

Shortest Paths in Directed Graphs

Theory and Practice of a New
Hierarchy-Based Algorithm

Master's Thesis by
Karsten Strandgaard Jørgensen

May 2004

Department of Computer Science, University of Århus,
Åbogade, 8200 Århus N, Denmark

Abstract

In this thesis, we give an introduction to the component hierarchy approach to the shortest paths problem, and present a recent hierarchy-based algorithm by Pettie. The algorithm runs on directed, real-weighted graphs, solving the all-pairs shortest paths problem in $O(mn + n^2 \log \log n)$ time, where m is the number of edges and n the number of vertices. Pettie's algorithm is currently the fastest for sparse graphs, improving the previously best $O(mn + n^2 \log n)$ -time achieved by repeating the Fibonacci-heap version of Dijkstra's classic shortest path algorithm n times.

We explain the concepts underlying the hierarchy approach, then examine the Pettie algorithm in detail and prove its running time and correctness. We also discuss implementation issues of both the Dijkstra and Pettie algorithms, and explore the practicality of Pettie by comparing it to Dijkstra in empirical tests.

"The path of least resistance is the path of the loser."
– H. G. Wells

Contents

1	Introduction	6
1.1	Applications	6
1.2	History	6
1.3	Organization	7
2	Theory	9
2.1	Definitions	9
2.2	101 Ways to Compute Shortest Paths	10
2.2.1	Notes on Shortest Paths Algorithms	11
2.2.2	SSSP	12
2.2.3	APSP	14
2.2.4	Where Are We?	15
2.3	High-level Generic Algorithm	16
2.4	Dijkstra's Algorithm	17
2.5	The Component Hierarchy Approach	18
2.5.1	Sorting Is Unnecessary	19
2.5.2	The Underlying Idea	20
2.5.3	Handling Independent Subproblems	21
2.5.4	Using the Hierarchy	22
2.6	Pettie's Algorithm	23
2.6.1	Building the Component Hierarchy	23
2.6.2	Computing Shortest Paths	29
2.6.3	Properties of the Hierarchy	33
2.7	Pettie Running Time	35
2.7.1	Building the Hierarchy	35
2.7.2	The Visit Calls	36
2.7.3	Total Running Time	38
2.8	Pettie Subroutines and Data Structures	39
2.8.1	Finding Strongly Connected Components	39

2.8.2	Handling D-values: Gabow Decrease-Split-Findmin . . .	41
2.8.3	The Bucket Structure	45
2.8.4	The Oracle Function	53
2.9	Pettie Correctness	58
2.9.1	Correctness of the Oracle	59
2.9.2	Correctness of VISIT	62
3	Practice	70
3.1	Common Factors	70
3.1.1	Graph Representation	70
3.1.2	Programming Language	71
3.1.3	Weight Data Type	71
3.2	The Dijkstra Algorithm	72
3.3	The Pettie Algorithm	73
3.3.1	Building the component hierarchy	73
3.3.2	The Oracle Function	74
3.3.3	The VISIT Procedure	74
3.3.4	The Bucket Structure	75
3.3.5	Dijkstra on Subproblems	78
3.3.6	Handling D -values	78
3.3.7	Experimental Setup	79
3.3.8	Timing	80
3.3.9	Measurements	81
3.3.10	Test Data	81
3.4	Results	84
3.4.1	The Dijkstra Priority Queue	84
3.4.2	Building the Component Hierarchy	86
3.4.3	The Oracle Function	86
3.4.4	The Bucket Structure	87
3.4.5	Dijkstra on Subproblems	91
3.4.6	Dijkstra vs. Pettie	92
3.5	Conclusions and Future Work	94
A	Operations in Our Computational Model	100
B	NP-hard Shortest Paths Problems	102
C	Ackermann's Function	104

Chapter 1

Introduction

This is a master's thesis. The topic is computation of shortest paths. Let's get started.

1.1 Applications

Computing the shortest paths in a graph has many different applications. Some are quite obvious, such as finding the shortest car route from your home address to the only place in the world where they sell your favorite kind of apples or aluminum. Or, if you need to find a clever way to route data packets through a complex computer network, finding shortest paths may be for you.

A perhaps less obvious application is in the field of speech recognition. Here, one can build graphs of associations between words and use shortest paths to determine the most likely meaning of a spoken sentence [27]. Shortest paths can also play a part in solving problems such as gene recognition [5], image segmentation [29], as well as in other areas too numerous to mention here.

1.2 History

The notion of finding shortest paths in graphs has been around for many years. The first efficient algorithm was formulated in the late 1950's by

E. W. Dijkstra [9], who also claims to have invented the problem(!). Since then, several new ways of finding shortest paths have been devised in the pursuit of more effective algorithms. Over the years, shortest path algorithms have become increasingly specialized. Different algorithms for different types of graphs and different models of computation. This means that it is important to clearly define one's corner of the world when dealing with shortest paths.

For the most part of this thesis, our corner of the world will be *all-pairs shortest paths in directed, non-negative, real-weighted graphs*. In this corner, real algorithmic innovations have been few and far between. For many years, the best algorithms were still based on Dijkstra's – only the implementation of the underlying priority queue changed. This improved the running time from the original $O(n^3)$ to the current $O(mn + n^2 \log n)$, where n is the number of vertices and m is the number of edges in the input graph.

Then, in 1999, Mikkel Thorup proposed a rather different way of approaching the problem. Using a *component hierarchy*, Thorup described an $O(mn)$ time algorithm [38] for undirected, integer-weighted graphs. Soon thereafter, Torben Hagerup [20] generalized Thorup's algorithm to work on directed graphs in time $O(mn + n^2 \log \log n)$. Finally, in 2003, Seth Pettie applied the component hierarchy approach to real-weighted graphs, producing an $O(mn + n^2 \log \log n)$ algorithm [31] – asymptotically faster than any Dijkstra variant.

The main focus of this thesis will be on the component hierarchy approach in general and its concrete manifestation in the Pettie algorithm. We will examine this algorithm in detail, prove its running time and correctness, discuss implementation details, and test its use in practice.

1.3 Organization

This thesis is logically divided into two chapters: theory and practice. The theory chapter covers an overview of the current state of affairs in the area of shortest path algorithms, a general introduction to hierarchy-based algorithms, and a thorough breakdown of the Pettie algorithm which is at the center of this thesis. The practice chapter discusses a number of

implementation issues of the Dijkstra and Pettie algorithms and empirically compares the new algorithm to the tried and true Dijkstra.

Chapter 2

Theory

In this chapter, we will give a brief overview of the current landscape of shortest paths algorithms. Then, starting from a generic high-level algorithm, we will show the similarities and differences between the Dijkstra and Pettie algorithms. The latter will be covered extensively, first by introducing the concept of a component hierarchy, and then by explaining the many bits and pieces that make up the algorithm. Along the way we prove its running time and finish with a formal proof of its correctness.

But before we dive into the algorithms, let us get some definitions straight to make it clear what we are dealing with.

2.1 Definitions

First, we define what we mean by a graph.

Graph: A non-negative, real-weighted, directed graph G is a 3-tuple: $G = (V, E, l)$, where V is a set of n vertices, E is a set of m edges (ordered pairs of vertices), and $l : E \mapsto \mathbb{R}_+$ assigns weights to edges.

Path: A path describes a way of getting from one vertex to another in a graph. Here is the formal definition:

A path P in the graph $G = (V, E, l)$ from vertex u to vertex v , both in V , is a finite, ordered sequence of edges: $P = \langle e_1, e_2, \dots, e_k \rangle$, where $e_i \in E$ for all

$i \in [1, \dots, k]$, $e_1 = (u, \cdot)$, $e_k = (\cdot, v)$, and if $e_i = (\cdot, w)$ then $e_{i+1} = (w, \cdot)$ for all $i \in [1, \dots, k - 1]$.

The length of P is $L(P) = \sum_{i=1}^k l(e_i)$. The cardinality of P is $|P| = k$.

Distance: The distance d from a vertex u to a vertex v in G is defined as $d(u, v) = \min\{L(P) \mid P \text{ is a path in } G \text{ from } u \text{ to } v\}$.

SSSP: A single-source shortest paths (SSSP) algorithm takes as input a graph G and a source vertex s . It produces an n -sized table of all $d(s, \cdot)$ -values in G .

MSSP: A multiple-sources shortest paths (MSSP) algorithm takes as input a graph G and a list of κ source vertices. It produces a κn -sized table of all $d(t, \cdot)$ -values in G , where t is in the list of sources.

APSP: An all-pairs shortest paths (APSP) algorithm takes as input a graph G . It produces an n^2 -sized table of all $d(\cdot, \cdot)$ -values in G .

2.2 101 Ways to Compute Shortest Paths

Monday morning, your boss enters your office exclaiming (as bosses do): “I have a graph. I need to find some shortest paths. What is the fastest algorithm?”. How do you respond? Unfortunately, you cannot give a straight answer right away (and bosses hate that). No algorithm for finding shortest paths is the fastest¹ on all types of graphs.

You need to know more about the graph: What kind of edges does it contain? What sort of weights do the edges have? Are they all integer? Are some of them negative? Is the graph sparse or dense? Do we need the exact result, or will an approximation suffice?

By now, your boss has left your office, and you are stuck with the graph problem. So let us take a look at the different types of algorithms and the graphs that suit them.

The following survey is based in part on [41].

¹While your boss is probably more interested in practical efficiency, we will only consider asymptotical running time in this theory chapter.

2.2.1 Notes on Shortest Paths Algorithms

Even though we will survey a broad range of shortest paths (SP) algorithms, we have to set limits, as the field of SP is quite large. We restrict ourselves to *exact, deterministic* SP-algorithms. So we exclude algorithms that are either randomized or provide only approximate results.

We will allow the algorithms to use different computational models. Those that handle real-weighted graphs use a comparison-addition model, where pointers and arrays are allowed. Values can have one of two types: “real” which represents a subset of \mathbb{R} , or “integer”, which represents a subset of \mathbb{Z} . The only operations available on the values of type real are additions and comparisons. Values of type integer can also be added and compared, and they can furthermore be used for array indexing. Conversion between the types is not allowed. The weights of the input graph are of type real, new reals are only created from existing reals, and they are allowed to be arbitrarily large. Integers can be created at will, but must be polynomially bounded in the size of the input. Implementations of more advanced operations are given in appendix A. This computational model is also the one used in the rest of the theory chapter following this survey.

The algorithms that operate on integer weights work in the word RAM model which again permits pointers and arrays. It is further assumed that input graph weights fit in a machine word. Algorithms are allowed to use addition, comparison, array-indexing, bit-shifting, bitwise boolean operations, and multiplication as constant-time operations. However, out of the word RAM algorithms we survey, only the Han algorithm [21] and the linear Thorup algorithm [38] require constant-time multiplication.

Where we allow negative edge weights, we face the problem of negative cycles in the input graph. If we allow the exploitation of negative cycles, the solution values are not very interesting: they become $-\infty$. If we try to remedy this by forbidding the use of the same edge twice in the same path, the problem of finding shortest paths becomes NP-hard. This is proved in appendix B. But if we avoid graphs containing negative cycles altogether, the problem suddenly becomes tractable. The presence of negative cycles can be detected in $O(mn)$ time using e.g. the Bellman-Ford algorithm [2, 13]. Notice that undirected graphs with negative weights always contain negative cycles.

Edges	Weights	Author	Running time
Undirected	\mathbb{R}	(Appendix B)	NP-hard
	\mathbb{R}_+	Dijkstra [9]	$O(m + n \log n)$
		Pettie-Ramachandran [32]	$O(m\alpha(m, n) + \min \left\{ \begin{array}{l} n \log n \\ n \log \log \left(\frac{l_{max}}{l_{min}} \right) \end{array} \right\})$
	\mathbb{Z}	(Appendix B)	NP-hard
\mathbb{N}	Thorup [38]	$O(m)^*$	
Directed	\mathbb{R}	(Appendix B)	NP-hard
	\mathbb{R} NNC	Bellman-Ford [2, 13]	$O(mn)$
	\mathbb{R}_+	Dijkstra [9, 14]	$O(m + n \log n)$
	\mathbb{Z}	(Appendix B)	NP-hard
	\mathbb{Z} NNC	Goldberg [19]	$O(m\sqrt{n} \log l_{min})$
		Bellman-Ford [2, 13]	$O(mn)$
\mathbb{N}	Hagerup [20] Han [21] Raman [35] ...	$O(m \log \log l_{max})$ $O(n \log \log n \log \log \log n)^*$ $O(m + n\sqrt{\log n \log \log n})$...	

*: requires constant-time multiplication.

Table 2.1: SSSP algorithms

2.2.2 SSSP

An overview of the current fastest SSSP algorithms is shown in table 2.1. In the table, NNC means No Negative Cycles, l_{min} is the smallest edge weight in the input graph, and l_{max} is the largest.

Undirected Graphs

For undirected graphs there are not too many SSSP algorithms to consider. As mentioned above, the presence of negative edge weights renders the problem either uninteresting or most likely intractable, so we focus on non-negative weights.

For graphs with integer weights, Thorup has produced a remarkable $O(m)$

algorithm that was the first to utilize the component hierarchy approach. However, as previously mentioned, it requires a constant-time multiplication operation to achieve the optimal linear-time bound. In the absence of constant-time multiplication, the running time becomes $O(m\alpha(m, n))$, where α is the extremely slow-growing inverse Ackermann function (see appendix C for details on Ackermann).

For real-weighted graphs we have the Dijkstra algorithm [9] as described in section 2.4 using e.g. a Fibonacci heap [14] as the priority queue, achieving running time $O(m + n \log n)$. The Fibonacci heap was invented by Fredman & Tarjan in 1987, so the algorithm perhaps ought to be called Dijkstra-Fredman-Tarjan, but traditionally, we just write “Dijkstra”.

There exists a quite different APSP version of Dijkstra due to Karger et al. [24], which runs in time $O(m'n + n^2 \log n)$, where m' is the number of “optimal” edges, i.e. the number of edges that are part of a shortest path. But of course $m' = m$ in the worst case, so it represents no asymptotical improvement.

Pettie & Ramachandran [32] have constructed an MSSP algorithm that, when used as SSSP, has a general worst-case time that is slightly slower than Dijkstra’s. But for certain classes of graphs (for instance those with very small variance in edge weights) it beats Dijkstra.

Directed Graphs

For graphs that include both positive and negative weights, we again restrict ourselves to those without negative cycles to avoid NP-hardness. Then, the very simple Bellman-Ford textbook algorithm [2, 13] is still the best there is. It works for real weights (so of course for integers as well) and runs in time $O(mn)$. For non-negative real weights, Dijkstra offers the usual $O(m + n \log n)$ time.

Now, as the input gets simpler, the algorithms become more complicated. For integer-weighted graphs without negative cycles, Goldberg has described an algorithm in [19] that uses integer-specific techniques to achieve an $O(m\sqrt{n} \log |l_{min}|)$ running time. And when we restrict the weights to natural numbers, there is a plethora of algorithms with various near-linear running times. See table 2.1 for a few examples.

