $v \notin c^F_{bl} c^r_{bl} \diamond u$, there will be a path from x' to y or x to y' avoiding u, v which is shorter than $xy \diamond [x', y']$. Otherwise one of $c^F_{bl} v \diamond u$ and $c^F_{bl} v \diamond u$ must be a subpath of $xy \diamond [x', y']$. Consider the relative position of u' and v on $c^F_{bl} c^r_{bl} \diamond u$

and the relative position of u and u' on xy. If u' is after v on $c_{bl}^F c_{bl}^r \diamond u$ and u is before u' on xy, then the path $c_{bl}^r y \diamond u'$ does not go through u, v, so the path reaches c_{bl}^r will reach y'. So we call the procedure of Corollary 5.5 to find the path $(xy \diamond [x', y']) \diamond [v, c_{bl}^r]$. u' on xy. If u' is before v on $c_{bl}^F c_{bl}^r \diamond u$ and u is after u' on xy, then the path $x c_{bl}^F y \diamond u'$ does not go through u, v, so the path reaches c_{bl}^F will reach x'. So we call the procedure of Corollary 5.5 to find the path $(xy \diamond [x', y']) \diamond [c_{bl}^F, v]$. When other cases happen, there must be a path reaching x' or y' shorter than $xy \diamond [x', y']$, which are similar to the case shown in Figure 5.9, so we do not need to consider those cases.

- When v is in $[c_{bl}, c_{bl}^L]$, it is symmetric to the case of $v \in [c_{bl}^F, c_{bl}^r]$.

- If v is in the range (c_{bl}^r, c_{bl}) and u is before or equal to u', then v cannot be before c_{bl}^L or after c_{bl}^F on $xy \diamond [x', y']$. Then u and v are not on the path from x to c_{bl}^L through $xy \diamond [x', y']$ and then through $c_{bl}^L y \diamond u'$, which is shorter than $xy \diamond [x', y']$ and goes through y' Thus, we do not need to consider the detour from x to y avoiding $[x', y']$.

- In a similar fashion, we do not need to consider the case when v is in the range (c_{bl}^r, c_{bl}) and u is after u', since the path $(x c_{bl}^F \diamond u')$ then from c_{bl}^F to y through $xy \diamond [x', y']$ is shorter than $xy \diamond [x', y']$.

- If v is not in $xy \diamond u'$ and u is before or equal to u', then the path from x to c_{bl}^L through $xy \diamond [x', y']$ concatenating $c_{bl}^L y \diamond u')$ does not contain v and is shorter than $xy \diamond [x', y']$, so we can abandon this case.

- Now we consider the last case: if v is not in $xy \diamond u'$ and u is after u', then we know that $u \notin x c_{bl} \diamond u'$ and $x c_{bl} \diamond u$ will go through x'. Then we perform a symmetric procedure for c_{br}, if it is not in the last case, then we can solve Case II.4 directly. If it is in the last case ($v \notin xy \diamond u''$ and u is before u''), we will have

$c_{br}y \diamond u$ will go through y'. Thus the detour $(xy \diamond [x', y']) \diamond [c_{bl}, c_{br}]$ in $B_2(x, y)$ can cover this case since if the shortest detour goes through c_{bl} or c_{br}, it must go through x' or y'.

5.6 Case III: Two failed vertices on xy

In this case both u and v are on the original shortest path from x to y, where u is *before* v in xy. In Section 5.6.1 we consider the situation where $|xu|$ or $|vy|$ is a power of 2; these queries are easily reducible to several Case I queries. However, in general we will need to use a fundamentally different approach to answering such queries. In Section 5.6.2 we introduce a *binary partition* data structure that is tailored to Case III queries and in Section 5.6.3 we give the complete Case III query algorithm.

5.6.1 If $|xu|$ or $|vy|$ is a power of 2

W.l.o.g, we only consider the case where $|xu|$ is a power of 2 and $v \in xy \diamond u$. As in Section 5.4.2 we find $v_l = \rho_v(\nabla v)$ and $v_r = \rho_v(vy)$, where ∇ is the convergence point of the paths $xy \diamond u$ and xy. The shortest detour belongs to one of the following types:

(i) The detour that avoids u and the range $[v_l, v_r]$ in $xy \diamond u$.
(ii) The detour that reaches some points in $(v, v_r]$.
(iii) The detour that reaches some point in $[v_l, v)$, but does not reach $(v, v_r]$.

The first type can be covered by $B_2(xy)$ as shown in Section 5.4.2, and the second type can also covered by $B_2(xv_r)$ since $(xv_r \diamond u) \diamond v$ is in B_2. The third type is reducible to Case I since $|v_l v|$ is a power of 2.

5.6.2 The binary partition structure

When both failed vertices lie on the shortest path xy we need to consider the possibility that the optimal detour departs from xy before u and returns to xy between

u and v, possibly departing and returning several times. If we could identify with certainty just *one* vertex m between u and v that lies on $xy \diamond \{u, v\}$, we could reduce our Case III query to two Case II queries: $xm \diamond \{u, v\}$ and $my \diamond \{u, v\}$. The binary partition structure allows us to answer a Case III query directly or reduce it to Case II queries. For each x, y and $i, j \leq \lfloor \log |xy| \rfloor$, we store the following structure $C_{i,j}(x, y)$:

Let $[x', y'] = [x \oplus 2^i, y \ominus 2^j]$. Define the following points on $[x', y']$:

$$m_{q,r} \in x'y', \qquad \text{such that } |x'm_{q,r}| = \left\lfloor \frac{r}{2^q} |x'y'| \right\rfloor,$$
$$\text{for all } 1 \leq q \leq \lceil \log |x'y'| \rceil, \;\; 0 \leq r \leq 2^q$$

These points define the following ranges:

$$R_{q,0} = [m_{q,0}, m_{q,1}], \forall 1 \leq q \leq \lceil \log |x'y'| \rceil$$

and

$$R_{q,r} = (m_{q,r}, m_{q,r+1}], \forall 1 \leq q \leq \lceil \log |x'y'| \rceil, 1 \leq r \leq 2^q - 1$$

where $R_{q,2^q-1}$ is truncated at y'. See Figure 5.10.

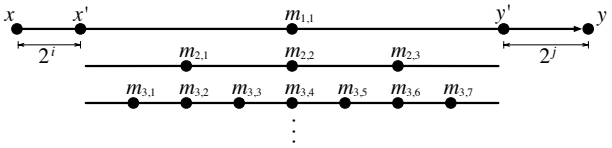


Figure 5.10: Different levels of the binary structure

Thus, in level q, we have 2^q disjoint ranges $R_{q,0}, R_{q,1}, ..., R_{q,2^q-1}$, and their union is the whole range $[x', y']$. For every level q, we store in $C_{i,j}(x, y)$ the length and number of vertices of the following paths. (Below the superscripts are mnemonics, where e, o, l, f, and b are for even, odd, last, first, and backwards.) The space needed for this structure is $O(n^2 \log^3 n)$.

(i) p_q^e:

$$p_q^e = \min_{\substack{r \in [0,2^q) \\ r \text{ even}}} xy \diamond (x'y' \setminus R_{q,r})$$

Let r_q^e be the index r for p_q^e. That is, among all paths from x to y that intersect *only one* of the even intervals, we store the one with minimum length. Define l_q^e to be the leftmost vertex of p_q^e in the range R_{q,r_q^e}, that is, $l_q^e \in p_q^e \cap R_{q,r_q^e}$ that minimizes $|xl_q^e|$.

(ii) p_q^o:

$$p_q^o = \min_{\substack{r \in [0,2^q) \\ r \text{ odd}}} xy \diamond (x'y' \setminus R_{q,r})$$

Let r_q^o be the index r for p_q^o. Store s_q^o as the rightmost vertex of p_q^o in the range R_{q,r_q^o}.

(iii) p_q^{el}: Define the last vertex on p_q^e which is in the subrange R_{q,r_q^e} as L_q^e. Store the path:

$$p_q^{el} = xy \diamond (x'y' \setminus (L_q^e, m_{q,r_q^e+1}])$$

i.e., p_q^{el} may only use vertices in the range $(L_q^e, m_{q,r_q^e+1}]$.

(iv) p_q^{of}: Define the first vertex on p_q^o which reaches the subrange R_{q,r_q^o} as F_q^o. Store the path:

$$p_q^{of} = xy \diamond (x'y' \setminus (m_{q,r_q^o}, F_q^o))$$

Parts 5-9 will use the following notation:

Let S and T be two disjoint adjacent subpaths in $x'y'$, where S precedes T, and let $X = xx', Y = y'y$. Let X' be the subpath between X and S and Y' be the subpath between T and Y. See Figure 5.11. Obviously X, X', S, T, Y', Y are disjoint and form the path xy. Define the path $D(S,T)$ to be:

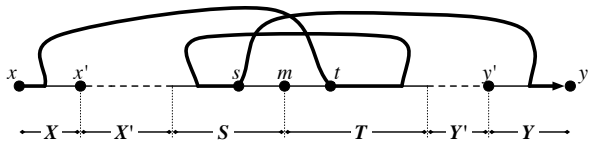


Figure 5.11: The form of the path $D(S,T)$. Here S,T are arbitrary adjacent intervals on xy and $X = xx'$ and $Y = y'y$.

$$D(S,T) = \min_{s\in S, t\in T} (xt\diamond(X'\cup S))\cdot(ts\diamond(st\cup X\cup X'\cup Y'\cup Y\setminus\{t,s\}))\cdot(sy\diamond(T\cup Y'))$$

That is, $D(S,T)$ is the shortest path from x to y that passes through T then S, and that never returns to T and avoids all other vertices in $x'y'$.

(v) p_q^b:

$$p_q^b = \min_{\substack{r\in[0,2^q-2]\\ r \text{ even}}} D(R_{q,r}, R_{q,r+1})$$

Let r_q^b denote the index r for p_q^b. Store l_q^b and s_q^b as the leftmost and rightmost vertex of p_q^b in the range $R_{q,r_q^b}\cup R_{q,r_q^b+1}$.

(vi) p_q^{bf}: Define the first vertex on p_q^b which reaches the subrange R_{q,r_q^b+1} as F_q^b, and store

$$p_q^{bf} = D(R_{q,r_q^b}, (m_{q,r_q^b+1}, F_q^b))$$

I.e., it further avoids the range $[F_q^b, m_{q,r_q^b+2}]$ from p_q^b. Store l_q^{bf} as the leftmost vertex of p_q^{bf} in the range R_{q,r_q^b}.