Edges	Weights	Author	Running time
Undirected	\mathbb{R}	(Appendix B)	NP-hard
	\mathbb{R}_+	Floyd-Warshall [12, 40] Pettie-Ramachandran [32]	$O(n^3)$ $O(mn \log \alpha(m, n))$
	\mathbb{Z}	(Appendix B)	NP-hard
	\mathbb{N}	Thorup [38]	$O(mn)^*$
Directed	\mathbb{R}	(Appendix B)	NP-hard
	\mathbb{R} NNC	Floyd-Warshall [12, 40] Johnson-Pettie [23, 31]	$O(n^3)$ $O(mn + n^2 \log \log n)$
	\mathbb{R}_+	Floyd-Warshall [12, 40] Pettie [31]	$O(n^3)$ $O(mn + n^2 \log \log n)$
	\mathbb{Z}	(Appendix B)	NP-hard
	\mathbb{Z} NNC	Floyd-Warshall [12, 40] Hagerup [20]	$O(n^3)$ $O(mn + n^2 \log \log n)$

*: requires constant-time multiplication.

Table 2.2: APSP algorithms

Notice how including negative weights severely hurts the running time in almost all SSSP cases. Thorup's linear algorithm breaks down and the problem becomes NP-hard, Dijkstra breaks down and the time becomes worst-case cubic instead of quadratic in n , and so on.

2.2.3 APSP

An overview of the current fastest APSP algorithms is shown in table 2.2.

Undirected Graphs

For the case of non-negative real weights, we distinguish between sparse and dense graphs, i.e. how many edges are present in the graph. For dense graphs (close to n^2 edges) the best choice is the simple Floyd-Warshall algorithm [12, 40] which runs in time $O(n^3)$.

For sparse graphs, we could apply the Dijkstra SSSP algorithm from each vertex for a total running time of $O(mn + n^2 \log n)$, but this is beaten by

the quite complex, hierarchy-based Pettie-Ramachandran MSSP algorithm, which (if used as APSP) has a running time of $O(mn \log \alpha(m, n))$.

For natural numbered weights, we can use Thorup's linear SSSP algorithm n times, yielding a total time of $O(mn)$.

Directed Graphs

In the realm of real weights, the situation is as follows. We can still use Floyd-Warshall for dense graphs, but if the graph is sparse, Floyd-Warshall is beaten by a hierarchy-based Pettie algorithm [31] (different from the Pettie-Ramachandran algorithm above) that runs in time $O(mn + n^2 \log \log n)$.

If the weights can be negative (but, as always, without negative cycles), we can use a scheme by Johnson [23] to compute new non-negative weights that preserve shortest paths. Then we apply Pettie, and finally reverse the Johnson scheme to find the actual path lengths. The edge transformation takes time $O(mn)$, so total time for Johnson-Pettie is still $O(mn + n^2 \log \log n)$.

As in the undirected case, we could use Dijkstra instead of Pettie, but it is again slower at $O(nm + n^2 \log n)$.

For integer weights we can use Floyd-Warshall or a hierarchy-based algorithm by Hagerup [20] that runs in $O(mn + n^2 \log \log n)$.

2.2.4 Where Are We?

Now we have an overview of the landscape of deterministic, exact shortest paths algorithms. But where does the subject of this thesis fit in? The answer is the small corner of the world defined by *APSP on directed, non-negative, real-weighted graphs*. In this corner, the traditional choice has been Dijkstra, but as we saw in section 2.2.3, the fastest algorithm is now Pettie, the latest addition to the family of hierarchy-based algorithms.

The hierarchy approach to the shortest paths problem was invented by Mikkel Thorup in 1999 based on a 1978 observation by Dinic [10]. As mentioned in the survey, Thorup's algorithm works only for undirected,

integer-weighted graphs. But soon after, Thorup’s algorithm was modified to work on directed graphs by Torben Hagerup who also presented a clearer exposition of the principles underlying hierarchy-based algorithms. In 2002, Seth Pettie and Vijaya Ramachandran used a minimum-spanning-tree technique to apply the hierarchy approach to real-weighted, undirected graphs. The practical value of this algorithm was shown to be promising in [33]. Finally, in 2003 Pettie produced the algorithm that is the focal point of this thesis, working on real-weighted, directed graphs.

Among the hierarchy-based algorithms, the Pettie algorithm works on the most general graphs and in the weakest computational model. So in that environment we will spend the remainder of this thesis. We will focus on two algorithms: The faster, quite complicated, new Pettie algorithm, and the straight-forward, classic Dijkstra.

Hence, a “graph” will from now on mean a non-negative, real-weighted, directed graph, unless otherwise stated. We will also assume that graphs are strongly connected. If they are not, we can make them so by adding edges with weights greater than the sum of all other weights. Doing this is not strictly necessary, but it helps simplify our arguments and does not influence the shortest paths computed.

Now we take a look inside the algorithms to understand what is going on.

2.3 High-level Generic Algorithm

When viewed at a very high level, the classic Dijkstra and the hierarchy-based Pettie are quite similar. Algorithm 1 shows a generic SSSP pseudocode skeleton shared by Pettie and Dijkstra. Both algorithms implement APSP by applying this SSSP procedure n times.

The algorithm maintains a set S of visited vertices which is initially empty. In each iteration of the while-loop, one vertex is added to S . Once all vertices are in S , we are done.

We also keep track of a D -value for each vertex. This is a tentative distance measure – an upper bound on the actual distance from the source to the vertex. Initially, D is set to ∞ for every vertex except the source, where it is 0. At termination, the bound has become perfect, i.e. the D -values are then equal to the $d(s, \cdot)$ -values.

Algorithm 1 *Generic SSSP pseudocode*

Input: A graph $G = (V, E, l)$ and a starting vertex $s \in V$ **Output:** $\forall v \in V : D(v) = d(s, v)$

```

1:  $D(s) \leftarrow 0$  and  $D(v) \leftarrow \infty$  for all  $v \neq s$ 
2: set of vertices  $S \leftarrow \{s\}$ 
3: while  $|S| < |V|$  do
4:    $w \leftarrow$  an appropriate vertex in  $V \setminus S$ 
5:   for all edges  $(w, v)$  where  $v \in V \setminus S$  do
6:      $D(v) = \min\{D(v), D(w) + l((w, v))\}$ 
7:   end for
8:    $S \leftarrow S \cup \{w\}$ 
9: end while

```

When a vertex is to be added to S , its outgoing edges are *relaxed* (lines 5-7) to decrease the D -values of its neighbors, if possible.

The only thing separating the Dijkstra and Pettie algorithms is the manner in which they choose the next vertex to be visited (the “appropriate vertex” in line 4). In the sections ahead we will see how each algorithm makes its choices.

We should mention that this generic algorithm skeleton by no means covers every imaginable shortest paths algorithm. Many of the algorithms described in section 2.2 (e.g. Floyd-Warshall) do not fit into this model.

Note: While algorithm 1 only shows how to find the *lengths* of shortest paths, it is possible to modify it to obtain the actual paths without slowing things down. We simply maintain an *origin* for each vertex, so that when an edge (v, w) is relaxed to a lower D -value, we set w 's origin to be v . The updates along the way each take constant time, and afterwards we can answer the question “What is the shortest path from s to w ?” in optimal $O(|P|)$ time, where $L(P) = d(s, w)$, by following the origin pointers from w to s and then reversing the resulting path.

2.4 Dijkstra's Algorithm

Now we have seen a generic algorithm which describes most of Dijkstra's behavior. Apart from this, variants of the Dijkstra algorithms usually just

use one data structure: a priority queue Q containing the vertices of $V \setminus S$ prioritized by their D -values. Whenever an edge is relaxed or a vertex is added to S , Q is updated. When we have to select the next vertex to visit, we choose the the vertex in Q with the lowest D -value and remove it from Q .

In Dijkstra’s original article, Q was implemented as an unordered list. Since then, scores of other implementations have been invented. Typically as some form of heap: binary heap, pairing heap, Fibonacci heap, etc. Other ideas have been tried as well: red-black search trees, skip lists, multiple queues, and more. The best asymptotic running time so far is achieved by using a Fibonacci heap [14].

Dijkstra is an example of a *greedy* algorithm. It tries to optimize the global solution by always making locally optimal choices. In the greedy approach employed by Dijkstra, the “appropriate vertex” is the unvisited vertex closest to s . This means that the set of visited vertices is only expanded by pulling in its neighbors.

This makes Dijkstra inherently as hard as sorting: We can sort n numbers using Dijkstra by placing them as weights on the edges of a star-shaped graph, and use the center as source. Then the Dijkstra SSSP will follow the edges in sorted order, essentially performing a heap-sort.

Dijkstra’s way of choosing the “appropriate vertex” is not a necessary condition for correctness. An algorithm can easily be correct while visiting the vertices in a completely different order. This will become clear later when we prove the correctness of the *non-greedy* Pettie algorithm.

But then how does the Pettie algorithm determine which vertex is next? This is where the component hierarchy comes into play.

2.5 The Component Hierarchy Approach

The component hierarchy is a data structure used by hierarchy-based algorithms such as the Pettie algorithm to determine the order in which to visit the vertices of the input graph and perform edge relaxation.

Given the input graph, we construct a tree which we call the component hierarchy. The leaves of the tree correspond to the vertices of the graph,

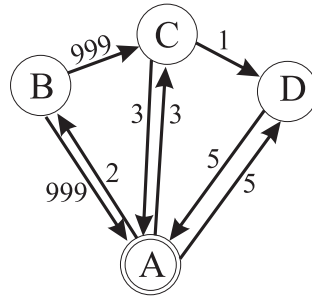


Figure 2.1: *Tiny graph where sorting is not needed*

and the internal nodes correspond to different carefully chosen subgraphs of the input graph. How to select these subgraphs will be explained later.

Once the hierarchy is built, a recursive VISIT procedure is let loose on it. This procedure traverses the tree by calling itself on certain hierarchy nodes with certain value intervals as arguments. In this way, it climbs around in the hierarchy while manipulating a few other auxiliary data structures.

When the VISIT procedure reaches a hierarchy leaf, the corresponding graph vertex becomes the “appropriate vertex” in line 4 in the generic algorithm. In other words, we add vertices to S in the order in which their hierarchy counterparts are visited by the VISIT recursion.

Why is this clever? All this building and climbing trees may sound like a lot of extra work compared to Dijkstra. But as we will prove later, the component hierarchy helps beat Dijkstra on the APSP problem. Intuitively, this is because we build the hierarchy quickly, once and for all before all the iterations of the generic algorithm. Moreover, we make sure that the hierarchy size is linear in n . So if we can keep the number of VISIT calls to a minimum while using very fast auxiliary data structures, we can achieve a better running time than the one caused by the sorting bottleneck in Dijkstra.

2.5.1 Sorting Is Unnecessary

We will try to convey an intuitive idea of why it is not necessary to visit the vertices in order sorted by their distance to the source. Suppose the tiny graph in figure 2.1 is our input graph and vertex A is the source.

Given this input, the Dijkstra SSSP will first visit B, then C, and finally D because of the sorting property. But what would happen if the order was C,D,B or C,B,D? Nothing. The algorithm would still give the same result, because B is somehow independent from C and D.

Then how about D,C,B? This would not work, because C must be visited before D to utilize the short edge (C, D). So the order cannot be completely arbitrary, but on the other hand sorting the vertices is overkill. We need a middle ground, and this is exactly what the component hierarchy provides.

2.5.2 The Underlying Idea

The component hierarchy approach is a way to classify which graph vertices are somehow “close together” and which are “far apart”. In this sense, it is a kind of divide-and-conquer approach where we try to divide the problem into independent subproblems. The idea being that we can solve the problem for vertices that are “close together” without considering those that are “far away” from them.

It is not an easy job dividing shortest paths into independent subproblems, because the shortest path between two vertices can be a hamiltonian path and thus visit every vertex in the graph.

In *quick-sort*, a standard divide-and-conquer algorithm, the dividing is done by one parameter: structure. One recursive call handles one part of the array, another call deals with the rest of the array, and afterwards we combine the solutions.

The hierarchy approach is a way to divide the problem by two parameters: structure and weight values. This means that the graph is divided into subgraphs that are independent with respect to shortest paths of a certain length. Simplistically, we can say that the edge structure of the graph determines which hierarchy nodes will exist, but the weights determine at which level a node is placed in the tree.

Let us return to the small example from earlier. The hierarchy corresponding to this input graph can be seen in figure 2.2.

As we can see, the leaves in the hierarchy that correspond to the graph vertices C, D, and A share a common parent, H_2 . This indicates that they are “close together” and that B is farther away. More precisely, it means

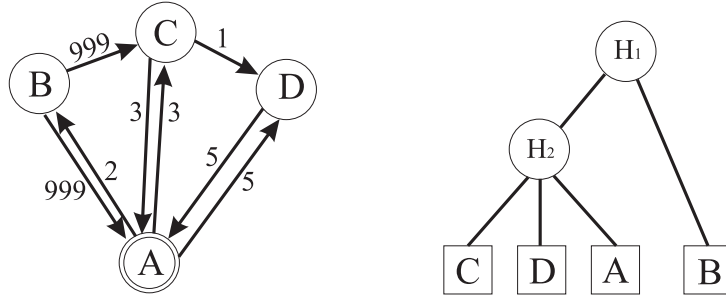


Figure 2.2: A tiny graph and its component hierarchy

that with respect to shortest paths of a certain length, we can consider the subgraph induced by C, D, and A an independent subproblem.

2.5.3 Handling Independent Subproblems

To illustrate how we will exploit the independence of subproblems, consider the following example (due to Pettie [31]).

Suppose we have a graph $G = (V, E, f)$ where V can be split into two non-empty vertex sets V_1 and V_2 such that every edge between V_1 and V_2 has weight at least t .

Now suppose we have three processes p_1 , p_2 , and p_{top} that are able to pass messages containing real intervals to one another. We assign p_1 and p_2 to handle V_1 and V_2 respectively, while p_{top} has the job of overseeing the other two processes.

If p_{top} sends the interval $[a, b)$ to p_i , it means that p_i must locate and visit all vertices $v \in V_i$ where $d(s, v) \in [a, b)$

Now we can construct a very simple SSSP algorithm: p_{top} passes the following intervals to both p_1 and p_2 : $[0, t)$, $[t, 2t)$, $[2t, 3t)$, \dots . This continues until all $v \in V$ are visited.

Now, what is the point of this construction? The point is that when p_i receives such a t -interval, it can do its job by *only* looking at the edges and D -values belonging to V_i . In other words, we have split the original problem into two independent subproblems.