(vii) p_q^{bfl}: Let the last vertex on p_q^{bf} in the subrange R_{q,r_q^b} be L_q^{bf} and store the path:

$$p_q^{bfl} = D((L_q^{bf}, m_{q,r_q^b+1}], (m_{q,r_q^b+1}, F_q^b))$$

Figure 5.13 in Section 5.6 illustrates this path.

(viii) p_q^{bl}: Let the last vertex on p_q^b in the subrange R_{q,r_q^b} be L_q^b and store the path:

$$p_q^{bl} = D((L_q^b, m_{q,r_q^b+1}], R_{q,r_q^b+1})$$

Store s_q^{bl} as the rightmost vertex of p_q^{bl} in the range R_{q,r_q^b+1}.

(ix) p_q^{blf}: Define the first vertex on p_q^{bl} which is in the subrange R_{q,r_q^b+1} to be F_q^{bl}, and store:

$$p_q^{blf} = D((L_q^b, m_{q,r_q^b+1}], (m_{q,r_q^b+1}, F_q^{bl})).$$

5.6.3 General Cases

We find $u_l = \rho_u(xu)$ and $v_r = \rho_v(vy)$ in constant time. The optimal detour can belong to one or more of the following types:

- III.1 The detour that reaches some point in $(v, v_r]$.
- III.2 The detour that reaches some point in $[u_l, u)$.
- III.3 The detour that avoids $[u_l, v_r]$
- III.4 The detour that avoids $[u_l, u]$ and $[v, v_r]$ in xy, but reaches some vertex between (u, v).

The first and second are considered in Section 5.6.1. The third one can also be covered by finding $x' = \rho_x(xu)$ and $y' = \rho_y(vy)$ and then returning $xy \diamond [x', y'] \in$

$B_1(x, y)$. However, things become more complicated when we consider the fourth case, which means the detour leaves xy *before* u_l and merges with xy *after* v_r and goes through some vertex between u and v. To deal with this case, we will need the binary partition structure introduced in the previous subsection.

Now consider the positions of u and v. Find the smallest level q in $C_{i,j}(x, y)$ ($i = \log\lfloor|xx'|\rfloor$, $j = \log\lfloor|y'y|\rfloor$) in which u and v are not in the same subrange. (This can be achieved by computing $|xu|$ and $|xv|$.) Let $u \in R_{q,r}$ and $v \in R_{q,r+1}$, where r is even. (If r is odd, then u and v are also in different subranges in level $q-1$.) Denote the rightmost vertex of $R_{q,r}$ by m. There are 4 possible types for detour *III.4*:

- III.4.a The shortest detour only goes through the vertices in $R_{q,r}$.
- III.4.b The shortest detour only goes through the vertices in $R_{q,r+1}$,
- III.4.c The shortest detour goes through some vertices in $R_{q,r}$, then to some vertices in $R_{q,r+1}$.
- III.4.d The shortest detour goes through some vertices in $R_{q,r+1}$, then to some vertices in $R_{q,r}$ but does not reach m.

In Case III.4.a there are some possible subcases depending on the relative positions of u and the path p_q^e. See Figure 5.12.

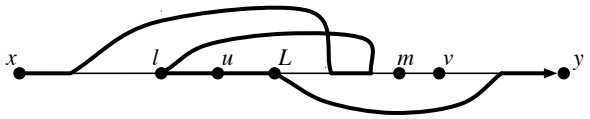


Figure 5.12: The illustration of the positions of u, L and m.

- III.4.a.i If p_q^e does not go through $R_{q,r}$ in $C_{i,j}(x, y)$, then there exists another path that only goes through R_{q,r_q^e} disjoint to $R_{q,r}$ but shorter than any path

only going through $R_{q,r}$. So p_q^e goes through some vertices in $[x', y']$ but does not touch the range $[u, v]$ in xy. Thus, it has already been covered by Cases III.1 or III.2, as we discussed above.

- III.4.a.ii If L_q^e is *before* u in xy, then p_q^e must be longer than $\|xL_q^e\| + \|L_q^e y \diamond [u, v]\|$, which will go through u_l. This possibility was dealt with in Case III.2.

- III.4.a.iii If u is *before* l_q^e, p_q^e is the shortest detour for Case III.4.a. (Remember here l_q^e is the leftmost vertex of p_q^e in the range R_{q,r_q^e}.)

- III.4.a.iv If $u \in [l_q^e, L_q^e]$, there are two types of detours depending on whether the shortest detour goes through the range $(u, L_q^e]$. From the definition of p_q^e, a shortest path that travels through some vertices in $(u, L_q^e]$ must travel through L_q^e. Thus, $xy \diamond \{u, v\}$ will be the concatenation of the paths from x to L_q^e and from L_q^e to y avoiding u and v, which are both in Case II. For the detours not going through the range $(u, L_q^e]$, p_q^{el} can cover this case.

The Case III.4.b is symmetric to Case III.4.a: just replace p_q^e by p_q^o, L_q^e by F_q^o, l_q^e by s_q^o, and $R_{q,r}$ by $R_{q,r+1}$. For the Case III.4.c, the shortest detour must go through the vertex m which separates these two ranges $R_{q,r}$ and $R_{q,r+1}$. We find the paths from x to m and from m to y avoiding u and v, which are both in Case II.

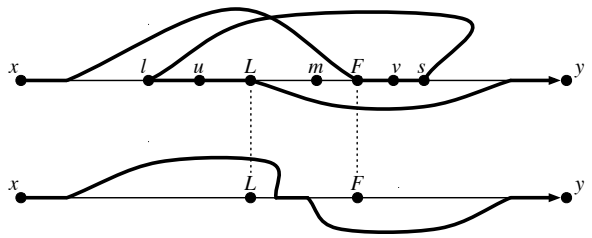


Figure 5.13: The fourth type in Case III

For Case III.4.d, there are some possible subcases depending on the relative positions of u, v and the path p_q^b. See Figure 5.13:

- III.4.d.i If p_q^b does not go through $R_{q,r}$ or $R_{q,r+1}$, i.e., $r \neq r_q^b$, then the shortest detour has already been covered by Cases III.1 or III.2

- III.4.d.ii If L_q^b is *before* u or F_q^b is *after* v in xy, we have already considered this situation in Cases III.1 and III.2.

- III.4.d.iii If $u \in [l_q^b, L_q^b]$, then any detours that reach some vertex in $(u, L_q^b]$ will go through L_q^b. To cover the possibility that the shortest detour goes through some vertices in $(u, L_q^b]$, we find the detours from x to L_q^b and from L_q^b to y avoiding u and v, which are both in Case II. To cover the possibility that the shortest detour avoids $(u, L_q^b]$, we can see p_q^{bl} satisfies this condition. Then in the path p_q^{bl}, there are some subcases:

  - If v is *after* s_q^{bl} in xy, p_q^{bl} is the shortest detour for this case.
  - If F_q^{bl} is *after* v in xy, p_q^{bl} must be longer then $\|xF_q^{bl} \diamond [u, v]\| + \|F_q^{bl}y\|$, which will go through v_r. This situation has been covered by Case III.2.
  - If $v \in [F_q^{bl}, s_q^{bl}]$, then any detours which reach some vertex in $[F_q^{bl}, v)$ will go through F_q^{bl}, so it can be covered by $xF_q^{bl} \diamond \{u, v\} \cdot F_q^{bl}y \diamond \{u, v\}$, which are both in Case II. Furthermore we can use the path p_q^{blf} to cover the case in which it does not go through F_q^{bl}.

- III.4.d.iv If $v \in [F_q^b, s_q^b]$, it is symmetric to the Case III.4.c.iii.

- III.4.d.v If u is *before* l_q^b and v is *after* s_q^b, then p_q^b is just the shortest detour for III.4.d.

This concludes the query algorithm for Case III. The total running time will be $O(\log n)$, which comes from the auto-reductions in Case I and the time needed to locate v_l and v_r.

CHAPTER VI

Dynamic Subgraph Connectivity Oracles

In the first part of this chapter, we will describe a worst-case dynamic subgraph connectivity structure with $\tilde{O}(m^{4/5})$ vertex update time and $\tilde{O}(m^{1/5})$ query time. We will utilize the worst-case edge update structure [28] as a component and maintain a multi-level hierarchy instead of the two-level one in [10]. In general, we will get faster update time for this structure if there are faster worst-case edge update connectivity structures. In the second part of this chapter, we will describe a new linear space subgraph connectivity structure with $\tilde{O}(m^{2/3})$ amortized vertex update time and $\tilde{O}(m^{1/3})$ query time. [1]

Techniques. The best worst-case edge update connectivity structure [28] so far has $O(n^{1/2})$ update time, much larger than the polylogarithmic amortized edge update structure [38, 59]. Inspired by [10], we will divide vertices into several levels by their degrees. In "lower" levels having small degree bounds, we maintain an edge update connectivity structure for the subgraph on active vertices at these levels. In "higher" levels having large degree bounds and small numbers of vertices, we only keep the subgraph at those levels and run a BFS to obtain all the connected components after an update. To reflect the connectivity between high-level vertices through low-level vertices, we will add two types of artificial edges to the high-level vertices. (a). In

[1] These results appears in my paper "New Data Structures for Subgraph Connectivity" [20] in ICALP 2010.

the "path graph", update on every vertex will change the edge set, but the number of edges changed is only linear to the degree of that vertex. (b). In the "complete graph", only low-level vertex updates will change the edge set, but the number of edges changed is not linear to the degree. In our structure, we only use the "complete graph" between top levels and bottom levels to bound the update time.

6.1 Basic Structures

In this section, we will define several dynamic structures as elements of the main structures. If we want to keep the connectivity of some vertex set V_1 through a disjoint set V_0, some "artificial edges" may need to be added into V_1. For every spanning tree in V_0, the vertices in V_1 adjacent to this spanning tree need to be connected. We will use the ET-tree ideas from Henzinger and King [37] to make such artificial edges efficiently dynamic when the spanning forest of V_0 changes. Here the artificial edges of V_1 associated with a spanning tree in V_0 will form a path ordered by the Euler Tour of that tree.