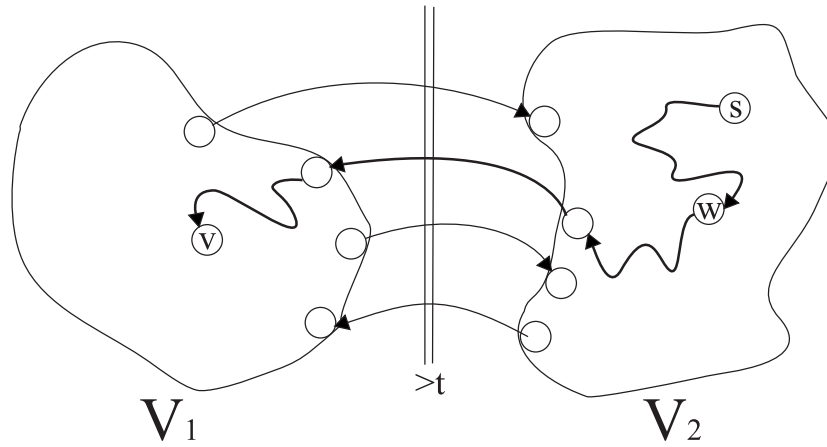


Figure 2.3: $d(s, v)$ and $d(s, w)$ cannot be in the same t -interval

How do we know they are really independent? Simple: it stems from our choice of interval-width, t . Let $v \in V_1$ be such that p_1 must visit it in round i , that means $d(s, v) \in [it, it + t)$. Then there can be no $w \in V_2$ with $d(s, w) \in [it, it + t)$ on the shortest path from s to v , because the shortest path from w to v has length at least t by definition of V_1 and V_2 (see figure 2.3).

So we have made sure that if a shortest path crosses the gap between V_1 and V_2 , then vertices from that path on different sides of the gap cannot be visited in the same round.

This was only a small example. In general we will have more than two subproblems and a more intricate interval passing scheme, but the basic idea remains the same. Pass around “permission intervals” that are narrow enough to ensure independence of subproblems.

2.5.4 Using the Hierarchy

Suppose we have a full-fledged component hierarchy. How do we use it? Quite the same way as in the small example in the previous section: We use it to pass around interval-messages. So now, the entities that pass the intervals are no longer “processes”, but simply the nodes of the hierarchy.

At the top of the hierarchy, the root node *Root* starts things off by receiving a message containing the interval $[0, \infty)$. When an internal hierarchy node

receives an interval, it partitions it into a list of new fixed-width intervals. The width of the new intervals depends on the size of the subgraph that the hierarchy node represents. It then passes the new smaller intervals on to its children, one by one, and so on.

When a leaf node receives an interval message, it checks whether its D -value (the D -value of the graph vertex v that it represents) is in the interval. If it is, then the v is visited, and its edges are relaxed; v has become the “appropriate vertex” from algorithm 1.

In order for this scheme to work, it is necessary to prioritize the children of each hierarchy node. This is done in various ways in different hierarchy-based algorithms. In section 2.6.1 we explain how the Pettie algorithm does it.

Now, once we understand that the component hierarchy can be useful in finding shortest paths, new questions arise: Exactly how do we build it? Does it work? And how do we use it efficiently, e.g. by not issuing too many interval messages? These questions are best answered based on a concrete algorithm. So now we move on to the specifics of the hierarchy-based Pettie algorithm.

2.6 Pettie's Algorithm

At the outermost level, the Pettie APSP algorithm consists of two phases: building a component hierarchy and applying the recursive VISIT procedure n times on the hierarchy.

2.6.1 Building the Component Hierarchy

The component hierarchy, \mathcal{H} , is essentially a tree. It has a number of leaves and some internal nodes, one of which is the root.

Each node in the hierarchy represents a part of the input graph. Stated more precisely, a hierarchy node represents a strongly connected component of the subgraph produced by limiting the edge set of the input graph to just those edges with weights in a certain interval.

A strongly connected component (SCC) is a largest possible subgraph in which every vertex is reachable from any other vertex.

The higher a node is positioned in the hierarchy, the larger the component it represents. A leaf in the hierarchy represents a single graph vertex and the hierarchy root represents the entire input graph.

The Build-Graph

When we set out to construct the hierarchy, we start from having just the leaves and employ a bottom-up construction. We do this by making a temporary graph called the build-graph. At first, the build-graph contains no edges, it just consists of the n input graph vertices.

Now we begin adding the input graph edges to the build-graph in measured quantities and in a particular order. Once we have added a group of edges, we find the newly emerged SCCs and create a new hierarchy node for each of them. Then, every new hierarchy node x is made parent of the nodes that represent parts of x 's SCC in the build-graph.

Henceforward, we do not need to remember what vertices and edges an SCC consists of. Therefore, we contract each of the SCCs into a single “compressed” vertex in the build-graph. Now we repeat the procedure: edges are added to the build-graph, new SCCs emerge (that may include compressed vertices), hierarchy nodes are created, SCCs are contracted, and so forth.

At the end, we have added all the edges and contracted all the SCCs. All that is left in the build-graph is a single compressed vertex which “contains” the entire input graph and corresponds to the root of the hierarchy.

From here on out, we can forget all about the build-graph and make use of our freshly constructed hierarchy along with the input graph.

But how can we know that we end up with exactly one compressed vertex (and therefore one hierarchy tree)? This is because it does not matter that we contract it bite for bite, the end result is the same as that of one big contraction. And we will always end up with a single compressed vertex, because we require the whole input graph to be an SCC.

Strata and Levels

In the previous section we were deliberately vague about the details of adding edges to the build-graph. We will now describe this process more precisely.

What we want to do is to divide the edges into groups, so that edges in the same group have roughly the same weight. We will then add the edges to the build-graph group by group. This will ensure that the resulting hierarchy is useful for our purpose (further explanation and proofs later).

We have two layers of such groups: strata and levels. A stratum may contain many levels, and a level may contain many edges. As we have seen, it is the addition of edges to the build-graph that causes creation of hierarchy nodes. Therefore, we will associate strata and levels with edges as well as the hierarchy nodes they induce.

To assign strata and level numbers to the edges, we first sort all the input edges by weight in non-decreasing order². Let l_1, \dots, l_m be the sorted edge weights. Then we find edge weights that are much larger than their neighbor in the sorted list. These *normalizing edge weights* mark the beginning of a new stratum. As the normalizing weights we choose $\{l_1\} \cup \{l_i : l_i > 2n \cdot l_{i-1}\} \cup \{\infty\}$. Let l_j^* denote the j^{th} smallest normalizing weight.

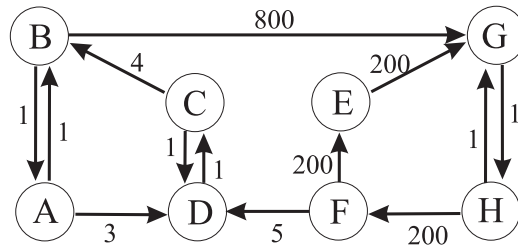
An edge of weight l_k , where $l_j^* \leq l_k < l_{j+1}^*$, is at stratum j . We define its level as the minimum i such that $l_k < l_j^* \cdot 2^i$. In other words, $i = \lfloor \log(\frac{l_k}{l_j^*}) \rfloor + 1$, except our computational model does not directly allow such operations.

Figure 2.4 shows an example graph, along with a table of the levels and strata of its edges, as well as the resulting hierarchy.

As mentioned above, we associate strata and levels with the induced hierarchy nodes as well. However, the hierarchy leaves are not induced by any edges, but we just define all leaves to be at stratum 1, level 0.

We can use the stratum/edge-labeling of hierarchy nodes to define the following very useful measure:

²Note that this does not cause a Dijkstra-like sorting bottleneck, as we only build the hierarchy once to solve APSP.



weight	1	1	1	1	1	3	4	5	200	200	200	200	800
stratum	1	1	1	1	1	1	1	1	2	2	2	2	2
level	1	1	1	1	1	2	3	3	1	1	1	1	3

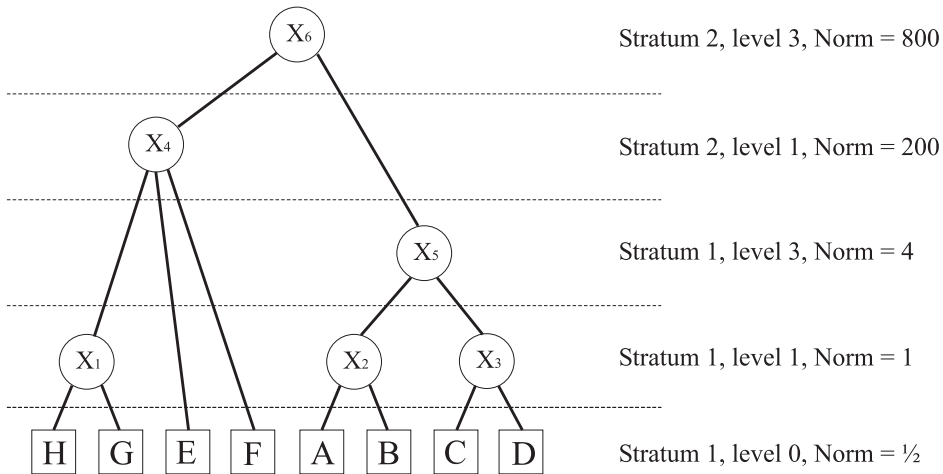


Figure 2.4: An example graph, a table showing the levels and strata of its edges, and the resulting hierarchy.

Definition 1

$$Norm(x) = l_j^* \cdot 2^{i-1} \text{ where } x \text{ is on level } i \text{ in stratum } j.$$

We will show the usefulness of this *Norm* shortly. First, we need a few more definitions.

We will refer to the subgraph corresponding to a hierarchy node x as C_x .

By the construction above, C_x is an SCC in the input graph restricted to edges shorter than $2 \cdot \text{Norm}(x)$.

By $p(x)$ we will mean x 's parent, and by $\text{Children}(x)$ we will mean x 's set of hierarchy children. We denote $|\text{Children}(x)|$ by $\text{Deg}(x)$. If we let $\text{Children}(x) = \{y_1, \dots, y_{\text{Deg}(x)}\}$, then we define C_x^c as the result of taking C_x , contracting all C_{y_i} , and keeping only the edges shorter than $\text{Norm}(x)$. The nodes of C_x^c then correspond to the children of x . See figure 2.5 for an example.

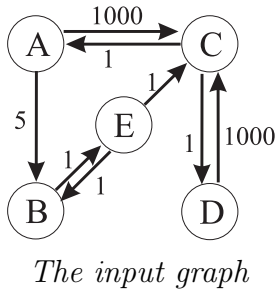
Prioritizing Siblings

An important part of what makes the hierarchy work is that siblings in the hierarchy are prioritized. That is, they appear in their parent's list of children in a certain order. What we want to do is to order x 's children according to a topological ordering on C_x^c . Intuitively, this is so that the shortest path to a node is only affected by its siblings "to the left" when we consider a certain permission interval. This will help us later in the VISIT recursion. For a topological ordering to be possible, we need to prove that C_x^c is acyclic:

Lemma 2 $\forall x \in \mathcal{H} : C_x^c$ is acyclic

Proof: Let x be a node at stratum i , level j having child y . By definition, all edges in C_y are shorter than $2 \cdot \text{Norm}(y)$. If y is at stratum i , level j then $2 \cdot \text{Norm}(y) = 2 \cdot l_j^* \cdot 2^{i-2} = l_j^* \cdot 2^{i-1} = \text{Norm}(x)$. If y is on a lower level, then no edges with weights in $[2 \cdot \text{Norm}(y), \text{Norm}(x))$ could cause a cycle if added to C_y – otherwise x would not be y 's parent but a more distant ancestor. Being an SCC, C_y is a *maximal* strongly connected subgraph. By definition, if C_x^c contained a cycle then all the participating edges would be shorter than $\text{Norm}(x)$. This would contradict the maximality of C_y . \square

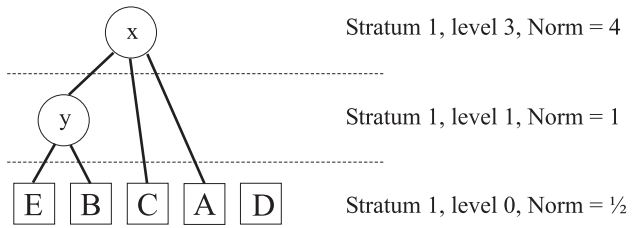
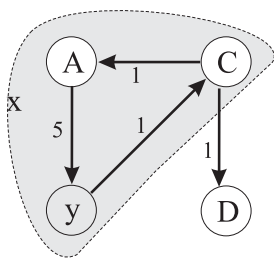
Now we arrive at the really useful property of the *Norm*: we can use it as the width of the "permission interval" introduced in section 2.5.3. In other words, the subgraphs represented by sibling nodes are at least *Norm* of their parent apart. Actually, using the topological ordering above, we just need that every edge that "goes to the left" among siblings is at least *Norm*.



weight	1	1	1	1	1	5	1000	1000
stratum	1	1	1	1	1	1	2	2
level	1	1	1	1	1	3	1	1

The edges

After adding the edges of length 1 to the build-graph, the vertices E and B form an SCC, so we contract them into y . Then, as we add the edge of length 5, the nodes A , C , and y form an SCC, so we create x :



Then C_x is the graph induced by $\{A, B, C, E\}$ restricted to edges shorter than $2\text{Norm}(x) = 8$. And C_x^c is the graph where B and E are contracted to form y , and we restrict ourselves to edges shorter than $\text{Norm}(x) = 4$.

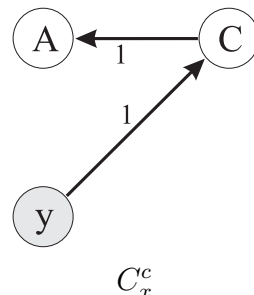
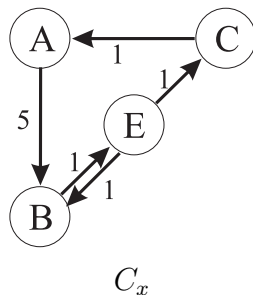


Figure 2.5: A demonstration of the build-graph, C_x , and C_x^c for a small graph.

Lemma 3 *Let $y_L, y_R \in \text{Children}(x)$ such that y_L is somewhere to the left of y_R in x 's list of children. If $(v_R, v_L) \in E$ is such that $v_R \in V(C_{y_R})$ and $v_L \in V(C_{y_L})$ then $l(v_R, v_L) \geq \text{Norm}(x)$.*

Proof: Suppose $l(v_R, v_L) < \text{Norm}(x)$. Then that edge must exist in C_x^c as (C_{y_R}, C_{y_L}) . This means that C_{y_R} precedes C_{y_L} in the topological ordering of C_x^c . This implies that y_R is to the left of y_L which contradicts the premise. \square

So now we know what is required to build the hierarchy. We need to sort the edges, find values for levels and strata, and add edges to the build-graph while finding SCCs and sorting topologically. Let us move on to the actual SP-computations.

2.6.2 Computing Shortest Paths

Once we have the component hierarchy ready, we can start running some SSSP computations. At the heart of the Pettie SSSP algorithm is the recursive VISIT procedure shown in algorithm 2. To understand what a call to VISIT actually accomplishes, we need a definition:

$$X^{[a,b]} = \{v \in X : d(s, v) \in [a, b]\} \quad \text{where } X \text{ is a vertex set and } a, b \in \mathbb{R} \quad (2.1)$$

In other words, X^I is the vertices in X whose distances from the source are in I .

The VISIT procedure is called on a hierarchy node x with an interval argument $[a, b)$. When the call returns, the vertices $V(C_x)^{[a,b)}$ have all been added to S , the set of visited vertices. We prove that this is correct in section 2.9.

So, solving SSSP is done as follows: Just as in the generic algorithm, set $S = \{s\}$, and $D(x) = \infty$ for all graph vertices x , except $D(s) = 0$. Then call $\text{VISIT}(\text{Root}, [0, \infty))$. When this call returns, $S = V$ and the lengths of the shortest paths from s can be read in the D -values of the vertices.

However, looking at algorithm 2, we can see that it is not quite that simple. We will start at something familiar by looking at lines 3-6 of the pseudocode. They are exactly the same as those in the generic algorithm that perform edge relaxations and add a vertex to S . Looking at lines 1

Algorithm 2 *The VISIT algorithm*

Input: Component hierarchy node x , real value interval $[a, b)$ **Output:** Ensures that $V(C_x)^{[a,b)} \subseteq S$. Returns whether x needs more visits.

```

1: if  $x$  is a leaf then
2:   if  $D(x) \in [a, b)$  then
3:      $S \leftarrow S \cup \{x\}$ 
4:     for all edges  $(x, v)$  in the graph do
5:        $D(v) = \min\{D(v), D(x) + l(x, v)\}$ 
6:     end for
7:     return false
8:   else
9:     return true
10:  end if
11: else
12:  if this is the first VISIT call on  $x$  then
13:     $NotDone(x) \leftarrow$  number of children of  $x$ 
14:     $t_x \leftarrow \begin{cases} D(x) & \text{if } D(x) + Diam(x) < b \\ b - Norm(x) \lceil \frac{b-D(x)}{Norm(x)} \rceil & \text{otherwise} \end{cases}$ 
15:    initialize  $x$ 's buckets: each has width  $Norm(x)$ , the first starts in  $t_x$ 
16:     $a_x \leftarrow t_x$ 
17:  else
18:     $a_x \leftarrow a$ 
19:  end if
20:  while  $a_x < b$  and  $NotDone(x) > 0$  do
21:    while bucket  $[a_x, a_x + Norm(x))$  is not empty do
22:       $y \leftarrow$  the leftmost child in bucket  $[a_x, a_x + Norm(x))$ 
23:       $yAgain \leftarrow VISIT(y, [a_x, a_x + Norm(x)))$ 
24:      if  $yAgain$  then
25:        move  $y$  to bucket  $[a_x + Norm(x), a_x + 2Norm(x))$ 
26:      else
27:        remove  $y$  from bucket  $[a_x, a_x + Norm(x))$ 
28:         $NotDone(x) \leftarrow NotDone(x) - 1$ 
29:      end if
30:    end while
31:     $a_x \leftarrow a_x + Norm(x)$ 
32:  end while
33:  return  $(NotDone(x) > 0)$ 
34: end if

```

and 2, we see that these edge relaxations are only performed when the VISIT recursion reaches a leaf with an appropriate D -value.

Regarding the D -values, we maintain essentially the same invariant as Dijkstra:

Invariant 4 For $v \in S : D(v) = d(s, v)$, and for $v \notin S : D(v)$ is the distance from s to v in the subgraph induced by $S \cup \{v\}$.

We store all D -values in an auxiliary data structure, \mathcal{DSF} , described in section 2.8.2.

The rest of the VISIT procedure (lines 11-34) deals with what happens when it is called on internal nodes. To avoid making too many recursive VISIT calls, we assign to each hierarchy node a *bucket structure* in which to keep its children. Conceptually, a bucket structure is an array of buckets where the buckets are said to *cover* consecutive real intervals. We want to use the buckets to determine which leaf is next in line for a visit. Therefore, a hierarchy node x is stored in the bucket covering the lowest D -value in C_x . This leads to the following definition:

$$\forall x \in \mathcal{H} \quad : \quad D(x) = \min_{v \in C_x} \{D(v)\} \quad (2.2)$$

Note that the definition implies that the D -value of a graph vertex is equal to the D -value of the corresponding hierarchy leaf.

Getting back to the pseudocode for internal nodes, in lines 12-19, we first check whether this is the first VISIT call on the node. If it is, we initialize the bucket structure by assigning proper intervals to buckets. The initialization also includes bucketing those nodes whose D -values make them eligible for a place in a bucket. The variable $Diam(x)$ present in the calculation of intervals will be defined shortly.

In lines 20-32, we have two nested while-loops. The outermost loop takes care of advancing across the “permission interval” $[a, b)$ that we have been given. We do this in chunks of $Norm(x)$ as is visible in line 31. The innermost while-loop empties the current bucket. Child nodes are extracted in prioritized order (signified by “leftmost”) and VISIT is called recursively on $Norm$ -sized intervals.

$\text{VISIT}(x, \cdot)$ returns whether it needs to be called again with the next interval, i.e. whether $V(C_x) \not\subseteq S$. We use a simple counter $\text{NotDone}(x)$ to determine this.

An important fact about the structure of our component hierarchy is that as soon as $\text{VISIT}(x, \cdot)$ is called, the D -value of x is fixed, and is equal to $d(s, x)$, the distance from the source vertex s to x . See section 2.9 for a proof of this.

When to Bucket

To be very precise about when and where to bucket nodes, we present the following invariant. We refer to a node as *active* if VISIT has been called on it, and *inactive* otherwise.

Invariant 5 *Let $x, y \in \mathcal{H}$ be such that y is a child of x .*

If both x and y are inactive, then y appears in no bucket, because x has not initialized any yet.

If x is active and y is inactive, then either y appears in bucket $[q, q + \text{Norm}(x))$ and $D(y) \in [q, q + \text{Norm}(x))$ or $D(y) = \infty$ and y appears in no bucket.

If both x and y are active, then either y appears in a bucket $[q, q + \text{Norm}(x))$ and $V(C_y)^{[0, q)} \subseteq S$ and $V(C_y)^{[q, q + \text{Norm}(x))} \not\subseteq S$ or y appears in no bucket and $V(C_y) \subseteq S$.

To explain invariant 5, we will take a look at the bucket life of a node y . At the beginning of the VISIT recursion, y is not in any bucket. At some point, y 's parent becomes active, but y 's D -value may still be ∞ . So perhaps after awhile, y is put into a bucket. As edge relaxations happen, $D(y)$ decreases, and y may need to be moved to a lower-numbered bucket a number of times. After $D(y)$ reaches $d(s, y)$, VISIT is called on y , so it becomes active. From here on, as VISIT is called on y 's descendants, y may need to be moved back up to higher-numbered buckets until all of C_y has been added to S , whereupon it ceases to be bucketed. More on this chain of events in section 2.8.3.

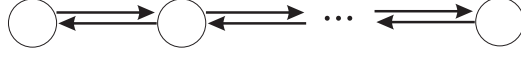


Figure 2.6: A “string-graph” results in a large diameter.

2.6.3 Properties of the Hierarchy

In the sections ahead, we will need to reason about the hierarchy. Thus, it will be helpful to know, for instance, the number of nodes in it, or the size of the graph each node represents. So in this section we prove a few important properties of the hierarchy.

We first define an upper bound on the longest internal distance of a node. That is: among all the shortest paths in a subgraph, which is the longest? We call this the *diameter* of a hierarchy node x :

Definition 6

$$Diam(x) = 2 \cdot Norm(x) \cdot (Deg(x) - 1) + \sum_{y \in Children(x)} Diam(y)$$

The definition reflects the fact that the type of graphs that give the largest diameter are “string-graphs” (similar to the one depicted in figure 2.6).

Now we bound the size of the hierarchy itself to being linear in n :

Lemma 7

$$\sum_{x \in \mathcal{H}} Deg(x) < 2n - 1$$

Proof: Building the hierarchy, we only create new hierarchy nodes on the basis of non-trivial SCCs, so each internal node has at least two children. There are n leaves, so there is a total of $2n - 1$ nodes. The sum counts all the nodes once, except the root which is not counted. \square

Next, we provide an upper bound on the normalized diameter of a node:

Lemma 8

$$\forall x \in \mathcal{H} : \frac{Diam(x)}{Norm(x)} < 2n$$

Proof: C_x is strongly connected and restricted to edges shorter than $2 \cdot \text{Norm}(x)$, so we have that

$$\text{Diam}(x) < (|V(C_x)| - 1) \cdot 2 \cdot \text{Norm}(x) < 2n \cdot \text{Norm}(x) \text{ implying the lemma.} \quad \square$$

We also compute an upper bound on the sum of normalized diameters for the entire hierarchy:

Lemma 9

$$\sum_{x \in \mathcal{H}} \frac{\text{Diam}(x)}{\text{Norm}(x)} < 4n$$

Proof: We denote by $\text{Anc}(k, y)$ the k^{th} ancestor of y . By $\text{Subtree}(x)$ we mean x and all its descendants in \mathcal{H} .

$$\begin{aligned} \sum_{x \in \mathcal{H}} \frac{\text{Diam}(x)}{\text{Norm}(x)} & \stackrel{=1}{=} \sum_{x \in \mathcal{H}} \frac{2 \cdot \text{Norm}(x) \cdot (\text{Deg}(x) - 1) + \sum_{y \in \text{Children}(x)} \text{Diam}(y)}{\text{Norm}(x)} \\ & = \sum_{x \in \mathcal{H}} \frac{\sum_{y \in \text{Subtree}(x)} 2 \cdot \text{Norm}(y) \cdot (\text{Deg}(y) - 1)}{\text{Norm}(x)} \\ & \stackrel{=2}{=} \sum_{y \in \mathcal{H}} \sum_{k \geq 0} \frac{2 \cdot \text{Norm}(y) \cdot (\text{Deg}(y) - 1)}{\text{Norm}(\text{Anc}(k, y))} \\ & \stackrel{<3}{<} \sum_{y \in \mathcal{H}} \sum_{k=0}^{\infty} \frac{2 \cdot \text{Norm}(y) \cdot (\text{Deg}(y) - 1)}{2^k \cdot \text{Norm}(y)} \\ & = \sum_{y \in \mathcal{H}} \sum_{k=0}^{\infty} \frac{(\text{Deg}(y) - 1)}{2^{k-1}} \\ & = \sum_{y \in \mathcal{H}} 4 \cdot (\text{Deg}(y) - 1) \\ & < 4 \cdot \left(-n + \sum_{y \in \mathcal{H}} \text{Deg}(y) \right) \\ & \stackrel{<4}{<} 4 \cdot (-n + 2n - 1) \\ & < 4n \end{aligned}$$

- 1: By definition of *Diam*.
- 2: Instead of summing over all descendants of each ancestor, we sum over all ancestors of each descendant.
- 3: By definition of *Norm*, it at least doubles for each level we ascend to.
- 4: By lemma 7

□

From lemma 9 we get the following corollary:

Corollary 10

$$\left| \left\{ x \in \mathcal{H} : \frac{Diam(x)}{Norm(x)} \geq k \right\} \right| < \frac{4n}{k}$$

Proof: If this was not true, then by lemma 9, $4n > k \cdot \frac{4n}{k}$, which is obviously false.

□

2.7 Pettie Running Time

In this section, we will bound the running time of the entire Pettie algorithm. However, we factor out the details of the Pettie algorithm's many subroutines to section 2.8. This means that for now, we will focus on the time to construct the hierarchy, and the bare-bones running time of the VISIT procedure, meaning that we exclude the time used on bucketing and handling *D*-values. At the end of the section, we will sum up the complete Pettie running time to find that it does indeed run in $O(mn + n^2 \log \log n)$ time.

2.7.1 Building the Hierarchy

It is possible to build the component hierarchy in time $O(m \log n)$ using a method from [31] inspired by the “batched-search” found in [20]. This might be useful when using Pettie on very particular graphs that cause only low-diameter hierarchy nodes. However, in general we can afford to implement the construction of the hierarchy in $O(mn)$ time. The proof of lemma 11 describes how.

Lemma 11 *The component hierarchy can be constructed in $O(mn)$ time.*

Proof: We basically follow the recipe from section 2.6.1. First, we sort the graph edges by weight. This takes $O(m \log n)$ time. Then we find the normalizing edge weights in $O(m \log n)$ time (multiplication by N takes $O(\log N)$ time, see appendix A). By construction, there are at most m strata, and $O(m \log n)$ levels total. We can find all the values at which a new level starts in time $O(m \log n)$ because multiplying by 2 takes $O(1)$ time.

We run through the sorted list of edges, adding them one at a time to the build-graph and pausing at the end of each level. When we add an edge, we check to see if the source or destination vertex in the build-graph has now achieved both in- and out-degree greater than zero for the first time. If so, we say that this vertex has *come alive*. At each level pause, we check if any vertices have come alive since the last level pause. If not, we just continue adding edges.

If a vertex has come alive, then we run the subroutine found in section 2.8.1 to find all SCCs in the build-graph. By lemma 14 this takes $O(m + n)$ time. Then we create the corresponding hierarchy nodes and connect them to their child nodes. To ensure that the children are in the correct order, we run a standard textbook topological sort (see e.g. [6]) on each SCC, where we disregard all the latest edges with weights greater than $Norm(x)$, so that the subgraph appears acyclic. This takes linear time in the size of the SCC, which means time $O(m + n)$ for each vertex come alive. Finally, we contract all SCCs into new build-graph vertices. This, too, takes $O(m + n)$ time.

We continue until there are no more edges to add. In total, n vertices come alive, so the total running time is $O(n(m + n)) = O(mn)$. \square

2.7.2 The Visit Calls

Once the hierarchy is built, we perform n SSSP computations. An SSSP computation consists of a number of recursive VISIT calls that traverse the hierarchy while handling bucketing and D -values. To bound the running time of an SSSP computation, we first bound the total number of recursive VISIT calls with the following lemma.

Lemma 12 *In one SSSP calculation, the number of recursive calls to the VISIT procedure is less than $5n$.*

Proof: First: How many times is $\text{VISIT}(x, I)$ called for some fixed x ? We will see in lemma 30 that at the time of the first call, $D(x) = d(s, x) \in I$. And by construction of the hierarchy, $\max\{d(s, y) \mid y \in \text{Children}(x)\} \leq d(s, x) + \text{Diam}(x)$. From algorithm 2 we can see that $|I|$ always equals $\text{Norm}(p(x))$. In other words, we are advancing across an interval of width $\text{Diam}(x)$, and we are taking it in chunks of width $\text{Norm}(p(x))$. This, along with invariant 5, implies that for some hierarchy node x , we call $\text{VISIT}(x, I)$ at most $\left\lceil \frac{\text{Diam}(x)}{\text{Norm}(p(x))} \right\rceil + 1$ times.