6.1.1 Euler Tour List

For a tree T, let $L(T)$ be a list of its vertices encountered during an Euler tour of T [37], where we only keep *any one* of the occurrences of each vertex. Note that $L(T)$ can start at any vertex in T. Now we count the number of cut/link operations on the Euler tour lists when we cut/link trees. One may easily verify the following theorem:

Theorem 6.1. *When we delete an edge from T, T will be split into two subtrees T_1 and T_2. We need at most 2 "cut" operations to split $L(T)$ into 3 parts, and at most 1 "link" operation to form $L(T_1)$ and $L(T_2)$.*

When we add an edge to link two tree T_1 and T_2 into one tree T, then we need

to change the start or end vertices of $L(T_1)$ and $L(T_2)$ and link them together to get $L(T)$, which will take at most 5 "cut/link" operations.

6.1.2 Adjacency Graph

In a graph $G = (V, E)$, let $V_0, V_1, V_2, ..., V_k$ be disjoint subsets of V, and let F be a forest spanning the connected components of the subgraph of G induced by the active vertices of V_0. We will construct a structure $R(G, F, V_1, V_2, ..., V_k)$ containing artificial edges on the active vertices of the sets $V_1, V_2, ..., V_k$ which can represent the connectivity of these vertices through V_0.

Definition 6.2. For $1 \leq i \leq k$, the *active adjacency list* $A_G(v, V_i)$ of a vertex $v \in V_0$ is the list of active vertices in V_i which are adjacent to v in G. The *active adjacency list* $A_G(T, V_i)$ induced by a tree $T \in F$ is the concatenation of the lists $A_G(v_1, V_i), A_G(v_2, V_i), ..., A_G(v_k, V_i)$ where $L(T) = (v_1, v_2, ..., v_k)$. Note that a vertex of V_i can appear multiple times in $A_G(T, V_i)$.

Definition 6.3. Given a list $l = (v_1, v_2, ..., v_k)$ of vertices, define the edge set $P(l) = \{(v_i, v_{i+1}) | 1 \leq i < k\}$.

Definition 6.4. In the structure $R(G, F, V_1, V_2, ..., V_k)$, for a tree $T \in F$, we maintain the list $A_G(T)$ of active vertices which is the concatenation of the lists $A_G(T, V_1)$, $A_G(T, V_2), ..., A_G(T, V_k)$. Then the set of artificial edges in $R(G, F, V_1, V_2, ..., V_k)$ is the union $\bigcup_{T \in F} P(A_G(T))$. We call the edges connecting different $A_G(T, V_i)$ $(1 \leq i \leq k)$ "inter-level edges". So the degree of a vertex v of V_i $(1 \leq i \leq k)$ in $R(G, F, V_1, V_2, ..., V_k)$ is at most twice its degree in G, and the space of this structure is linear to G.

We can see that deleting a vertex in l will result in deleting at most two edges and inserting at most one edge in $P(l)$, and inserting a vertex in l will result in inserting

at most two edges and deleting at most one edge in $P(l)$. Also, one can easily verify the following properties of the adjacency graph:

Note 6.5. *For a spanning tree $T \in F$, the vertices in $A_G(T, V_i)$ are connected by the subset of $R(G, F, V_1, V_2, ..., V_k)$ induced only by V_i, for all $1 \leq i \leq k$.*

Lemma 6.6. *For any two active vertices u, v in $V_1 \cup V_2 \cup ... \cup V_k$, if there is a path with more than one edge connecting them, whose intermediate vertices are active and in V_0, then they are connected by the edges $R(G, F, V_1, V_2, ..., V_k)$.*

Also if u, v are connected in $R(G, F, V_1, V_2, ..., V_k)$, they are connected in the subgraph of G induced by the active vertices.

Lemma 6.7. *The cost needed to maintain this structure:*

(i) Making a vertex v active or inactive in $V_1 \cup V_2 \cup ... \cup V_k$ will require inserting or deleting at most $O(min(deg_G(v), |V_0|))$ edges in $R(G, F, V_1, V_2, ..., V_k)$. (Here $deg_G(v)$ denotes the degree of v in the graph G.)

(ii) Adding or removing an edge in F will require inserting or deleting $O(k)$ edges to this structure.

(iii) Making a vertex $v \in V_0$ active or inactive will require inserting or deleting $O(k \cdot deg_G(v))$ edges.

(iv) Inserting or deleting an "inter-level" edge (u, v) in G where $u \in V_0$, $v \in V_1 \cup V_2 \cup ... \cup V_k$ will require inserting or deleting at most 3 edges in $R(G, F, V_1, V_2, ..., V_k)$. ($G$ may be not the original graph, but another dynamic graph.)

6.1.3 ET-list for adjacency

Here we describe another data structure for handling adjacency queries among a dynamic spanning tree F and a disjoint vertex set V_1. By this structure, when we

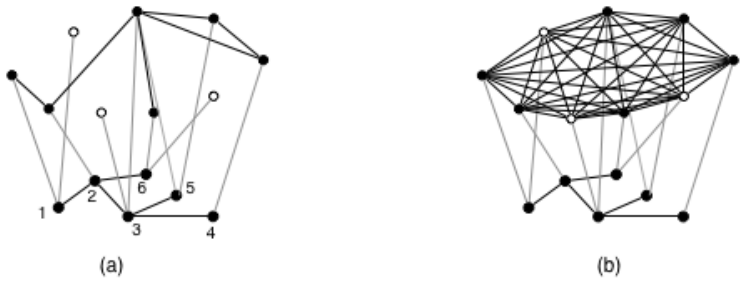


Figure 6.1: In this figure, the black points denote active vertices while the white points denote inactive vertices. Here we show a tree T and a set of vertices in V_1 adjacent to T. The figure (a) shows the edge set $R(G, T, V_1)$, in which the number on vertices shows the order of vertices in $L(T)$. We can see the artificial edges added to V_1 can reflect the connectivity through T between vertices of V_1, and the degree of a vertex v in V_1 in this edge set is linear to the degree of v in G. The figure (b) shows a complete graph which reflect the connectivity through T on V_1 used in [10] and in $\bar{E}_T$ in this chapter. So we do not need to change any edges in (b) when switching a vertex in V_1, but we may change most edges when updating a vertex in T.

intend to obtain all the vertices in V_1 adjacent to a tree $T \in F$, we do not need to check all the edges connecting T to V_1, but only check whether v is adjacent to T for all $v \in V_1$. This takes $O(|V_1|)$ time for finding all such vertices. Note that since this structure keeps all the vertices in V_1 no matter whether they are active or not, so we do not need to update it when switching a vertex in V_1.

Theorem 6.8. *Let $G = (V, E)$ be a graph and V_0, V_1 be two disjoint subsets of V. Let F be a spanning forest on the subgraph of G induced by the active vertices of V_0. There is a data structure $ET(G, F, V_1)$ with linear size that accepts edge inserting/deleting updates in F. Given a vertex $v \in V_1$ (active or inactive) and a tree $T \in F$, we can answer whether they are adjacent in G in constant time. The update time for a vertex v in V_0 of this structure is $O(deg_G(v)|V_1|)$.*

Proof. In $ET(G, F, V_1)$, for every vertex $v \in V_1$ and every $T \in F$, we keep a list of vertices in T adjacent to v ordered by $L(T)$. From Theorem 6.1, when we link two trees or cut a tree into two subtrees in F, it takes $O(V_1)$ time to merge or split the

lists for all $v \in V_1$. When a vertex in V_0 is turned active or inactive, we need to add/delete $deg_G(v)$ edges in F and add/delete that vertex in the lists for all $v \in V_1$. The space will be $O(m)$ since every edge will contribute at most one appearance of vertex in the lists. □

6.2 Dynamic Subgraph Connectivity with Sublinear Worst-case Update Time

In this section, we will describe our worst-case dynamic subgraph connectivity structure with sublinear update time. We divide the vertices into several levels by their degrees. The structure of adjacency graph in Section 6.1.2 will be used to reflect the connectivity between high-level vertices through low-level vertices. We will use the dynamic spanning tree structure of $O(n^{1/2})$ worst-case edge update time [28] to keep the connectivity of vertices in low-levels of lower degree bounds. However, in high-levels with high degree bounds, we only store the active vertices and edges and run a BFS after each update to obtain the new spanning trees.

Theorem 6.9. *Given a graph $G = (V, E)$, there exists a dynamic subgraph connectivity structure occupying $\tilde{O}(m)$ space and taking $\tilde{O}(m^{6/5})$ preprocessing time. We can switch every vertex to be "active" or "inactive" in this structure in $\tilde{O}(m^{4/5})$ time, and answer the connectivity between any pair of vertices in the subgraph of G induced by the active vertices in $\tilde{O}(m^{1/5})$ query time.*

6.2.1 The structure

First we divide all the vertices of G into several parts based on their degrees in the whole graph G, so the sets are static.

- V_A: The set of vertices of degrees less than $m^{1/5}$
- V_B: The set of vertices v satisfying $m^{1/5} \leq deg_G(v) < m^{3/5}$.

- V_C: The set of vertices v satisfying $m^{3/5} \leq deg_G(v) < m^{4/5}$.

- V_D: The set of vertices v satisfying $deg_G(v) \geq m^{4/5}$.

So we can see that $|V_B| \leq 2m^{4/5}, |V_C| \leq 2m^{2/5}, |V_D| \leq 2m^{1/5}$.

In order to get more efficient update time, we continue to partition the set V_B into $V_0, V_1, V_2, ..., V_k$ where $k = \lfloor \frac{2}{5} \log_2 m \rfloor$ and:

$$V_i = \{v | v \in V_B, 2^i m^{1/5} \leq deg_G(v) < 2^{i+1} m^{1/5}\}, \forall 0 \leq i \leq k \tag{6.1}$$

Thus, $|V_i| \leq 2^{1-i} m^{4/5}$. For all the disjoint vertex sets $V_A, V_0, V_1, ..., V_k, V_C, V_D$ ordered by their degree bounds, we say that a vertex u is *higher than* a vertex v if u is in the set of higher degree bound than v.

For the set V_A, the following structure will be built to keep the connectivity between vertices in other sets through vertices of V_A:

- Maintain a dynamic spanning forest F_A on the subgraph of G induced by the active vertices of V_A, which will support $O(\sqrt{n})$ edge update time. [28]

- Maintain the edge set (and the structure) $E_A = R(G, F_A, V_0, V_1, ..., V_k, V_C)$.