To find the total number of VISIT calls, we sum over all hierarchy nodes:

$$\begin{aligned}
\sum_{x \in \mathcal{H}} \left\lceil \frac{\text{Diam}(x)}{\text{Norm}(p(x))} \right\rceil + 1 &= |\mathcal{H}| + \sum_{x \in \mathcal{H}} \left\lceil \frac{\text{Diam}(x)}{\text{Norm}(px)} \right\rceil \\
&\leq^1 |\mathcal{H}| + \sum_{x \in \mathcal{H}} \left\lceil \frac{\text{Diam}(x)}{2\text{Norm}(x)} \right\rceil \\
&= |\mathcal{H}| + \sum_{\text{internal } x \in \mathcal{H}} \left\lceil \frac{\text{Diam}(x)}{2\text{Norm}(x)} \right\rceil + \sum_{\text{leaf } x \in \mathcal{H}} \left\lceil \frac{\text{Diam}(x)}{2\text{Norm}(x)} \right\rceil \\
&=^2 |\mathcal{H}| + \sum_{\text{internal } x \in \mathcal{H}} \left\lceil \frac{\text{Diam}(x)}{2\text{Norm}(x)} \right\rceil \\
&<^3 |\mathcal{H}| + (n - 1) + \sum_{\text{internal } x \in \mathcal{H}} \frac{\text{Diam}(x)}{2\text{Norm}(x)} \\
&= |\mathcal{H}| + (n - 1) + \frac{1}{2} \sum_{\text{internal } x \in \mathcal{H}} \frac{\text{Diam}(x)}{\text{Norm}(x)} \\
&<^4 |\mathcal{H}| + (n - 1) + \frac{1}{2}4n \\
&\leq 2n - 1 + n - 1 + 2n \\
&< 5n
\end{aligned}$$

- 1: $\text{Norm}(p(x)) \geq 2\text{Norm}(x)$ follows from definition 1.
- 2: $\text{Diam}(x) = 0$ if x is a hierarchy leaf.
- 3: There are at most $n - 1$ internal nodes.
- 4: By lemma 9.

□

If we look at the pseudocode in algorithm 2, then the only operation that is not obviously constant-time (aside from bucketing and handling D -values) is the computation in line 14 of t_x , the starting value for the first bucket. We will now show how to compute these starting values.

Lemma 13 *We can calculate t_x for all $x \in \mathcal{H}$ in $O(n)$ total time.*

Proof: In algorithm 2 we write:

$$t_x \leftarrow \begin{cases} D(x) & \text{if } D(x) + \text{Diam}(x) < b \\ b - \text{Norm}(x) \lceil \frac{b-D(x)}{\text{Norm}(x)} \rceil & \text{otherwise} \end{cases}$$

So, first we need to determine whether $D(x) + \text{Diam}(x) < b$ which is done in $O(1)$ time. Setting t_x to $D(x)$ is of course also $O(1)$. If we need to compute $b - \text{Norm}(x) \lceil \frac{b-D(x)}{\text{Norm}(x)} \rceil$, we can do it by starting in b and then keep subtracting $\text{Norm}(x)$ until the result is $\leq D(x)$. This requires $O(\frac{b-D(x)}{\text{Norm}(x)})$ subtractions, each constant-time. We only perform this computation when $D(x) + \text{Diam}(x) \geq b$, so $\text{Diam}(x) \geq b - D(x)$ which means that the time to compute one t_x is $O(\frac{\text{Diam}(x)}{\text{Norm}(x)})$. The total time is then $O(\sum_{x \in \mathcal{H}} \frac{\text{Diam}(x)}{\text{Norm}(x)})$ which is $O(n)$ by lemma 9. \square

2.7.3 Total Running Time

We now list the pieces that make up the Pettie algorithm as well as their running times, and at the end we tally up the total Pettie running time.

- Construction of the hierarchy (including calculation of *Norms* and *Diams*) takes $O(mn)$ time by lemma 11.
- Running the VISIT procedure (not counting bucketing and handling D -values) takes $O(n)$ time per SSSP computation by lemmas 12 and 13.
- We will show in lemma 15 that handling D -values takes $O(m\alpha(m, n))$ time per SSSP computation which is $O(m + n \log \log n)$ by lemma 35
- We will show in lemma 18 that handling all node bucketing takes $O(mn + \kappa(m + n \log \log n))$ time for κ SSSP computations.

When we put all this together, we arrive at the total running time of $O(mn + n^2 \log \log n)$ for the n SSSP computations that make up the Pettie APSP algorithm. This, along with the correctness that we will show in lemma 33, yields our main theorem.

Theorem 1 *The Pettie algorithm solves the all-pairs shortest paths problem on non-negative, real-weighted, directed graphs in $O(mn + n^2 \log \log n)$ time.*

2.8 Pettie Subroutines and Data Structures

In this section we dive into the specifics of each of the auxiliary algorithms and data structures used by the main Pettie algorithm. We give descriptions of their inner workings, and show their running times. For additional details and formal correctness proofs omitted here, see the referenced works.

2.8.1 Finding Strongly Connected Components

To build the component hierarchy, we need to be able find the strongly connected components of a directed graph. To this end, we use the asymptotically optimal algorithm by Tarjan [37], presented here in a slightly modified form.

In algorithm 3, we assign a color, a component number, and a component “top” to each vertex in the graph. The top will eventually point to the first node that was painted in the node’s SCC. Also, we have two global values: a vertex stack and a component index.

The algorithm performs a depth-first search as many times as necessary, painting the nodes blue and pushing them onto the stack on the way. Initially, each node thinks that it is the top of its component, but upon discovering that it is in the same SCC as an earlier node, this is adjusted.

When we reach a true component top on the way “back” from the recursive DFS, we pop the nodes of the SCC from the stack and label them with a component index.

Algorithm 3 *Tarjan's SCC algorithm*

Input: Graph $G = (V, E, l)$ **Output:** $\forall v, w \in V : \text{Comp}(v) = \text{Comp}(w)$ iff v and w are in the same SCC

```

1: Global Stack  $S \leftarrow \emptyset$ 
2: Global  $index \leftarrow 0$ 
3: for all  $v \in V$  do  $\text{Comp}(v) \leftarrow 0$  ;  $\text{Color}(v) \leftarrow \text{white}$  end for
4: for all  $v \in V$  do if  $\text{Color}(v) = \text{white}$  then  $\text{Paint}(v)$  end if end for
5:
6: procedure  $\text{Paint}(v)$  do
7:    $\text{Top}(v) \leftarrow v$  ;  $\text{Color}(v) \leftarrow \text{blue}$  ; Push  $v$  onto  $S$ 
8:   for all outgoing edges  $(v, w) \in E$  do
9:     if  $\text{Color}(w) = \text{white}$  then  $\text{Paint}(w)$  end if
10:    if  $\text{Comp}(w) = 0$  then  $\text{Top}(v) \leftarrow \text{First}(\text{Top}(v), \text{Top}(w))$  end if
11:  end for
12:  if  $\text{Top}(v) = v$  then
13:     $index \leftarrow index + 1$ 
14:    repeat
15:      pop  $w$  from  $S$  ;  $\text{Comp}(w) \leftarrow index$ 
16:    until  $w = v$ 
17:  end if
18: end procedure

```

Running Time

The initializations in lines 1-3 take time $O(n)$. During the DFS's the vertices are only painted blue by Paint , and once a vertex is blue, Paint is not called on it. Thus, the number of Paint -calls is $O(n)$. By a similar argument, each vertex is only pushed once onto the stack, so in total there are $O(n)$ stack operations.

Each graph edge is only examined once, because Paint is only called once on each vertex, so that contributes $O(m)$ time.

The function First in line 10 returns the node on which Paint was first called. This can be implemented by a simple counter and so returns a result in $O(1)$ time.

If we had wanted to return the SCCs, we could build a list of lists at no extra cost instead of just labeling in line 15.

All in all, we have the following lemma:

Lemma 14 *Given a graph $G = (V, E, l)$ we can identify the strongly connected components of G in time $O(n + m)$ where $n = |V|$ and $m = |E|$.*

2.8.2 Handling D-values: Gabow Decrease-Split-Findmin

During the SSSP computations, we need to handle the tentative distances, the D -values. We require two operations: $\text{SETD}(x, D)$ and $\text{GETD}(x)$ where x is a hierarchy node and D is a real number. SETD is used in edge relaxations, and GETD is used in VISIT for bucketing purposes. To implement these operations, we employ a decrease-split-findmin data structure invented by Gabow [16].

Given an list of elements $\langle x_1, \dots, x_n \rangle$ each having a real-valued key, the Gabow structure \mathcal{DSF} supports the following operations:

$\text{INITIALIZE}()$:	initializes the structure
$\text{DECREASEKEY}(x, k)$:	sets $\text{key}(x) \leftarrow \min\{\text{key}(x), k\}$
$\text{SPLIT}(x_i)$:	splits the list $\langle \dots, x_i, \dots \rangle$ in two: $\langle \dots, x_i \rangle$ and $\langle x_{i+1}, \dots \rangle$
$\text{FINDMIN}(x)$:	returns the minimum key present in x 's list

We use \mathcal{DSF} as follows:

Just after the hierarchy has been built, we perform a single traversal of \mathcal{H} to store in each hierarchy node x a pointer $\text{Leaf}(x)$ to the rightmost leaf in its subtree. This takes time $O(|\mathcal{H}|)$ which is obviously dominated by the time to construct \mathcal{H} , and thus, asymptotically speaking, does not add to the total running time.

Then, at the beginning of each SSSP calculation, we initialize \mathcal{DSF} with the leaves of the hierarchy as elements and their D -values as keys.

During the VISIT -recursion, the first time we visit a hierarchy node x , we call $\text{SPLIT}(\text{Leaf}(y))$ for each $y \in \text{Children}(x)$. This ensures that the \mathcal{DSF} -lists always reflect which hierarchy nodes have been visited. More precisely, each list in \mathcal{DSF} corresponds to an inactive hierarchy node whose parent is active. These inactive nodes are exactly the ones we store in buckets, and therefore the ones on which we need to call GETD .

Because the D -value of a hierarchy node is the minimum D -value of all its descendants, $\text{GETD}(x)$ is now simply a call to $\text{FINDMIN}(\text{Leaf}(x))$. And $\text{SETD}(x, D)$ is implemented by calling $\text{DECREASEKEY}(x, D)$ ³.

Gabow's \mathcal{DSF} implementation (sketched in the section ahead) provides the following time bounds: INITIALIZE takes $O(n)$ time, and a sequence of $O(M)$ FINDMINS , SPLITS and DECREASEKEYS can be done in $O(M\alpha(M, n))$ time.

When an internal hierarchy node is first VISITED , we call a FINDMIN to find its D -value (at that point, we know it will never change). Also, we call one SPLIT and one FINDMIN for every internal node (except the root node) when its parent is first VISITED . So by lemma 7, these operations are called $O(n)$ times. Each edge in the input graph can at most cause one DECREASEKEY , so that is $O(m)$ DECREASEKEYS total. This means that M is $O(m)$, so we have arrived at the total time bound for maintaining D -values:

Lemma 15 *Maintaining D -values for all nodes in \mathcal{H} during one SSSP computation takes $O(m\alpha(m, n))$ time.*

\mathcal{DSF} Implementation

In this section, we sketch the implementation of the Gabow \mathcal{DSF} structure. For more details, see [16].

The \mathcal{DSF} works on Gabow-lists. To create a Gabow-list L , imagine that we have all of the hierarchy leaves in a linked list. We call these the *elements*. Now we choose some arbitrary element z and call the elements from the beginning and up to z the *head*, and the elements after z the *tail*. What we want to do is assign a *super-element* to each element, such that every super-element presides over an ‘‘Ackermann number’’ of consecutive elements while enforcing that a super-element cannot preside over both head and tail elements.

So we set $i = \alpha(M, n)$, and look at the i^{th} row of the Ackermann matrix from appendix C. In that row we find the largest entry C smaller than the number of elements in the head. Now we create a super-element e that

³Remember that D -values never increase during the VISIT recursion.

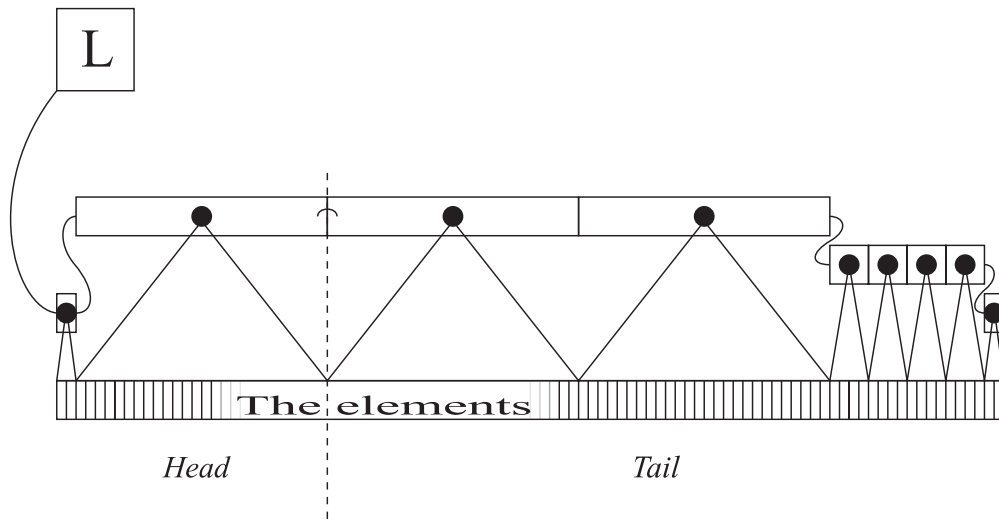


Figure 2.7: An example of a Gabow DSF structure. The head is presided over by two super-elements, both singletons. The tail is presided over by seven super-elements grouped into two sublists of size 2 and 4 plus a singleton. In the figure, the pointers from elements to super-elements and from sublists and singletons to L have been omitted for clarity.

presides over the C nodes from z and to the left, and let each of these elements have a pointer up to e . We then repeat the process, only this time the C has to be smaller than the number of now remaining elements in head. This continues until there are no more elements in the head, and then we perform the symmetrical equivalent on the tail.

With these super-elements created, we start grouping them into what we will call *sublists*. If two adjacent super-elements preside over the same number of elements, they belong in the same sublist. If a super-element does not share its C with any of its neighbors, we call it a *singleton* super-element. Finally we let each sublist and each singleton have a pointer to L , the Gabow-list object, and we let L keep all the sublists and singletons in a linked list. Now we have the situation depicted in figure 2.7.

Notice the pointers that are *not* allowed. Neither elements nor super-elements in sublists have pointers to or from L .

Each element has a key, (for leaves: the D -value). Each super-element maintains the minimum key of the elements over which it presides. L

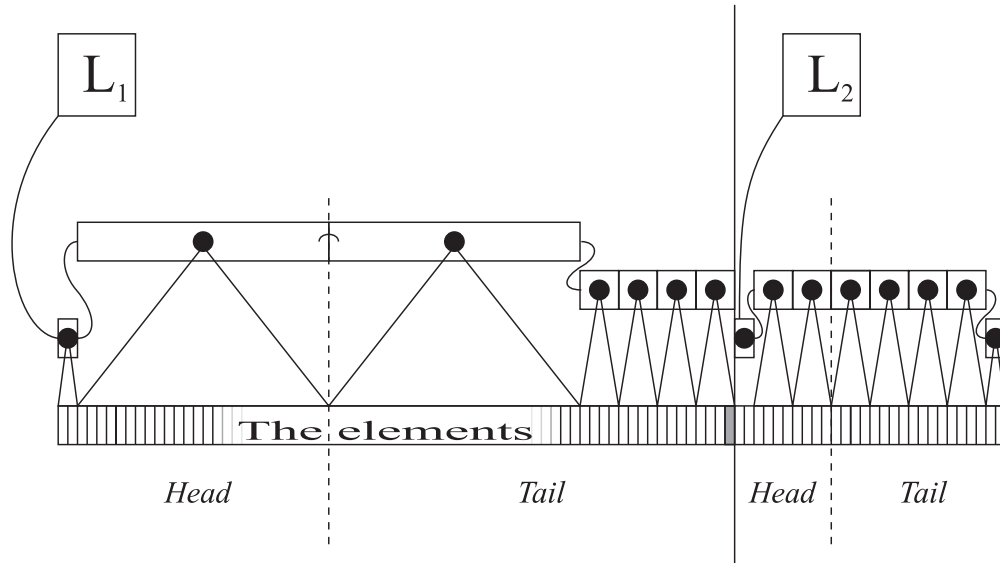


Figure 2.8: Given the structure in figure 2.7, we perform a SPLIT on the grayed-out element to form the new lists L_1 and L_2 from L . As in figure 2.7, the “upwards” pointers are omitted for clarity.

maintains the minimum key over all the elements.

Now we start to address the operations: INITIALIZE, DECREASEKEY, SPLIT, and FINDMIN. What we have described above is essentially the INITIALIZE operation. Except – and here is the trick – we regard the sublists as Gabow-lists so we call recursively on them. We start by executing the operation on the outermost Gabow-list that consists of all the elements and has $i = \alpha(M, n)$, and then we call recursively with $i - 1$ as the new i . We call the initialization of the head and tail INITHEAD and INITTAIL, respectively.

The remaining operations are also recursive. Let us consider DECREASEKEY(x, k). If k is the new global minimum key and x 's super-element is part of a sublist, then it seems we have a problem. The element x has no (direct or indirect) pointer to L , so it cannot update L 's key. We fix this by letting the recursive call to DECREASEKEY(x, k) return the list containing x . Then we just call recursively on super-elements until we hit a singleton, which has a pointer to its elements' L . When the recursive call returns, we can set the key of both element, super-element

and Gabow-list correctly.

We implement $\text{FINDMIN}(x)$ by calling $\text{DECREASEKEY}(x, \infty)$ to obtain the list L containing x and return L 's key.

To perform $\text{SPLIT}(x)$, we first call recursively on x 's super-element e , and then call SPLIT recursively on e 's left neighbor (if present) so that e becomes a singleton. Then we consider the elements that have e as super-element. We call INITTAIL on those elements from x and to the left, and INITHEAD on those to the right of x .⁴ See figure 2.8 for an illustration of SPLIT .

So the SPLITs determine which elements are in the head of their list and which are in the tail. Only at the first initialization can we choose the boundary between them arbitrarily. This concludes the description of Gabow's \mathcal{DSF} structure.

2.8.3 The Bucket Structure

Now we move on to something different, namely the "buckets" mentioned in the section about the VISIT algorithm. We assign to each internal node in the component hierarchy a set of numbered buckets. This is called the node's bucket structure. A node x has a bucket structure consisting of $\lceil \frac{\text{Diam}(x)}{\text{Norm}(x)} \rceil + 1$ buckets. Each bucket covers a specific real value interval.

At any given moment, a bucket in x 's bucket structure contains zero or more child nodes of x . When it is time for a child node to be bucketed, it is put into the bucket that covers its D -value. Should this D -value change, the child node may need to be rebucketed accordingly.

First, we look at the necessary operations on the bucket structure, and then we discuss the data structures needed.

Operations

Looking at the VISIT pseudocode in algorithm 2, it is clear that we need the following bucket operations:

⁴Note that in Gabow's article [16], the calls to INITHEAD and INITTAIL in SPLIT are erroneously interchanged.

INIT()	:	Creates and initializes the buckets. Is called once and for all for each node when \mathcal{H} is built.
INSERT(<i>child</i> , <i>D</i>)	:	Inserts the child into the bucket covering <i>D</i> .
UPDATE(<i>child</i> , <i>newD</i>)	:	Ensures that the child is bucketed in accordance with the <i>newD</i> -value. Rebuckets if necessary.
BUMP(<i>child</i>)	:	Moves the child from the front bucket into the following bucket.
ISEMPTY()	:	Returns whether the front bucket is empty.
GETLEFTMOST()	:	Returns the leftmost child in the front bucket.
POPLEFTMOST()	:	Removes the leftmost child from the front bucket.

The “front bucket” is the one currently considered by the VISIT procedure. Whenever a front bucket is discovered to be empty, the following bucket becomes the front bucket.

Implementation

We can implement each bucket as a doubly-linked list of child nodes. INSERT then simply appends the child to the correct list. If UPDATE needs to rebucket, the node is unhooked from its current list and appended to the new list.

The front bucket, however, is not a linked list. We can implement it using a van Emde Boas (vEB) heap [39]⁵. The vEB-heap works on integers, so we do not use it to store the actual child nodes, but rather their index in their parent’s child array. This way we can convert to and from the vEB-integers in constant time and the smallest vEB-integer represents the leftmost child in the bucket.

When a bucket becomes the front bucket, we insert the indices corresponding to all the bucket’s nodes into the vEB-heap. GETLEFTMOST is done by retrieving the minimum index from the vEB-heap. POPLLEFTMOST calls GETLEFTMOST, then deletes the result from the

⁵The van Emde Boas heap keeps its usual running time even in our restricted computational model without integer division or bit-shifting, see [26]

vEB-heap. And BUMP calls POPLLEFTMOST, then inserts the corresponding node into the bucket following the front bucket.

All indices are of course less than $Deg(x)$, so in the vEB-heap, insertions and deletions take $O(\log \log Deg(x))$ time.

By implementing the bucket structure in this straightforward manner, we can achieve the following amortized time bounds for the operations above:

INIT()	:	$O(\frac{Diam(x)}{Norm(x)})$
INSERT(<i>child</i> , <i>D</i>)	:	$O(\log \frac{Diam(x)}{Norm(x)})$
UPDATE(<i>child</i> , <i>newD</i>)	:	$O(1)$ or $O(\log \frac{Diam(x)}{Norm(x)})$
BUMP(<i>child</i>)	:	$O(\log \log Deg(x))$
ISEMPTY(<i>bucketNo</i>)	:	$O(1)$
GETLEFTMOST(<i>bucketNo</i>)	:	$O(1)$
POPLLEFTMOST(<i>bucketNo</i>)	:	$O(\log \log Deg(x))$

The time to UPDATE depends on whether a rebucketing is needed. The $\log \frac{Diam(x)}{Norm(x)}$ -times reflect the floor-dividing⁶ needed to calculate the appropriate bucket number given the *D*-value.

Lazy Bucketing

Unfortunately, the time bound for UPDATE achieved above will turn out to be too slow for us. So to improve it, we use a lazy bucketing scheme from [32]. This scheme exploits the fact that we do not need to keep all the nodes in their correct bucket at all times. The VISIT procedure really just requires that the *front* bucket contains the nodes that belong there. The rest of the buckets can behave in any way they want – until they become the front bucket.

Lazy bucketing uses “super-buckets” that each do the job of several regular buckets. The further a regular bucket is from becoming the front bucket, the larger the super-bucket that represents it. In this way, much fewer buckets are needed, and the following amortized time bounds are achieved:

⁶See appendix A.

INIT()	:	$O(\log \frac{Diam(x)}{Norm(x)})$
INSERT(<i>child</i> , <i>D</i>)	:	$O(\log \frac{Diam(x)}{Norm(x)})$
UPDATE(<i>child</i> , <i>newD</i>)	:	$O(1)$
BUMP(<i>child</i>)	:	$O(\log \log Deg(x))$
ISEMPTY(<i>bucketNo</i>)	:	$O(1)$
GETLEFTMOST(<i>bucketNo</i>)	:	$O(1)$
POPLEFTMOST(<i>bucketNo</i>)	:	$O(\log \log Deg(x))$

In the following section, we explain how the lazy bucketing achieves these running times. Later, we bound the total bucketing time spent by the Pettie algorithm.

Lazy Bucket Implementation

To implement lazy bucketing, we use an array of super-buckets to represent the actual logical buckets. The super-bucket at index i in the array represents 2^i logical buckets and is said to be at *level* i . Though super, the buckets are still just implemented with a linked list.

Each super-bucket at level i for hierarchy node x is assigned a real interval $[a_i, b_i)$ where $b_i = a_i + 2^i \cdot Norm(x)$ and a_i is equal to either a_{i-1} or b_{i-1} . Initially, all a_i are equal to t_x , the lowest value covered by x 's first logical bucket.

Because we double their size with each new level, we need $\lceil \log \frac{Diam(x)}{Norm(x)} \rceil$ such super-buckets to cover x 's bucketing interval. This accounts for the INIT time.

When we INSERT a node, we just put it in the level 0 super-bucket, no matter which logical bucket it was supposed to go into. Along with the node, we keep its D -value.

We now introduce an operation on logical buckets called *close*. When a bucket is closed, all the nodes in the level 0 super-bucket who have $D \in [a_0, b_0)$ are moved to the front bucket. As usual, the front bucket is implemented as a vEB-heap. The nodes in the level 0 super-bucket that did not really belong in that interval, will be dealt with shortly. But first, all super-buckets that cover the closed bucket move one step to the right. In other words, if $a_i = a_0$ then add $2^i \cdot Norm(x)$ to both a_i and b_i for $i \geq 0$. Then by construction, the new a and b values satisfy $b_{i-1} = a_i$. So, the

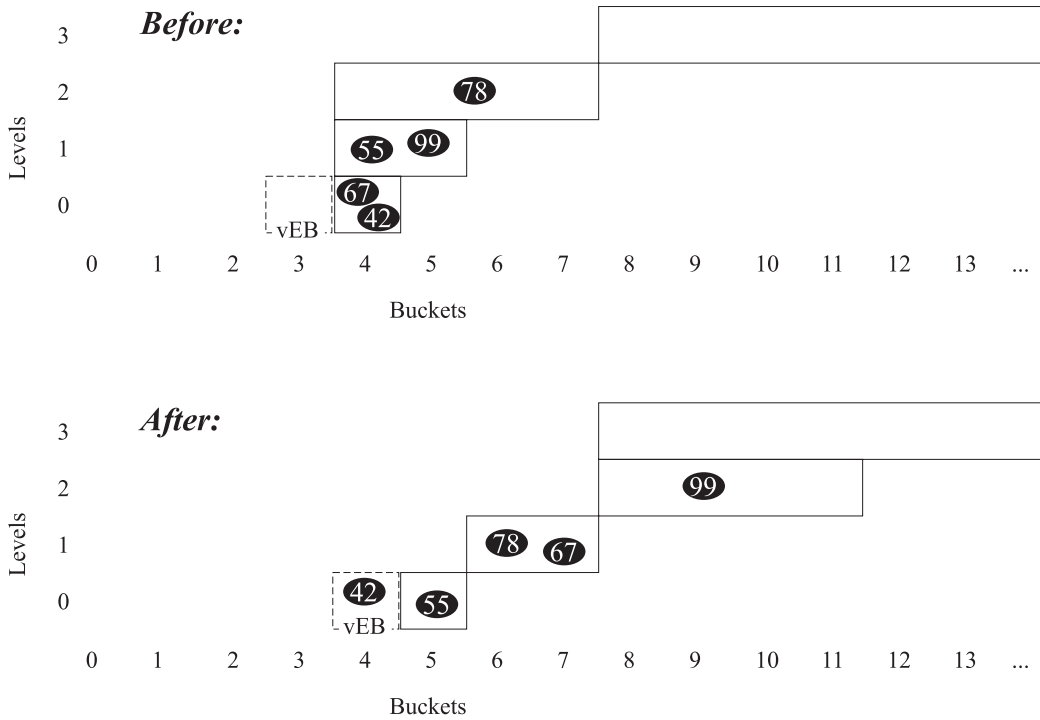


Figure 2.9: Example of a close on bucket 4. Here, $Norm$ is 10, and the start value t_x is 0. So e.g. logical bucket 7 covers $[70, 80)$. The boxes are super-buckets and the ovals are nodes displaying their D -value.

super-buckets that were just moved, now represent consecutive intervals. We then determine for each node in a moved level i super-bucket which of the neighboring intervals $[a_{i-1}, b_{i-1})$, $[a_i, b_i)$, and $[a_{i+1}, b_{i+1})$ suits its D -value best, and place the node in that super-bucket. So the node either moves down a level, stays on the same level or moves up a level. An example of *close* can be seen in figure 2.9.

When the VISIT algorithm comes around to collect the nodes of the first bucket, we just *close* that bucket. The subsequent calls to ISEMPY, GETLEFTMOST, and POPLLEFTMOST only deal with the vEB-heap, so their running times are not affected by this lazy scheme.