- Maintain the structures $ET(G, F_A, V_C)$,$ET(G, F_A, V_D)$ so that we can find the vertices of V_C and V_D (including active and inactive) adjacent to a tree T of F_A in G in $O(|V_C|)$ time by Theorem 6.8. Denote the vertices of V_C and V_D adjacent to T by $V_C(T)$ and $V_D(T)$, respectively.

- For every spanning tree $T \in F_A$, arbitrarily choose an **active** vertex $u_T \in V_B$ which is adjacent to T in G (if there is one). Call it the "representative" vertex of T. Define the edge set $\bar{E}_T = \{(u, v) | u \in V_C(T) \cup V_D(T), v \in V_D(T)\} \cup \{(u_T, v) | v \in V_D(T)\}$.

- Define $G_0 = (V, E \cup E_A \cup \bigcup_{T \in F_A} \bar{E}_T)$. Note that E_A only contains edges connecting active vertices, but $\bar{E}_T$ may contain edges associate with inactive vertices. When considering the connectivity of G_0, we only consider the subgraph of G_0 induced by the active vertices and ignore the inactive vertices.

We have added artificial edges on the vertices of V_B, V_C, V_D to G_0 so that the subgraph of G_0 induced by the active vertices of these sets can represent the connectivity in the dynamic graph G. Note that we do not store the set $\bar{E}_T$ for every $T \in F_A$, but only store the final graph G_0 to save space. We can get every $\bar{E}_T$ efficiently from the adjacency lists.

Then we will build structures for the connectivity on $V_B \cup V_C \cup V_D$ through $V_0, ..., V_k$. For $i = 0$ to k, perform the following two steps:

(i) Maintain a dynamic spanning forest F_i on the subgraph of G_i induced by the active vertices of V_i. The structure will support $O(\sqrt{|V_i|}) = O(m^{2/5}/2^{i/2})$ edge update time. [28]

(ii) Maintain the edge set $E_{i+1} = R(G_i, F_i, V_{i+1}, ..., V_k, V_C, V_D)$, and define the graph $G_{i+1} = (V, E(G_i) \cup E_{i+1})$, where $E(G_i)$ is the set of edges in G_i.

We denote $H = G_{k+1}$ which contains all the artificial edge. Note that only the edges connecting vertices higher than V_i will be added to G_{i+1}, so the spanning forest F_i (F_A) still spans the connected components of the subgraph of H induced by the active vertices of V_i (V_A), and also $E_A = R(H, F_A, V_0, V_1, ..., V_k, V_C)$, $E_{i+1} = R(H, F_i, V_{i+1}, ..., V_k, V_C, V_D)$ for all $0 \leq i \leq k$.

Discussion: Why we need $\bar{E}_T$ but not simply construct $E_A = R(G, F_A, V_0, ..., V_k, V_C, V_D)$? Since there are no specific bounds for $|V_D|$ and the number of spanning trees in F_A, if $E_A = R(G, F_A, V_0, ..., V_k, V_C, V_D)$, from Lemma 6.7(1), the update time may become linear when we switch a vertex in V_D. Remind that $\bar{E}_T$

contains the edges connecting active and inactive vertices in V_D, so we do not need to change the edge sets $\bar{E}_T$ when switching a vertex of V_D.

When we consider the connectivity of vertices of V_C and V_D in H after an update, we just run a BFS on the subgraph of H induced by the active vertices of V_C and V_D which takes $O((|V_C| + |V_D|)^2) = O(m^{4/5})$ time and get a spanning forest F_{CD}. Due to page limit, some proofs of the following lemmas are omitted and will be given in the full version.

Lemma 6.10. *The space for storing H is $\tilde{O}(m)$, and it takes $\tilde{O}(m^{6/5})$ time to initialize this structure.*

Lemma 6.11. *(Consistency of $\bar{E}_T$) For any two active vertices $u \in V \setminus V_A, v \in V_D$, if there is a path longer than one connecting them whose intermediate vertices are all active and in V_A, then for some $T \in F_A$, they are connected by the subset of edges $\bar{E}_T \cup E_A$ induced by the active vertices.*

Proof. From the conditions, all the intermediate vertices on the path between u and v will be in the same spanning tree $T \in F_A$. So if $u \in V_C \cup V_D$, there is an edge connecting u and v in $\bar{E}(T)$. If $u \in V_B$ and $v \in V_D$, by Lemma 6.6, u will be connected to the representative vertex u_T in E_A, and there is an edge connecting u_T and v directly in $\bar{E}(T)$. □

From Lemma 6.6 and 6.11, the artificial edges in higher level generated by a spanning tree in a lower level can reflect the connectivity between active higher level vertices through this spanning tree. The subgraph of H induced by a subset will contain all the artificial edges and original edges of G, so it can reflect the connectivity in this subset and lower sets between its active vertices. We have the following lemma:

Lemma 6.12. *For any two active vertices u, v in the set V_i ($0 \leq i \leq k+1$) or higher, u and v are connected in the subgraph of H induced by the active vertices of $V_i \cup ... \cup V_k \cup V_C \cup V_D$ if and only if they are connected in the subgraph of G induced*

by the active vertices. Particularly for u, v in $V_C \cup V_D$, u and v are connected in the the subgraph of H induced by the active vertices of $V_C \cup V_D$ if and only if they are connected in the subgraph of G induced by the active vertices.

Proof. The "only if" part is obvious, since every artificial edge we add into H can reflect the connectivity in G from Lemma 6.6 and 6.11.

Turning to the "if" part, we prove the first statement by induction. For $i = 0$, for any two active $u, v \in V \setminus V_A$, if they are connected in G by a path p and all the intermediate vertices of p are in V_A, from Lemma 6.6 and Lemma 6.11, they are connected by $E_A \cup (\bigcup_{T \in F_A} \bar{E}_T)$ when p is longer than 1. And they are connected by E if p consists of a single edge. By concatenation, any p can be divided into such subpaths, so u and v are connected in the subgraph of G_0 (thus $H = G_{k+1}$) induced by the active vertices of V_B, V_C, V_D.

Suppose the statement holds for $i = q$, consider the case that $i = q + 1$. For any two active vertices u, v in the set V_{q+1} or higher, if they are connected in the dynamic graph G, by the inductive assumption, they are connected by a path p in H induced by the active vertices of $V_q, ..., V_k, V_C, V_D$. If all the intermediate vertices of p are in V_q, from Lemma 6.6, u and v are connected by E_{q+1} if p is longer than 1 or by a single edge in H otherwise. Similarly, by concatenation, the statement holds for $i = q + 1$. □

6.2.2 Switching a vertex

In this section we show how this structure is maintained in $\tilde{O}(m^{4/5})$ time when changing the status of a vertex v. From Lemma 6.7(4), deleting or inserting an inter-level edge in H may cause changing at most 3 higher inter-level edges in the adjacency graph. However, there are at most $\Theta(\log n)$ vertex sets in this structure, so we need other schemes to bound the number of edges updated during a vertex update. Note that after any vertex update, we will run a BFS on the active vertices of V_C and V_D

in $H = G_{k+1}$.

When v is in V_B.

Lemma 6.13. *The degree of any vertex of V_i in H is at most $(i+1)2^{i+2}m^{1/5}$.*

Lemma 6.14. *Changing the status of a vertex v in V_i will not affect the lower-level dynamic spanning forests $F_A, F_0, F_1, ..., F_{i-1}$. It can update at most $O(2^i m^{1/5} i \log^3 n) = \tilde{O}(2^i m^{1/5})$ edges in $F_i, F_{i+1}, ..., F_k, F_{CD}$, respectively. Similarly, changing the status of a vertex v in V_A can update at most $\tilde{O}(m^{1/5})$ edges in F_A, F_0, F_1, $..., F_k, F_{CD}$.*

Proof. Changing the status of a vertex in V_i can only lead to inserting or deleting edges in H associated with a vertex in V_i or higher levels. Thus it will not affect the dynamic spanning forests $F_0, F_1, ..., F_{i-1}$. There are two types of updates:

- Updating v affects a tree $T \in F_i$, so we need to update the list for T in $R(H, F_i, V_{i+1}, ..., V_k, V_C, V_D)$. From Lemma 6.7(3) and 6.13, it results in inserting/deleting $O(i \cdot k2^i m^{1/5})$ edges in $F_{i+1}, ..., F_k, F_{CD}$ or inter-level edges in H.

- From Lemma 6.7(4), updating an inter-level edge e from a spanning tree T' to a higher level vertex in the previous step will change other edges in H. Here we bound the number of such edges. Let $T' \in F_j$, then in $R(H, F_j, V_{j+1}, ..., V_k, V_C, V_D)$, there are at most $O(k)$ inter-level edges induced by T', and from Note 6.5, there is only one spanning tree adjacent to T' in every higher level in H. So the number of inter-level edges changed by e is $O(k^2)$. So we need to update $O(2^i m^{1/5} i \log^3 n)$ such inter-level edges. From Lemma 6.7(4), the total number of edges updated in H is still $\tilde{O}(2^i m^{1/5})$.

□

Thus, the time needed to update the graph H and the dynamic spanning forests $F_A, F_0, ..., F_k$ when switching a vertex in V_i is equal to $\tilde{O}(2^i m^{1/5})|V_i|^{1/2} = \tilde{O}(2^{(1+i)/2} m^{3/5})$. When $i = k = \lfloor 2/5 \log m \rfloor$, the time bound reaches $\tilde{O}(m^{4/5})$.

When $v \in V_B$, if v is the representative vertex of a tree $T \in F_A$ and v is turned inactive, we need to find another active vertex as the representative for T. If v is turned active and there is no active vertices of V_B associated with a tree $T \in F_A$ adjacent to v, v is chosen as the representative vertex of T. In both cases, we need to find all the vertices in V_D adjacent to T and update $\bar{E}_T$, which takes $O(|V_D|) = O(m^{1/5})$ time using $ET(G, F_A, V_D)$ and Theorem 6.8. Since v is adjacent to $O(m^{3/5})$ spanning trees in F_A, this procedure takes $O(m^{4/5})$ time.

When v is in V_A. We follow these steps, which also takes $\tilde{O}(m^{4/5})$ time:

- From Lemma 6.14, changing the status of a vertex v in V_A may update $\tilde{O}(m^{1/5})$ edges in H on $V_B \cup V_C$, and it may update $\tilde{O}(m^{1/5})$ edges in $F_A, F_0, ..., F_k$, F_{CD}, so the time needed for this step is $\tilde{O}(\sqrt{n} m^{1/5}) = \tilde{O}(m^{7/10})$.