If a node y is at level i and we want to UPDATE its D -value, we check whether $newD < b_i$. If not, we do nothing other than set $D = newD$. If it is, then we bubble y down to the lowest level j such that $newD \in [a_j, b_j)$. This requires $i - j + 1$ comparisons. See an example of UPDATE in

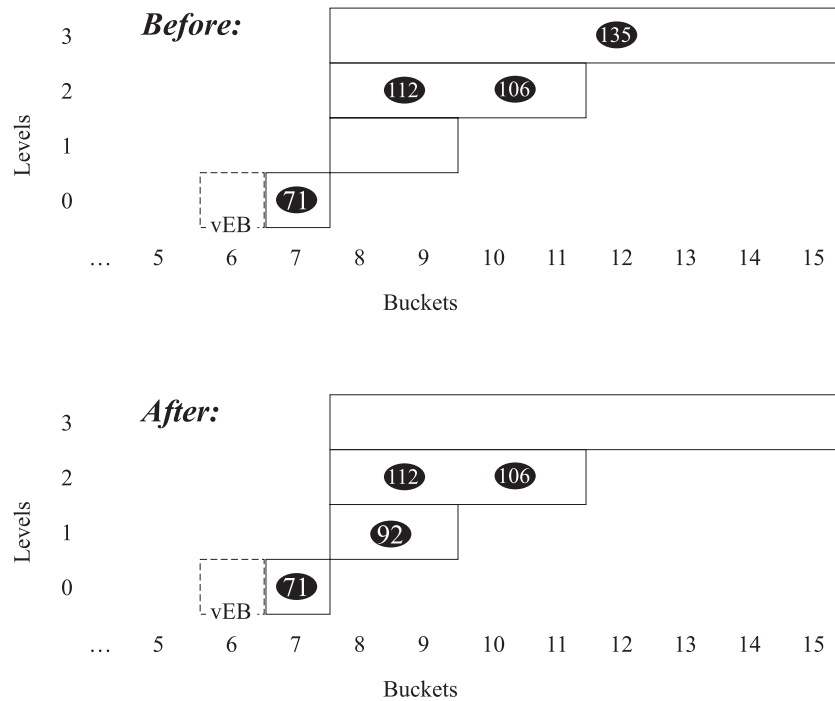


Figure 2.10: Example of an UPDATE. As in figure 2.9, $Norm = 10$, $t_x = 0$. The node on level 3 changes its D -value from 135 to 92 and is bubbled down to level 1. If instead it had changed its D -value to 122, nothing would have happened.

figure 2.10.

We can implement BUMP by extracting the node from the vEB-heap and then put it on a waiting list. When the vEB-heap is about to represent the next front bucket, nodes on the waiting list are reinserted. One extraction plus one insertion is still $O(\log \log Deg(x))$.

Looking at the running times we are supposed to achieve, this lazy bucketing scheme may seem odd. For instance, how can UPDATE be constant-time when it must perform bubbling? This is because the running times are amortized. We charge the entire rebucketing costs to INSERT – even though it hardly does any work.

By construction of the scheme, a node y can either move to “the right”, meaning to a super-bucket spanning an interval with larger a -value, or it can bubble down into super-buckets with equal a -value. If it moves to the

right, it either goes up one level, stays on the same level or goes down a level. If it stays on the same level i , we know that $D(y) \in [a_i, b_i)$, so it will never again move to the right. If it goes down a level to $i - 1$, we know that $D(y) \in [a_{i-1}, b_{i-1})$, so it will never again move to the right. If it bubbles down to level j we know that $D(y) \in [a_j, b_j)$, so once again it will never move to the right.

This means that there is a number of (constant-time) operations where the node moves up, then at most one operation where the level is unchanged, and finally a number of operations where the node moves down. As mentioned before, there are only $\left\lceil \log \frac{Diam(x)}{Norm(x)} \right\rceil$ super-buckets, so a node will never go to a higher level than that. Therefore, if we charge INSERT $2 \cdot \left\lceil \log \frac{Diam(x)}{Norm(x)} \right\rceil + 1$ dollars and hand them over to the inserted node, then we can charge the node 1 dollar for each operation and thereby pay for the entire rebucketing. The only thing unpaid for is then the comparison between $newD$ and b_i in UPDATE, which is constant-time.

This concludes the description of the inner workings of the bucketing scheme. We now zoom out a bit and measure the time spent on bucketing given the bounds that we have just shown.

Bucketing Time

The bucket life of a hierarchy node is as follows: When its parent is first visited, the node is bucketed. Its D-value is then decreased a number of times, sometimes causing a rebucketing. At some point, we call VISIT on it, and afterwards we may need to BUMP the node to the next bucket. This VISIT-and-BUMP routine may happen a number of times, and finally, when no more rebucketing is needed, we remove the node as the leftmost child of the front bucket.

Tallying up, we get that each child node is responsible for one INSERT, zero or more UPDATES, zero or more BUMPS, and one POPLEFTMOST. Also, as many IEMPTYs and GETLEFTMOSTs as there are BUMPS (see algorithm 2 lines 21-30).

The total number of BUMPS for all nodes is $O(n)$, because it must be less than the total number of VISIT calls⁷, which is $O(n)$ by Lemma 12. So

⁷Every BUMP is followed by a VISIT, but not all VISITs follow a BUMP.

total BUMP time is $O(n \log \log n)$, including GETLEFTMOST and ISEMPY.

The total UPDATE time is $O(m)$, because each graph edge causes at most one UPDATE, so summed over all nodes there are at most m UPDATES, each constant-time.

The total POPLEFTMOST time is

$$O\left(\sum_{x \neq \text{Root}} \log \log \text{Deg}(p(x))\right) = O\left(\sum_{x \in \mathcal{H}} \text{Deg}(x) \log \log \text{Deg}(x)\right)$$

which is $O(n \log \log n)$ because there are $O(n)$ nodes in the hierarchy (lemma 7).

The total INSERT time is

$$O\left(\sum_{x \neq \text{Root}} \log \frac{\text{Diam}(p(x))}{\text{Norm}(p(x))}\right) = O\left(\sum_{x \in \mathcal{H}} \text{Deg}(x) \log \frac{\text{Diam}(x)}{\text{Norm}(x)}\right)$$

The total initialization time is

$$O\left(\sum_{x \in \mathcal{H}} \log \frac{\text{Diam}(x)}{\text{Norm}(x)}\right)$$

which is $O(n)$ by lemma 9.

This brings us to the total bucketing time:

$$O\left(m + n \log \log n + \sum_{x \in \mathcal{H}} \left(\text{Deg}(x) \log \frac{\text{Diam}(x)}{\text{Norm}(x)}\right)\right)$$

This entire argument also works if we do not sum over all of the nodes in \mathcal{H} , but just a subset thereof, so we get the following lemma.

Lemma 16 *For $X \subseteq \mathcal{H}$, the total bucketing time for one SSSP computation is*

$$O\left(m + n \log \log n + \sum_{x \in X} \left(\text{Deg}(x) \log \frac{\text{Diam}(x)}{\text{Norm}(x)}\right)\right)$$

This was not the time bound we were looking for, but all is not lost. If we let X be the nodes x that satisfy $\frac{Diam(x)}{Norm(x)} < (\log n)^{O(1)}$ then the total bucketing time for those nodes clearly becomes $O(m + n \log \log n)$, our goal.

But what about the nodes not included in X ? The problem is that their large diameter translates into an awful lot of buckets – too many for the lazy bucketing to handle in the desired time. Therefore, we will give these large-diameter nodes a special treatment so that the bucketing of their child nodes becomes more precise. This treatment is described in the following section.

2.8.4 The Oracle Function

When the VISIT procedure relaxes edges, hierarchy nodes are gradually moved to lower numbered buckets in their parent’s bucket structure. What we will do is precompute an oracle function that is able to tell us approximately which bucket a node will eventually end up in. This will save us a lot of rebucketing, because given the oracle, we can simply defer the bucketing of a node until it approaches its final destination. This means that we weaken invariant 5 to say that if y is a child of x , then y is not bucketed if we know that its D -value will decrease in the future.

The trade-off is between the time spent computing the oracle function and the time saved on rebucketing. It turns out that we get the time bounds we want if we compute the oracle function for all nodes x that satisfy $\frac{Diam(x)}{Norm(x)} \geq \log n$.

To define the oracle function, we first introduce the concept of *relative distances*. In loose terms, it is a measure of how deeply a vertex is buried in a subgraph seen from the perspective of the source vertex. A hierarchy node that represents graph vertices at a short relative distance from the source will eventually end up in a low-numbered bucket in its parent’s bucket structure. Similarly, a long relative distance means the node will never get below the high-numbered buckets.

We define the distance from $u \in V$ to $y \in Children(x)$ relative to $x \in \mathcal{H}$ as:

$$\Delta_x(u, y) = d(u, y) - d(u, x)$$

We want the oracle function to return relative distances, but to speed up the computation, we make the function return a normalized integer

approximation $\widehat{\Delta}$ to the relative distance:

$$\begin{aligned} \widehat{\Delta}_x(u, y) &= \text{some } i \in \mathbb{Z} \text{ such that } |\epsilon_x \cdot i - \Delta_x(u, y)| < \epsilon_x \\ \text{where } \epsilon_x &= \frac{\text{Norm}(x)}{2} \end{aligned}$$

It follows from the definition that $\widehat{\Delta}_x(u, v)$ will always be either $\left\lfloor \frac{\Delta_x(u, v)}{\epsilon_x} \right\rfloor$ or $\left\lceil \frac{\Delta_x(u, v)}{\epsilon_x} \right\rceil$.

Using the Oracle

Suppose we are given the oracle function, meaning all $\widehat{\Delta}_x(\cdot, \cdot)$ -values. How do we use it?

When we bucket a child node y of x by its D -value, it goes into bucket number $\left\lfloor \frac{D(y) - t_x}{\text{Norm}(x)} \right\rfloor$. But if we assume that the D -value has reached d , then it will never decrease during the rest of the SSSP computation. That means that y 's final bucket number is:

$$\begin{aligned} \left\lfloor \frac{D(y) - t_x}{\text{Norm}(x)} \right\rfloor &\stackrel{1}{\in} \left\{ \left\lfloor \frac{D(y) - D(x)}{\text{Norm}(x)} \right\rfloor, \left\lfloor \frac{D(y) - D(x)}{\text{Norm}(x)} \right\rfloor + 1 \right\} \\ &\stackrel{2}{=} \left\{ \left\lfloor \frac{D(y) - d(s, x)}{\text{Norm}(x)} \right\rfloor, \left\lfloor \frac{D(y) - d(s, x)}{\text{Norm}(x)} \right\rfloor + 1 \right\} \\ &\stackrel{3}{=} \left\{ \left\lfloor \frac{d(s, y) - d(s, x)}{\text{Norm}(x)} \right\rfloor, \left\lfloor \frac{d(s, y) - d(s, x)}{\text{Norm}(x)} \right\rfloor + 1 \right\} \\ &= \left\{ \left\lfloor \frac{\Delta_x(s, y)}{\text{Norm}(x)} \right\rfloor, \left\lfloor \frac{\Delta_x(s, y)}{\text{Norm}(x)} \right\rfloor + 1 \right\} \\ &\stackrel{4}{=} \left\{ \left\lfloor \frac{\epsilon_x \widehat{\Delta}_x(s, y) - \epsilon_x}{\text{Norm}(x)} \right\rfloor, \dots, \left\lfloor \frac{\epsilon_x \widehat{\Delta}_x(s, y) + \epsilon_x}{\text{Norm}(x)} \right\rfloor + 1 \right\} \\ &= \left\{ \left\lfloor \frac{\widehat{\Delta}_x(s, y) - 1}{2} \right\rfloor, \dots, \left\lfloor \frac{\widehat{\Delta}_x(s, y) + 1}{2} \right\rfloor + 1 \right\} \\ &= \left\{ \left\lfloor \frac{\widehat{\Delta}_x(s, y) - 1}{2} \right\rfloor, \dots, \left\lfloor \frac{\widehat{\Delta}_x(s, y) + 3}{2} \right\rfloor \right\} \end{aligned}$$

1: By definition of t_x , $D(x) - t_x < \text{Norm}(x)$.

2: By lemma 30.

3: By assumption, $D(y) = d(s, y)$.

4: By definition of $\widehat{\Delta}_x$, $|\Delta_x(s, y) - \epsilon_x \widehat{\Delta}_x(s, y)| < \epsilon_x$.

So given the oracle function and a node, we can identify three buckets and guarantee that one of them will be the node's final bucket. During an SSSP computation, every time the node's D -value is decreased, we check whether the new D -value is covered by one of the three buckets. If so, we bucket the node, and if not, we do nothing. This means that both INSERT and UPDATE are now constant-time. And since we perform $O(n)$ insertions, $O(m)$ updates, and $O(n)$ front bucket operations, we get the following lemma:

Lemma 17 *Given the $\widehat{\Delta}_x(s, y)$ -values, the total bucketing time for all y in the SSSP computation from s is $O(m + n \log \log n)$.*

Total Bucketing Time

In the section to come, we will show how to compute the $\widehat{\Delta}_x$ -values. That section will end up with lemma 20 showing that we can perform the calculations for all $x \in \mathcal{H}$ where $\frac{\text{Diam}(x)}{\text{Norm}(x)} \geq \log n$ in $O(mn)$ time.

To sum up the bucketing of the Pettie algorithm: We use the lazy bucketing to handle the low-diameter nodes in $O(m + n \log \log n)$ time by lemma 16. And using $\widehat{\Delta}$ -assisted bucketing for high-diameter nodes, the combination of lemmas 17 and 20 tells us that there is a one-time cost of $O(mn)$ followed by $O(m + n \log \log n)$ per SSSP. So we have arrived at the total bucketing time for the Pettie algorithm:

Lemma 18 *The total time spent by the Pettie algorithm on bucketing in κ SSSP computations is $O(mn + \kappa(m + n \log \log n))$.*

Computing the Oracle

In this section, we will show how to compute all $\widehat{\Delta}_x(\cdot, \cdot)$ -values for a fixed $x \in \mathcal{H}$. The method is this: First we produce a graph of approximate relative distances, then we find shortest paths in the graph. The lengths of those shortest paths will turn out to work as our $\widehat{\Delta}_x$ -values.

Now it may look as if we are going in circles: Pettie's shortest paths algorithm requires bucketing, and bucketing requires the finding of shortest

paths... However, the graph we construct here has integer weights, and we only compute shortest paths for selected pairs in it, so we do not spend too much time on this phase.

We define two new weight functions, $\delta_x : E \rightarrow \mathbb{R}_+$ and $\widehat{\delta}_x : E \rightarrow \mathbb{N}$ as:

$$\delta_x(u, v) = l(u, v) + d(v, x) - d(u, x)$$

$$\widehat{\delta}_x(u, v) = \begin{cases} \left\lfloor \frac{\delta_x(u, v)}{\epsilon'_x} \right\rfloor & \text{if } \delta_x(u, v) \leqslant \text{Diam}(x) \\ \infty & \text{otherwise} \end{cases}$$

$$\text{where } \epsilon'_x = \frac{\epsilon_x}{n} = \frac{\text{Norm}(x)}{2n}$$

If δ is one of these new weight functions then we let $G^\delta = (V, E, \delta)$ be the input graph with δ substituted for the usual l . Similarly, we let d^δ be the distance function for G^δ .

With these definitions in place, the algorithm is as follows:

1. Construct $G^{\widehat{\delta}_x}$.
2. Find $d^{\widehat{\delta}_x}(v, y)$ for all $v \in V, y \in \text{Children}(x)$.
3. Set $\widehat{\Delta}_x(v, y) = \left\lceil \frac{d^{\widehat{\delta}_x}(v, y)}{n} \right\rceil$.

Later (section 2.9.1) we prove the correctness of this scheme, but for now we provide implementation details and argue running times.