- Maintain the structures $ET(G, F_A, V_C)$ and $ET(G, F_A, V_D)$ after updating F_A will take $O(m^{3/5})$ time, because at most $m^{1/5}$ edges will be changed in F_A, and from Theorem 6.8, every link/cut operation in F_A will take $O(|V_C| + |V_D|) = O(m^{2/5})$ time.

- Consider the edges in $\bar{E}_T$ for a tree $T \in F_A$ connecting V_B and V_D. For all the old spanning trees T of F_A, delete $\bar{E}(T)$ from H. For $\bar{E}_{T'}$ on every new spanning tree T' in F_A after cutting or linking, we find a new active representative vertex in V_B and then construct $\bar{E}_{T'}$. Since there can be at most $m^{1/5}$ link/cut operations in F_A, this may change at most $m^{1/5}|V_D| = O(m^{2/5})$ edges in all the edge sets $\bar{E}_T$ and H.

- Consider all other edges in $\bar{E}_T$ for $T \in F_A$. The number of edges changed in $\bar{E}_T$ when performing a cut/link in F_A is $O((|V_C| + |V_D|)|V_D|) = O(m^{3/5})$. So in

fact we need to update $O(m^{4/5})$ such edges in H.

When v is in V_C or V_D Note that the sets $\bar{E}_T$ do not need to update. If $v \in V_C$, update the structures $R(H, F_i, V_{i+1}, ..., V_k, V_C, V_D)$ $(0 \leq i \leq k)$ and $R(G, F_A, V_0, V_1, ..., V_k, V_C)$ takes $\tilde{O}(m^{4/5})$ time since the degree of v is bounded by $m^{4/5}$. If $v \in V_D$, we still need to update $R(H, F_i, V_{i+1}, ..., V_k, V_C, V_D)$ $(0 \leq i \leq k)$, since the size of V_B is bounded by $2m^{4/5}$, from Lemma 6.7(1), this will also take $\tilde{O}(m^{4/5})$ time.

Discussion: Why $\tilde{O}(m^{4/5})$ in the worst-case update time? The $O(n^{1/2})$ worst-case edge update connectivity structure [28] is the main bottleneck for our approach. The set of vertices of degrees in the range $[p, q]$ will contain $\leq 2m/p$ vertices, so the vertex update time will be $\tilde{O}(q(m/p)^{1/2}) \geq \tilde{O}(p^{1/2}m^{1/2})$, if we use the edge update structure. However, when p is large enough, we can run a BFS to get connected components after a update, which takes $\tilde{O}(m^2/p^2)$ time. When balancing these two, the update time will be $\tilde{O}(m^{4/5})$. Also, we can get $\tilde{O}(m^{4/5+\epsilon})$ update time and $\tilde{O}(m^{1/5-\epsilon})$ query time by simply changing the degree bound between V_C and V_D to $O(m^{1/5-\epsilon})$.

6.2.3 Answering a query

To answer a connectivity query between u and v in the subgraph of G induced by the active vertices, first find the spanning trees $T(u)$ and $T(v)$ in $F_A, F_0, ..., F_k, F_{CD}$ containing u and v, respectively. Then find all higher level spanning trees connecting to $T(u)$ or $T(v)$ and check whether $T(u)$ and $T(v)$ are connected to a common spanning tree in higher levels.

By symmetry, we only discuss finding such spanning trees for u. If $u \in V_A$, we first find $T(u) \in F_A$ which contains u, and then find the spanning trees in $F_0, F_1, ..., F_k, F_{CD}$ which is adjacent to the spanning tree $T(u)$ in H. By Note 6.5 and Lemma 6.11, there is only one tree in each forest satisfying this condition. Since we maintain the full active adjacency lists $A_G(T, V_0), ..., A_G(T, V_k), A_G(T, V_C)$

in $R(G, F_A, V_0, V_1, ..., V_k, V_C)$, we can find the trees $T_0, ..., T_k, T_C$ in $F_0, ..., F_k, F_{CD}$ adjacent to $T(u)$ in G in $O(k) = O(\log n)$ time. Those trees are also the ones adjacent to T in H. To find spanning trees in F_{CD} adjacent to T that only contain active vertices in V_D, we need to check whether u' is adjacent to T for all active $u' \in V_D$ by $ET(G, F_A, V_D)$, which takes $O(m^{1/5})$ time.

For any spanning tree $T_i \in F_i$ we have found in V_B or u itself is in a tree T_i of V_B, we recursively run this procedure and find all the trees in $F_{i+1}, ..., F_k, F_{CD}$ connecting to T_i in H, this will take $O(\log n)$ time. Since u can only be connected to one spanning tree in a higher level forest, the time for all T_i will be $O(\log^2 n)$. After this, we check whether there is a common tree in the set of trees connecting to u and v that we found. The running time for the query algorithm is $\tilde{O}(m^{1/5})$.

The correctness of this query algorithm is easy to see from Lemma 6.12. If we find a common tree connecting to u and v, then u, v must be connected. If u, v are connected in the dynamic G, let w be the highest vertex on the path connecting u, v, then u, v will be connected to the tree containing w in the subgraph without higher vertices than w, so we have found such spanning tree in our procedure. A complete proof of the correctness will be given in the full version.

6.3 Dynamic Subgraph Connectivity with $\tilde{O}(m^{2/3})$ Amortized Update Time and Linear Space

In this section, we briefly describe a dynamic subgraph connectivity structure of $\tilde{O}(m^{2/3})$ amortized update time and $\tilde{O}(m^{1/3})$ query time, which improves the structure by Chan, Pătraşcu and Roditty [10] from $\tilde{O}(m^{4/3})$ space to linear space.

Theorem 6.15. *There is a dynamic subgraph connectivity structure for a graph G with $\tilde{O}(m^{2/3})$ amortized vertex update time, $\tilde{O}(m^{1/3})$ query time, $O(m)$ space and $\tilde{O}(m^{4/3})$ preprocessing time.*

As before, define the subsets of vertices in V by their degrees:

- V_L: vertices of degrees at most $m^{2/3}$.
- V_H: vertices of degrees larger than $m^{2/3}$. So $|V_H| < 2m^{1/3}$.

As in [10], we divide the updates into phases, each consisting of $m^{2/3}$ updates. The active vertices in V_L will be divided into two sets P and Q, where P only undergoes deletions and Q accepts both insertions and deletions. At the beginning of each phase, P contains all the active vertices in V_L and Q is empty. So when a vertex of V_L is turned active, we add it to Q. At the end of that phase, we move all the vertices of Q to P and reinitialize the structure. So the size of Q is bounded by $m^{2/3}$. We also define the set $\bar{Q}$ to be all the vertices that have once been in Q within the current phase, so $|Q| \leq |\bar{Q}| \leq m^{2/3}$. Notice that P and Q only contain active vertices but V_H and $\bar{Q}$ may contain both active and inactive vertices. Then we maintain the following structures for each phase:

- Keep a dynamic spanning forest F in the subgraph of G induced by P which supports edge deletions in polylogarithmic amortized time. [59]
- Maintain the active adjacency structure $E_Q = R(G, F, Q)$.
- Maintain the structure $ET(G, F, V_H)$.
- For every edge $e = (u, v)$ where $u \in P$ and $v \in \bar{Q} \cup V_H$ within the current phase, let T be the spanning tree of F containing u. Then for every vertex w in V_H adjacent to T, we add an edge (v, w) into the set E_H. Since $E_H \in (\bar{Q} \cup V_H) \times V_H$, we just need $O(m)$ space to store E_H.
- Construct a dynamic graph G' containing all the active vertices of $Q \cup V_H$, and all the edges in $E \cup E_Q \cup E_H$ connecting two such vertices. So the number of vertices in G' is $O(m^{2/3})$. Maintain a dynamic spanning forest F' of G'

which supports insertions and deletions of edges in polylogarithmic amortized time. [59]

We can see both E_Q and E_H take linear space to store, and from Theorem 6.8 and the dynamic structure for edge updates [59], the total space is still linear. It takes linear time to initialize $F, E_Q, ET(G, F, V_H), G'$ and $\tilde{O}(m^{4/3})$ time to initialize E_H. To see the consistency of this structure, we have the following lemma:

Lemma 6.16. *Two active vertices of $Q \cup V_H$ are connected in G' if and only if they are connected in the subgraph of G induced by the active vertices.*

Proof. If there is an edge in E_Q or E_H connecting u and v, u and v are connected in the subgraph of G induced by the active vertices. So the"only if" direction is obvious.

When two active $u, v \in Q \cup V_H$ are connected through a connected component in P, if $u, v \in Q$ they will be connected by the edges of E_Q by Lemma 6.6, otherwise one of them is in V_H, then there is an edge in E_H connecting u and v directly. Thus when $u, v \in Q \cup V_H$ are connected in G, the path can be divided into parts of the above case and original edges in E, so u and v are still connected in G'. □

When updating a vertex in V_L, we analyze the update time by structures:

Maintaining F and E_Q. When deleting a vertex from P, we may split a spanning tree of F into at most $m^{2/3}$ subtrees. So it takes $\tilde{O}(m^{2/3})$ time to maintain E_Q. When updating a vertex in Q, we need to update $O(m^{2/3})$ edges of E_Q from Lemma 6.7. So it takes $\tilde{O}(m^{2/3})$ time to update F'.

Maintaining E_H. We need to update E_H when a new vertex is inserted to $\bar{Q}$ or when a vertex is deleted from P. When a new vertex is inserted to $\bar{Q}$, we check all the edges associated with it and find the spanning trees in F adjacent to it, then update E_H. When deleting a vertex of P, we find the vertices of V_H adjacent to T which contains that vertex and delete all the outdated edges of E_H. It is hard to bound the time for updating E_H within one update, so we consider the total time needed in one

phase. For every edge $e = (u, v)$ where $u \in P$, when v appears in $\bar{Q}$ or V_H, we add $O(m^{1/3})$ edges (v, w) to E_H where w is in V_H and adjacent to the spanning tree in F containing u. As long as u is still in P, the number of such w in V_H can only decrease since P supports deletion only. So only deletions will take place for the edges in E_H induced by e. Thus, updating E_H and the corresponding F' will take $\tilde{O}(m^{4/3})$ time per phase, so we get $\tilde{O}(m^{2/3})$ amortized time.

Maintaining $ET(G, F, V_H)$. By the same reasoning, maintaining the structure $ET(G, F, V_H)$ for one vertex in V_H within one phase will take $\tilde{O}(m)$ time.

Updating a vertex in V_H. We only need to maintain the graph G' when updating a vertex in V_H, which will take $\tilde{O}(m^{2/3})$ time.

Answering a query of connectivity between u and v. If both are in $Q \cup V_H$, by Lemma 6.16, we check whether they are connected in G'. Otherwise suppose $u \in P$ (or v), we need to find an active vertex u' (or v') in $Q \cup V_H$ which is adjacent to the spanning tree $T \in F$ containing u (v). Similarly to the worst-case structure, we need to check all the active vertices in V_H whether they are adjacent to T by $ET(G, F, V_H)$, which takes $O(m^{1/3})$ time. Thus when only $u \in P$, u, v are connected iff u' and v are connected in G' since the path must go through a vertex in G'. When both of them are in P, they are connected in G iff they are in the same tree of F or u' and v' are connected in G'.

CHAPTER VII

All-Pair Bounded-Leg Shortest Paths

In this chapter, we consider the all-pair bounded-leg shortest paths problem. In a weighted, directed graph an L-bounded leg path is one whose constituent edges have length at most L. For any *fixed* L, computing L-bounded leg shortest paths is just as easy as the standard shortest path algorithm. We give an algorithm for preprocessing a directed graph in order to answer *approximate* bounded leg distance and bounded leg shortest path queries. In particular, we can preprocess any graph in $O(n^3\epsilon^{-1}\log^3 n)$ time, producing a data structure with size $O(n^2\epsilon^{-1}\log n)$ that answers $(1+\epsilon)$-approximate L-bounded leg distance queries in $O(\log\log n)$ time for any pair of vertices u, v and leg bound L. If the corresponding $(1+\epsilon)$-approximate shortest path has l edges, it can be returned in $O(l\log\log n)$ time. [1]These bounds are all within polylog(n) factors of the best standard all-pairs shortest path algorithm and improve substantially the previous best bounded-leg shortest path algorithm, whose preprocessing time and space are $O(n^4)$ and $\tilde{O}(n^{2.5})$. [53]

7.1 The notations

Let $G = (V, E)$ be a directed graph with a length function $w : E \to \mathbb{R}^+$. Our aim is to construct a table such that for every ordered pair of vertices (u, v) in V and any

[1]This result appears in Duan and Pettie's paper "Bounded-leg Distance and Reachability Oracles" [21] in SODA 2008.

positive real number $L \in \mathbb{R}^+$, we can obtain a $(1+\epsilon)$-approximate L-bounded leg distance from that table. Denote the L-bounded leg distance between $u, v \in V$ by $\delta^L(u,v)$. We say y is a $(1+\epsilon)$-approximation of x when $x \leq y \leq (1+\epsilon)x$.

Let $E_0 = (e_1, e_2, ..., e_m)$ be the list of edges in increasing order. Let $G_i = (V, E^{[1,i]})$, where $E^{[x,y]} = \{e_x, e_{x+1}, ..., e_y\}$, and abbreviate $\delta_{G_i}(u,v)$ by $\delta_i(u,v)$.

In this chapter, v is reachable from u in a graph G is represented by $u \xrightarrow{G} v$, and v is not reachable from u in G by $u \overset{G}{\nrightarrow} v$. The *bottleneck distance* from u to v is defined by

$$L(u,v) = \min\{w(e_i) | u \xrightarrow{G_i} v\}$$

If $u \overset{G}{\nrightarrow} v$, then define $L(u,v) = \infty$.

As the leg bound L increases the set of usable edges grows. Therefore, the length of the shortest path from u to v in this insert-only dynamic subgraph can only decrease. When $L \geq w(e_m)$, the subgraph becomes the entire graph G. We can see that all edges in the path from u to v under leg bound $L(u,v)$ are no longer than $L(u,v)$, so $\delta^{L(u,v)}(u,v) \leq (n-1)L(u,v)$. Any path from u to v in G must contain an edge no shorter than $L(u,v)$, so $\delta_G(u,v) \geq L(u,v)$. Thus, we only need $\log_{1+\epsilon}(n-1)$ different distances for each pair of vertices to be able to return a $(1+\epsilon)$-approximate distance under any leg bound. The main problem is how to construct this set of distances efficiently. An obvious solution is to insert one edge at a time, then check in $O(1)$ time for every pair of vertices whether its distance changes. The total time for this trivial algorithm is $O(mn^2)$. We will use a natural divide-and-conquer method to reduce the running time to $O(\frac{1}{\epsilon}n^3 \log^3 n)$.

Our aim is to construct, for every pair of vertices (u,v), a set of bounded-leg distance entries: $D(u,v) = \{(L_1, d^{L_1}(u,v)), (L_2, d^{L_2}(u,v)), ..., (L_k, d^{L_k}(u,v))\}$, where $L(u,v) = L_1 < L_2 < ... < L_k = w(e_m)$ and $d^{L_i}(u,v)$ is an *approximation* of the distance from u to v under leg bound L_i. For any leg bound L, the distance between u and v should be $(1+\epsilon)$-approximated by some $d^{L_i}(u,v) \in D(u,v)$ where L_i is the

maximum among those $L_i \leq L$. Denote this by $d^L(u,v) = d^{L_i}(u,v)$. If $L < L_1$, $d^L(u,v) = \infty$. Moreover, for every (u,v), we guarantee that $|D(u,v)| \leq 2\log_{1+\epsilon} n$, then $|D(u,v)| = O(\log_{1+\epsilon} n)$, so for every distance query under leg bound L, we can find $d^L(u,v)$ in $O(\log\log_{1+\epsilon} n)$ time.

Same structure for exact BLSP?

Since there can be $\Theta(n^2)$ different edges in the graph, the distance between any pair of vertices can change at most $O(n^2)$ times, that is, at most $O(n^4)$ different bounded-leg distances needed if our data structure must store every distance that it could return (i.e. no addition).

However, it is not clear whether such a graph exists. In fact, if there is a graph H in which there exists a pair of vertices (u,v) having $\Theta(n^2)$ different bounded-leg distances, then we can add $2n$ vertices in H: $\{u_1, u_2, \ldots, u_n\}$ and $\{v_1, v_2, \ldots, v_n\}$, and also directed edges with very short lengths $\{(u_1,u),(u_2,u),\ldots,(u_n,u)\}$ and $\{(v,v_1),(v,v_2),\ldots,(v,v_n)\}$. Then in this extended graph H', there are $3n$ vertices, and for any pair of u_i and v_j, their distance varies $\Theta(n^2)$ times when leg bound increases, so in total there are $\Theta(n^4)$ different bounded-leg distances.

Now consider the following directed graph $H = (V,E)$: $V = \{u = a_1, a_2, \ldots, a_k = b_0, b_1, b_2, \ldots, b_k, v\}$, and: $E = \{(a_i, a_{i+1}) | 1 \leq i \leq k-1, w(a_i, a_{i+1}) = 4k\} \cup \{(b_i, v) | 0 \leq i \leq k, w(b_i, v) = 2k+1-2i\} \cup \{(a_i, b_j) | 1 \leq i \leq k-1, 1 \leq j \leq k, w(a_i, b_j) = k^2 - ik + 3k + j\}$. It is a good exercise to show that the distance from u to v varies $\Theta(k^2)$ times, thus there exist graphs with $\Theta(n^4)$ different bounded-leg distances. This implies that to improve the $\Theta(n^4)$ exact BLSP oracles or $\Theta(\epsilon^{-1}n^2\log n)$ $(1+\epsilon)$-approximate BLSP orables, the query algorithms must add or subtract numbers to calculate an answer.

```
Modified-Floyd(d, P)
  d: an n × n matrix
  P: a set of vertex pairs

  for k = 1 to n do
    for all (s, t) in P do
      d[s, t] ← min{d[s, t], d[s, v_k] + d[v_k, t]}
  return d
```

Figure 7.1: Modified Floyd Algorithm: As inputs, d is a matrix that contains the approximate distances for all pairs except the pairs in P. The algorithm returns the approximate distance matrix d.

7.2 A Binary Partition Algorithm

The high-level idea of our algorithm is to find a small set of distances ($O(\log_{1+\epsilon} n)$ per vertex pair) that can $(1+\epsilon)$-approximate any L-bounded leg distance. Suppose that we have just found a reasonably accurate estimate to the distances in G_i and G_j respectively, $i < j$. Call these estimates d_i and d_j. If $d_i(u,v)/d_j(u,v)$ is sufficiently close to 1 then $d_i(u,v)$ can be considered a good-enough estimate of $\delta_{i'}(u,v)$, for all $i < i' < j$. Thus, we can focus on vertex pairs, call them P, whose distance drops significantly between G_i and G_j. Our idea is to compute a reasonably good estimate of the distances of the median $G_{(i+j)/2}$ using a version of the Floyd-Warshall algorithm (Figure 1) that just considers the pairs P. The correctness and time complexity of our algorithm will follow from two lemmas. The first says, essentially, that if the *Modified-Floyd* algorithm starts off with a good approximation to the distances on all vertex pairs besides P, it ends with a good approximation for all vertex pairs, including P. One problem in our divide-and-conquer approach is that errors accumulate as we break the problem into smaller pieces. The second lemma bounds the growth of these errors.

Lemma 7.1. *Let $G' = (V', E')$ be a graph, let $P \subseteq V' \times V'$ be a set of pairs of vertices. If initially for all $(s,t) \in (V' \times V') \setminus P$, $d(s,t)$ is an α-approximation of $\delta(s,t)$, and for all $(s,t) \in P \cap E'$, $\delta(s,t) \leq d(s,t) \leq w(s,t)$, then the matrix d*

returned by this Modified Floyd procedure satisfies: for any pair $(s,t) \in P$, $d(s,t)$ is an α-approximation of $\delta(s,t)$.

Proof. Notice that this algorithm can never underestimate a distance $\delta(s,t)$ if there are no underestimates originally. Denote the real shortest path from s to t in G' by $s \rightharpoonup t$. For any $(s,t) \in P$, if the shortest path $s \rightharpoonup t$ is composed of only one edge, then $(s,t) \in E'$ and $\delta(s,t) = w(s,t) = d(s,t)$, so this case is trivial. Now assume that after k rounds ($k \geq 1$), for every pair of vertices $(s,t) \in P$ such that $s \rightharpoonup t$ includes only intermediate vertices from $\{v_1, \ldots, v_k\}$, $d(s,t)$ is an α-approximation of $\delta(s,t)$. In the $(k+1)$th round, if $k+1$ is the index of the highest intermediate vertex in $s \rightharpoonup t$, for $(s,t) \in P$, then the highest indices in the paths $s \rightharpoonup v_{k+1}$ and $v_{k+1} \rightharpoonup t$ are both at most k. So, by the inductive hypothesis, $d(s, v_{k+1})$ and $d(v_{k+1}, t)$ are already α-approximations of $\delta(s, v_{k+1})$ and $\delta(v_{k+1}, t)$ respectively. Therefore, after the $(k+1)$st round, $d(s,t) \leq d(s, v_{k+1}) + d(v_{k+1}, t) \leq \alpha\delta(s, v_{k+1}) + \alpha\delta(v_{k+1}, t) = \alpha\delta(s,t)$, so $d(s,t)$ is also an α-approximation of $\delta(s,t)$. □

Suppose that we have a pretty good approximation to the distances in G_i and G_j. We want to find an approximation to the distances in G_q, where $q = \lfloor (i+j)/2 \rfloor$. If the distances of some pairs change slightly between G_i and G_j, then we can just use their distances in G_i to estimate their distance in G_q. We can focus our attention on the pairs whose distance changes a lot between G_i and G_j.

Lemma 7.2. *Let d_i and d_j be α^l-approximations of δ_i and δ_j, where $i < j$. Then we can find an α^{l+1}-approximation of δ_q, where $q = \lfloor (i+j)/2 \rfloor$, in $O(n|P| + j - i)$ time, where $P = \{(s,t) \mid (s,t) \in V \times V$ and $\frac{d_i(s,t)}{d_j(s,t)} > \alpha\}$.*

Proof. By definition: for all $(s,t) \in V \times V$, we have $\delta_i(s,t) \leq d_i(s,t) \leq \alpha^l \delta_i(s,t)$ and $\delta_j(s,t) \leq d_j(s,t) \leq \alpha^l \delta_j(s,t)$. Because for all $(s,t) \notin P$, $d_i(s,t) \leq \alpha d_j(s,t)$, it follows that

$$\delta_i(s,t) \leq d_i(s,t) \leq \alpha d_j(s,t) \leq \alpha^{l+1} \delta_j(s,t)$$

Since the bounded-leg distance can only decrease with a larger leg-bound, for all $i \le q \le j, \delta_j(s,t) \le \delta_q(s,t) \le \delta_i(s,t)$. Therefore

$$\delta_q(s,t) \le \delta_i(s,t) \le d_i(s,t) \le \alpha^{l+1}\delta_j(s,t) \le \alpha^{l+1}\delta_q(s,t)$$

Thus $d_i(s,t)$ is an α^{l+1}-approximation of $\delta_q(s,t)$ for any $(s,t) \in (V \times V) \setminus P$.

We can add the edge set $E^{[i+1,q]} = \{e_{i+1}, e_{i+2}, ..., e_q\}$ into d_i, that is, for all $(s,t) \in E^{[i+1,q]}$, if $(s,t) \in P$, set $d_i(s,t) = \min\{d_i(s,t), w(s,t)\}$. This takes $q - i = O(j - i)$ time. We can ignore $E^{[1,i]}$ because for all $(s,t) \in E^{[1,i]}$, $\delta_i(s,t) \le w(s,t) \le w(e_i)$. If $\delta_j(s,t) < \delta_i(s,t)$ then $\delta_j(s,t) \ge w(e_i) \ge \delta_i(s,t)$, which is a contradiction. Thus, if $(s,t) \in E^{[1,i]}$ then $(s,t) \notin P$. Now for all $(s,t) \in P \cap E^{[1,q]} = P \cap E^{[i+1,q]}$, $\delta_q(s,t) \le \delta_i(s,t) \le d_i(s,t) \le w(s,t)$. From lemma 2.1, if we take d_i and P as the input of the *Modified Floyd* procedure, in $O(n|P|)$ time we can find an α^{l+1}-approximation of δ_q; call it d_q. □

Corollary 7.3. *Let $k = \frac{m}{2^l}$. If we already have an α^l-approximation $d_{\lfloor i \cdot k \rfloor}$ of $\delta_{\lfloor i \cdot k \rfloor}$ for all $0 \le i \le 2^l$, then we can find an α^{l+1}-approximation $d_{\lfloor i \cdot \frac{k}{2} \rfloor}$ of $\delta_{\lfloor i \cdot \frac{k}{2} \rfloor}$ for all $0 \le i \le 2^{l+1}$ in $O(n^3 \log_\alpha n)$ time.*

Proof. Apply lemma 2.2 to all pairs of adjacent graphs $G_{\lfloor i \cdot k \rfloor}$ and $G_{\lfloor (i+1) \cdot k \rfloor}$ $(0 \le i < 2^l)$, and let P_i be the set of pairs P for them. Since $\delta^{L(u,v)}(u,v) \le (n-1)\delta_G(u,v)$, the number of times (u,v) can appear in the sets P_i is $O(\log_\alpha n)$. Thus, the total time taken by this procedure is $O(n \cdot \sum_{i=0}^{2^l-1} |P_i| + m) = O(n^3 \log_\alpha n)$.

□

Now we can apply Corollary 2.1 repeatedly and obtain the main algorithm.

Theorem 7.4. *For any graph G of n vertices and m edges, we can construct the set $D(u,v)$, for every pair of vertices (u,v), that contains a $(1+\epsilon)$-approximation of $\delta_q(u,v)$ for any $0 < q \le m$, in $O(\epsilon^{-1} n^3 \log^3 n)$ time.*

Proof. First, set $d_0(u,v) = +\infty$ for all (u,v), and utilize the original Floyd-Warshall algorithm to compute $d_m(u,v) = \delta_G(u,v)$ for all pairs (u,v) in $O(n^3)$ time.

Then set $\alpha = (1+\epsilon)^{\frac{1}{\log m}}$, and run the procedure of Corollary 2.3 for $l = 0, 1, ...,$ $\log_2 m - 1$. Finally we can get an $(\alpha^{\log_2 m} = 1+\epsilon)$-approximation of all bounded-leg distances for δ_q where $0 < q \leq m$. Thus the total time of this algorithm is $O(n^3 \log n \log_\alpha n) = O(\epsilon^{-1} n^3 \log^3 n)$.

Every time we finish a run of the *Modified Floyd* algorithm in graph G_q, we insert $(w(e_q), d_q(u,v))$ into $D(u,v)$ for every pair (u,v) in P. This will take time $O(|P| \log\log_{1+\epsilon} n)$, which is much less than the *Modified Floyd* algorithm itself. Finally we can see that in $D(u,v)$, the two entries $(L_i, d^{L_i}(u,v))$ and $(L_{i+2}, d^{L_{i+2}}(u,v))$ must satisfy $\frac{d^{L_i}(u,v)}{d^{L_{i+2}}(u,v)} > 1+\epsilon$ otherwise the intermediate entry $(L_{i+1}, d^{L_{i+1}}(u,v))$ would not be computed in this algorithm. So the size of $D(u,v)$ is bounded by $2\epsilon^{-1} \log n$. □

We can see that the space complexity for every execution of the procedure is $O(n^2)$, and the depth of the recursion is $O(\log n)$. So the total space complexity is $O((1+\epsilon^{-1})n^2 \log n)$.

7.3 Answer a bounded-leg shortest path query

In addition to answering approximate bounded-leg distance queries, we also want to find a path of that distance satisfying the leg bound. Answering path queries is what made the space bound of Roditty-Segal's algorithm [53] $O(n^{2.5})$. So, given a pair of vertices (u,v) and a leg bound L, we want to find a path γ such that $\forall e \in \gamma$, $w(e) \leq L$ and $\sum_{e\in\gamma} w(e) = d^L(u,v)$, where $d^L(u,v)$ is the $(1+\epsilon)$-approximation we obtained from the structure $D(u,v)$.

It is easy to achieve this since all our distances are obtained from the *Modified Floyd* algorithm. We can save the intermediate vertex in every step of Floyd algorithm, then recursively find the two subpaths. We will slightly change our structure

GetPath(u, v, L)
Find $(L', d^{L'}(u,v), \pi^{L'}(u,v)) \in D(u,v)$ such that L' is the largest satisfying $L' \leq L$
If $\pi^{L'}(u,v) = nil$, return the edge (u,v).
Else Let $w = \pi^{L'}(u,v)$.
GetPath(u, w, L')
GetPath(w, v, L')

Figure 7.2: Algorithm for Finding Paths

and algorithm. For any pair (u,v), any entry $(L_i, d^{L_i}(u,v)) \in D(u,v)$, we define a function $\pi^{L_i}(u,v) \in V$ to be the vertex with the highest index in the real path from u to v of distance $d^{L_i}(u,v)$ under leg bound L_i; if the path only consists of one edge, then $\pi^{L_i}(u,v) = nil$. So, in the third line of the algorithm in Figure 1:

$$d(s,t) \leftarrow \min(d(s,t), d(s,v_k) + d(v_k,t))$$

If $d(s,t)$ does not change after executing this line, then $\pi(s,t)$ also does not change. If $d(s,t) = d(s,v_k) + d(v_k,t)$, then $\pi(s,t)$ will be set to v_k. After this procedure, we can add the entry $(L, d^L(u,v), \pi^L(u,v))$ to $D(u,v)$.

The procedure to find the path is shown in Figure 2:

Since the leg bound L can only decrease in the recursion, this recursive procedure will output the approximate bounded-leg shortest path from u to v in $O(\log(\epsilon^{-1} \log n))$ time per edge.

7.4 A one-level algorithm for all-pair bounded-leg distance

In section 2.2 we execute a divide-and-conquer algorithm with a binary partition. Of course, it is an efficient algorithm for the graph of $\Theta(n^2)$ edges because there is no truly sub-cubic all-pair shortest path algorithm for only the whole graph [9]. However, when it is performed on a sparse graph, the time for this algorithm cannot be reduced since the time taken by the *Modified-Floyd* algorithm does not depend on

the number of edges. To get rid of this, we cannot use the modified Floyd algorithm, but only execute a one-level partition. First, we need the following lemma:

Lemma 7.5. *For the two subgraphs of G: G_i and G_j $(i < j-1)$, if we already have the α^l-approximations of δ_i and δ_j: d_i and d_j, then we can find the α^{l+1}-approximation of $\delta_q(u,v)$ for all $(u,v) \in V \times V$ and $i < q < j$ in $O((j-i)|P|)$ time, where $P = \{(u,v)|(u,v) \in V \times V$ and $\frac{d_i(u,v)}{d_j(u,v)} > \alpha\}$.*

Proof. From the proof of lemma 2.2, we can see that $d_i(u,v)$ is already an α^{l+1}-approximation of $\delta_q(u,v)$ for any $(u,v) \notin P$. Then we insert the edges $e_{i+1}, e_{i+2}, ..., e_{j-1}$ in the increasing order of their indices into the graph G_i. When inserting edge $e_k = (s,t)$, we check for every $(u,v) \in P$ that:

$$d_k(u,v) \leftarrow \min\{d_{k-1}(u,v), d_{k-1}(u,s) + w(s,t) + d_{k-1}(t,v)\}$$

where $d_{k-1}(u,v)$ is the same as $d_i(u,v)$ if $(u,v) \notin P$. Since

$$\delta_k(u,v) = \min\{\delta_{k-1}(u,v), \delta_{k-1}(u,s) + w(s,t) + \delta_{k-1}(t,v)\}$$

We can conclude that if d_{k-1} is an α^{l+1}-approximation of δ_{k-1}, then d_k is an α^{l+1}-approximation of δ_k. By induction, the lemma holds. It is obvious that the running time for this procedure is $O((j-i)|P|)$. □

Using the $O(mn+n^2 \log\log n)$ time APSP algorithm [50], we can compute the all-pair shortest path for the graphs $G_0, G_{\lfloor m/k \rfloor}, G_{\lfloor 2m/k \rfloor}, ..., G_m$ for some k, then apply Lemma 2.3 to obtain the $(1+\epsilon)$-approximation for all pairs of vertices in any G_q $(0 < q \leq m)$. So, the time needed is $O(kmn + kn^2 \log\log n) + O(\frac{m}{k} \cdot n^2 \log_{1+\epsilon} n)$. If $m \geq n \log\log n$, for $k = \sqrt{n \log_{1+\epsilon} n}$, the running time is $O(mn^{3/2}\sqrt{\log_{1+\epsilon} n})$, and if $m < n \log\log n$, for $k = \sqrt{\frac{m \log_{1+\epsilon} n}{\log\log n}}$, the running time is $O(n^2\sqrt{m \log_{1+\epsilon} n \log\log n})$. They are fast than binary partition algorithm described in Section 2.2 when m is less

than $n^{3/2} \log^2 n \sqrt{\log_{1+\epsilon} n}$.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and analysis of computer algorithms*. Addison-Wesley, Reading, MA, 1975.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings 41st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

[3] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proceedings 4th Latin American Symp. on Theoretical Informatics (LATIN), LNCS Vol. 1776*, pages 88–94, 2000.

[4] M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004.

[5] A. Bernstein and D. Karger. Improved distance sensitivity oracles via random sampling. In *Proceedings 19th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 34–43, 2008.

[6] A. Bernstein and D. Karger. A nearly optimal oracle for avoiding failed vertices and edges. In *Proceedings 41st Annual ACM Symposium on Theory of Computing (STOC)*, pages 101–110, 2009.

[7] P. Bose, A. Meheswari, G. Narasimhan, M. Smid, and N. Zeh. Approximating geometric bottleneck shortest paths. *Computational Geometry: Theory and Applications*, 29:233–249, 2004.

[8] T. Chan. Dynamic subgraph connectivity with geometric applications. *SIAM J. Comput.*, 36(3):681–694, 2006.

[9] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. 39th ACM Symposium on Theory of Computing (STOC)*, pages 590–598, 2007.

[10] T. M. Chan, M. Pătraşcu, and L. Roditty. Dynamic connectivity: Connecting to networks and geometry. In *Proceedings 49th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 95–104, 2008.

[11] D. Coppersmith. Rectangular matrix multiplication revisited. *J. Complex.*, 13(1):42–49, 1997.

[12] D. Coppersmith and T. Winograd. Matrix multiplication via arithmetic progressions. In *Proc. 19th ACM Symp. on the Theory of Computing (STOC)*, pages 1–6, 1987.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[14] A. Czumaj, M. Kowaluk, and A. Lingas. Faster algorithms for finding lowest common ancestors in directed acyclic graphs. *Theoretical Computer Science*, 380(1–2):37–46, 2007.

[15] C. Demetrescu and G. F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, 2008.

[16] C. Demetrescu, M. Thorup, R. A. Chowdhury, and V. Ramachandran. Oracles for distances avoiding a failed node or link. *SIAM J. Comput.*, 37(5):1299–1318, 2008.

[17] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] D. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Info. Proc. Lett.*, 85:211–213, 2003.

[19] R. Duan and S. Pettie. Dual-failure distance and connectivity oracles. In *Proceedings 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 506–515, 2009.

[20] Ran Duan. New data structures for subgraph connectivity. In *ICALP '10: 37th International Colloquium on Automata, Languages and Programming*, pages 201–212. Springer, 2010.

[21] Ran Duan and Seth Pettie. Bounded-leg distance and reachability oracles. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 436–445, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[22] Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *SODA '09: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 506–515, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[23] Ran Duan and Seth Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *SODA '09: Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 384–391, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[24] Ran Duan and Seth Pettie. Approximating maximum weight matching in near-linear time. In *Proceedings 51st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 673–682, 2010.

[25] Ran Duan and Seth Pettie. Connectivity oracles for failure prone graphs. In *STOC '10: Proceedings of the 42nd ACM symposium on Theory of computing*, pages 465–474, New York, NY, USA, 2010. ACM.

[26] J. Edmonds. Maximum matching and a polyhedron with $0, 1$-vertices. *J. Res. Nat. Bur. Standards Sect. B*, 69B:125–130, 1965.

[27] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[28] D. Eppstein, Z. Galil, G. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

[29] G. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[30] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[31] M. L. Fredman and D. E. Willard. Surpassing the information-theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[32] D Frigioni and G. F. Italiano. Dynamically switching vertices in planar graphs. *Algorithmica*, 28(1):76–103, 2000.

[33] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 434–443, 1990.

[34] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for network problems. *SIAM J. Comput.*, 18(5):1013–1036, 1989.

[35] H. N. Gabow and R. E. Tarjan. Faster scaling algorithms for general graph-matching problems. *J. ACM*, 38(4):815–853, 1991.

[36] Harold N. Gabow. An efficient implementation of edmonds' algorithm for maximum matching on graphs. *J. ACM*, 23:221–234, April 1976.

[37] M. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, 1999.

[38] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48(4):723–760, 2001.

[39] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.

[40] X. Huang and V. Pan. Fast rectangular matrix multiplication and applications. *Journal of Complexity*, 14:257–299, 1998.

[41] T. Kameda and J. I. Munro. A $o(|v||e|)$ algorithm for maximum matching of graphs. *Computing*, 12(1):91–98, 1974.

[42] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[43] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart & Winston, New York, 1976.

[44] Z. Lotker, B. Patt-Shamir, and S. Pettie. Improved distributed approximate matching. In *Proceedings 20th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2008.

[45] J. Matoušek. Computing dominances in e^n. *Info. Proc. Lett.*, 38(5):277–278, 1991.

[46] Julián Mestre. Greedy in approximation algorithms. In *Proceedings of the 14th conference on Annual European Symposium - Volume 14*, pages 528–539, London, UK, 2006. Springer-Verlag.

[47] S. Micali and V. Vazirani. An $O(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 17–27, 1980.

[48] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings 38th ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[49] M. Pătraşcu and M. Thorup. Planning for fast connectivity updates. In *Proceedings 48th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 263–271, 2007.

[50] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1):47–74, 2004.

[51] S. Pettie and P. Sanders. A simpler linear time $2/3 - \epsilon$ approximation to maximum weight matching. *Info. Proc. Lett.*, 91(6):271–276, 2004.

[52] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Proc. 16th Symp. on Theoretical Aspects of Computer Science (STACS), LNCS 1563*, pages 259–269, 1999.

[53] L. Roditty and M. Segal. On bounded leg shortest paths problems. In *Proceedings 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 775–784, 2007.

[54] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings 36th ACM Symposium on Theory of Computing (STOC)*, pages 184–191, 2004.

[55] P. Sankowski. Faster dynamic matchings and vertex connectivity. In *Proceedings 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 118–126, 2007.

[56] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[57] A. Shapira, R. Yuster, and U. Zwick. All-pairs bottleneck paths in vertex weighted graphs. In *SODA*, pages 978–985, 2007.

[58] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. 40th IEEE Symp. on Foundations of Computer Science (FOCS)*, pages 605–614, 1999.

[59] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings 32nd ACM Symposium on Theory of Computing (STOC)*, pages 343–350, 2000.

[60] M. Thorup. Worst-case update times for fully-dynamic all-pairs shortest paths. In *Proceedings 37th ACM Symposium on Theory of Computing (STOC)*, pages 112–119, 2005.

[61] M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007.

[62] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *Proceedings 39th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 75–84, 1975.

[63] V. Vassilevska. *Efficient Algorithms for Path Problems in Weighted Graphs*. PhD thesis, Carnegie Mellon University, August 2008.

[64] V. Vassilevska. Personal communication. 2008.

[65] V. Vassilevska, R. Williams, and R. Yuster. All-pairs bottleneck paths for general graphs in truly sub-cubic time. In *STOC*, pages 585–589, 2007.

[66] V. V. Vazirani. A theory of alternating paths and blossoms for proving correctness of the $O(\sqrt{V}E)$ general graph maximum matching algorithm. *Combinatorica*, 14(1):71–109, 1994.

[67] D. E. D. Vinkemeier and S. Hougardy. A linear-time approximation algorithm for weighted matchings in graphs. *ACM Trans. on Algorithms*, 1(1):107–122, 2005.

[68] Raphael Yuster. Efficient algorithms on sets of permutations, dominance, and real-weighted apsp. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 950–957, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

[69] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49(3):289–317, 2002.