Step 1: By definition of $\widehat{\delta}_x$, constructing $G^{\widehat{\delta}_x}$ means we have to calculate all $d(\cdot, x)$ -values in G . We do this by first reversing all edges in G and contracting C_x into a single vertex x in $O(m + n)$ time. Then we run Dijkstra SSSP using x as the source vertex, spending $O(m + n \log n)$ time. The $d(x, \cdot)$ -values produced by Dijkstra in the reversed graph are then exactly the $d(\cdot, x)$ -values we need. Now, we calculate δ_x according to its definition for each edge, and for those that are below $\text{Diam}(x)$, we find the $\widehat{\delta}_x$ -values by floor-dividing by ϵ'_x . By Appendix A, this division takes time $O(\log \frac{\text{Diam}(x)}{\epsilon'_x})$ which is $O(\log(2n \frac{\text{Diam}(x)}{\text{Norm}(x)})) = O(\log n)$ by lemma 8. We

divide at most once per edge, so the total time to build $G^{\widehat{\delta}_x}$ is $O((m+n) + (m+n \log n) + m \log n) = O(m \log n)$.

Step 2: In step 2, we must calculate $d^{\widehat{\delta}_x}(v, y)$ for all $v \in V$, $y \in \text{Children}(x)$. In other words, we must find the shortest path from any vertex in $G^{\widehat{\delta}_x}$ to each of the graphs representing x 's children. To do this, we reverse all edges in $G^{\widehat{\delta}_x}$ and run as many Dijkstra computations as there are children. Because all weights in $G^{\widehat{\delta}_x}$ are integer, we can use Dial's version [8] of the Dijkstra algorithm. This means that we use an array of buckets as the priority queue, using the array index as D -value. Now we can perform all the Dijkstra computations simultaneously in just one array. A pair (y, v) is then present in Dial-bucket k iff $D(y, v) = k$. Edge relaxations simply consist of moving a pair to a new bucket. Dial's algorithm is a left-to-right scan of the bucket array, performing relaxations as pairs are scanned. We get the algorithm started by loading bucket 0 with all pairs (y, v) where $y \in \text{Children}(x)$, $v \in V_y$. The time spent is $O(m)$ on edge-reversal, $O(n \text{Deg}(x))$ on initialization, $O(m \text{Deg}(x))$ on relaxation and $O(\#(\text{Dial-buckets}))$ scanning the buckets. We will prove in lemma 24 that $\#(\text{Dial-buckets})$ is less than $2n \frac{\text{Diam}(x)}{\text{Norm}(x)}$, so all in all step 2 takes $O(m \text{Deg}(x) + n \frac{\text{Diam}(x)}{\text{Norm}(x)})$ time.

Step 3: Step 3 consists of integer division by n and rounding up. Unfortunately, our computational model does not include integer division, so we need something else. In step 2, before we start scanning, we assign a special index to each bucket in the array. Using two counters, we can assign special index 1 to the first n buckets, then special index 2 to the next n buckets, etc. This means a bucket k receives special index $\lceil \frac{k}{n} \rceil$. So when the Dial scan reaches the pair (y, v) in bucket k , we set $\widehat{\Delta}_x(v, y) = k$'s special index. Step 3 takes constant time per bucket, so $O(n \frac{\text{Diam}(x)}{\text{Norm}(x)})$ total time.

With the time bounds from steps 1 to 3, we have arrived at the following lemma:

Lemma 19 *For an $x \in \mathcal{H}$ we can compute $\widehat{\Delta}_x$ in $O(m \log n + m \text{Deg}(x) + n \frac{\text{Diam}(x)}{\text{Norm}(x)})$ time.*

Now we can prove the total time bound for the computation of the oracle function for all large-diameter nodes:

Lemma 20 *We can compute $\widehat{\Delta}_x$ for all $x \in \mathcal{H}$ where $\frac{Diam(x)}{Norm(x)} \geq \log n$ in $O(mn)$ time.*

Proof: Let $T(m, n, \beta)$ be the time necessary to compute $\widehat{\Delta}_x$ for all $x \in \mathcal{H}$ where $\frac{Diam(x)}{Norm(x)} \geq \beta$. Then we have:

$$\begin{aligned}
T(m, n, \beta) &=^1 \sum_{x : \frac{Diam(x)}{Norm(x)} \geq \beta} O(m \log n + m Deg(x) + n \frac{Diam(x)}{Norm(x)}) \\
&=^2 O(4 \frac{n}{\beta} m \log n) + \sum_{x : \frac{Diam(x)}{Norm(x)} \geq \beta} O(m Deg(x) + n \frac{Diam(x)}{Norm(x)}) \\
&=^3 O(4 \frac{n}{\beta} m \log n + 2mn) + \sum_{x : \frac{Diam(x)}{Norm(x)} \geq \beta} O(n \frac{Diam(x)}{Norm(x)}) \\
&=^4 O(4 \frac{n}{\beta} m \log n + 2mn + 4n^2) \\
&= O\left(\left\lceil \frac{\log n}{\beta} \right\rceil mn\right)
\end{aligned}$$

1: By lemma 19

2: By corollary 10

3: By lemma 7

4: By lemma 9

To achieve a total time of $O(mn)$, we set $\beta = \log n$. □

Actually, by lemma 35 in appendix C, we could still achieve the desired running time of $O(mn + n^2 \log \log n)$ for the entire Pettie APSP algorithm by setting β a bit lower at $\frac{\log n}{\alpha(m, n)}$, but for simplicity and MSSP performance we choose $\log n$.

2.9 Pettie Correctness

Up to this point, we have really only proved one thing: The Pettie algorithm runs fast. In this section we prove that it actually works. We first show the correctness of the algorithm in section 2.8.4 that calculates the oracle function, and we finish by proving that the VISIT procedure does what it claims to do.

2.9.1 Correctness of the Oracle

To show that we compute the $\widehat{\Delta}$ -values correctly, we need some helper lemmas. First we prove that the distance function d^{δ_x} is indeed the same as the exact relative distance measure Δ_x .

Lemma 21 *Let $x \in \mathcal{H}$, $y \in \text{Children}(x)$, and $u \in V$. Then $\Delta_x(u, y) = d^{\delta_x}(u, y)$.*

Proof: By $u_1 \rightsquigarrow u_j$ we will denote that there exists a path $\langle (u_1, u_2), \dots, (u_{j-1}, u_j) \rangle$ from u_1 to u_j . Then

$$\begin{aligned}
d^{\delta_x}(u, y) &=^1 \min \left\{ \sum_{i=1}^{j-1} \delta_x(u_i, u_{i+1}) \mid u = u_1 \rightsquigarrow u_j \in V(C_y) \right\} \\
&=^2 \min \left\{ \sum_{i=1}^{j-1} l(u_i, u_{i+1}) + d(u_{i+1}, x) - d(u_i, x) \mid u = u_1 \rightsquigarrow u_j \in V(C_y) \right\} \\
&=^3 \min \left\{ d(u_j, x) - d(u_1, x) + \sum_{i=1}^{j-1} l(u_i, u_{i+1}) \mid u = u_1 \rightsquigarrow u_j \in V(C_y) \right\} \\
&=^4 \min \left\{ -d(u_1, x) + \sum_{i=1}^{j-1} l(u_i, u_{i+1}) \mid u = u_1 \rightsquigarrow u_j \in V(C_y) \right\} \\
&=^5 -d(u, x) + d(u, y) \\
&=^6 \Delta_x(u, y)
\end{aligned}$$

- 1: By definition of d^{δ_x} .
- 2: By definition of δ_x .
- 3: By collapsing the telescoping sum.
- 4: Because $u_j \in V(C_y)$ and $y \in \text{Children}(x)$, so $u_j \in V(C_x)$.
- 5: By definition of d .
- 6: By definition of Δ_x .

□

Next, we bound the size of the distances in G_x^δ .

Lemma 22 *Let $x \in \mathcal{H}$, $y \in \text{Children}(x)$, and $u \in V$. Then $d^{\delta_x}(u, y) \leq \text{Diam}(x)$.*

Proof:

$$d^{\delta_x}(u, y) =^1 \Delta_x(u, y) =^2 d(u, y) - d(u, x) \leq^3 \text{Diam}(x)$$

1: From lemma 21.

2: By definition of Δ_x .

3: Follows from the fact that $V_y \subset V_x$

□

Then, we bound how far off the $\widehat{\delta}_x$ -distances are from the corresponding δ_x -distances.

Lemma 23 *Let $x \in \mathcal{H}$, $y \in \text{Children}(x)$, and $u \in V$. Then*

$$d^{\delta_x}(u, y) \in [\epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y), \epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y) + \epsilon_x].$$

Proof: For an arbitrary edge e , we know from the definitions of δ_x and $\widehat{\delta}_x$ that either

$$\delta_x(e) > \text{Diam}(x) \quad \text{and} \quad \widehat{\delta}_x(e) = \infty \quad (2.3)$$

or

$$\epsilon'_x \cdot \widehat{\delta}_x(e) \leq \delta_x(e) < \epsilon'_x \cdot (\widehat{\delta}_x(e) + 1) \quad (2.4)$$

If we let $e \in P_{uy}$, the shortest path from u to y in G^{δ_x} , then by lemma 22 $d^{\delta_x}(u, y) \leq \text{Diam}(x)$. So equation 2.4 must be true, because all $\delta_x \geq 0$.

Then we have that

$$\begin{aligned} & \epsilon'_x \cdot \widehat{\delta}_x(e) \leq \delta_x(e) < \epsilon'_x \cdot (\widehat{\delta}_x(e) + 1) \\ \Rightarrow & \sum_{e \in u \rightsquigarrow y} \epsilon'_x \cdot \widehat{\delta}_x(e) \leq \sum_{e \in u \rightsquigarrow y} \delta_x(e) < \sum_{e \in u \rightsquigarrow y} \epsilon'_x \cdot (\widehat{\delta}_x(e) + 1) \\ \Rightarrow^1 & \epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y) \leq d^{\delta_x}(u, y) < \epsilon'_x \cdot (d^{\widehat{\delta}_x}(u, y) + |P_{uy}|) \\ \Rightarrow & \epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y) \leq d^{\delta_x}(u, y) < \epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y) + \epsilon'_x \cdot n \\ \Rightarrow^2 & \epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y) \leq d^{\delta_x}(u, y) < \epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y) + \epsilon_x \end{aligned}$$

1: By definition of d .

2: By definition of ϵ'_x .

□

And finally, we bound the size of the $\widehat{\delta}_x$ -distances.

Lemma 24 *Let $x \in \mathcal{H}$, $y \in \text{Children}(x)$, and $u \in V$. Then $d^{\widehat{\delta}_x}(u, y) \leq 2n \frac{\text{Diam}(x)}{\text{Norm}(x)}$.*

Proof:

$$d^{\widehat{\delta}_x}(u, y) \stackrel{\leq 1}{\leq} \frac{d^{\delta_x}(u, y)}{\epsilon'_x} \stackrel{\leq 2}{\leq} \frac{\text{Diam}(x)}{\epsilon'_x} \stackrel{= 3}{=} 2n \frac{\text{Diam}(x)}{\text{Norm}(x)}$$

1: Follows from lemma 23.

2: By lemma 22.

3: By definition of ϵ'_x .

□

Now for the actual correctness proof.

Lemma 25 *The $\widehat{\Delta}$ -function computed in section 2.8.4 is correct, i.e. each value $\widehat{\Delta}_x(u, y)$ is an integer that satisfies $|\epsilon_x \cdot \widehat{\Delta}_x(u, y) - \Delta_x(u, y)| < \epsilon_x$.*

Proof: It is obvious from step 3 in section 2.8.4 that all $\widehat{\Delta}$ -values are integer.

From the computation in section 2.8.4, and the definition of ϵ'_x , we get

$$\begin{aligned} \widehat{\Delta}_x(u, y) &= \left\lceil \frac{d^{\widehat{\delta}_x}(v, y)}{n} \right\rceil \in \left[\frac{d^{\widehat{\delta}_x}(v, y)}{n}, \frac{d^{\widehat{\delta}_x}(v, y)}{n} + 1 \right) \\ \Rightarrow \epsilon_x \cdot \widehat{\Delta}_x(u, y) &\in \left[\epsilon_x \frac{d^{\widehat{\delta}_x}(v, y)}{n}, \epsilon_x \frac{d^{\widehat{\delta}_x}(v, y)}{n} + \epsilon_x \right) \\ \Rightarrow \epsilon_x \cdot \widehat{\Delta}_x(u, y) &\in \left[\epsilon'_x d^{\widehat{\delta}_x}(v, y), \epsilon'_x d^{\widehat{\delta}_x}(v, y) + \epsilon_x \right) \end{aligned}$$

But from lemmas 21 and 23 we know that

$$\Delta_x(u, y) = d^{\delta_x}(u, y) \in [\epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y), \epsilon'_x \cdot d^{\widehat{\delta}_x}(u, y) + \epsilon_x)$$

So $\epsilon_x \cdot \widehat{\Delta}_x(u, y)$ and $\Delta_x(u, y)$ belong to the same interval of width ϵ_x , which implies that

$$|\epsilon_x \cdot \widehat{\Delta}_x(u, y) - \Delta_x(u, y)| < \epsilon_x$$

□

2.9.2 Correctness of VISIT

The correctness proof of VISIT is structured in the following way. We start by introducing a few new concepts, then we prove various helper lemmas, and finally we have the actual proof that the VISIT procedure is correct.

First, we define the notion of internal distances as follows:

Definition 26 *For vertex set X containing u and v , we define the internal distance $d_X(u, v)$ as the distance from u to v in the input graph restricted to the nodes of X .*

We use internal distances to define the concept of (S, I) -independence:

Definition 27 *Let X and S be sets of vertices and I a real interval. We say that X is (S, I) -independent if $d(s, v) = d_{S \cup X^I}(s, v)$ for all $v \in X^I$.*

What definition 27 means, is that if we have an (S, I) -independent vertex set X , then the shortest paths in X with lengths in I can be computed by only considering the vertices in $S \cup X^I$. This is a formalization of our previous vague references to “independent” in “independent subproblems”. Along the same lines, the next definition makes precise what we mean by “subproblem”.

Definition 28 *Let X, X_1, \dots, X_k be sets of vertices. We call $\langle X_1, \dots, X_k \rangle$ a t -partition of X , if it is a partition of X and all edges (u, v) where $u \in X_i, v \in X_j$, and $j < i$ have $l(u, v) \geq t$.*

Now a lemma that shows how the concepts of independence and t -partitions are linked. When reading it, think of the X_i as the children of X , and the definition of S_i as what our S looks like after i iterations of the loops in the VISIT algorithm.

Lemma 29 *Suppose we have that:*

- X is $(S, [a, b])$ -independent
- $\langle X_1, \dots, X_k \rangle$ is a t -partition of X

- $I = [a, \min\{a + t, b\})$
- $S_i = S \cup X_1^I \cup \dots \cup X_i^I$

then we know that:

- (1) X is $(S_k, [a + t, b))$ -independent.
- (2) X_{i+1} is (S_i, I) -independent.

Proof:

First part 1:

By definition 27 of independence, we have that $d(s, v) = d_{S \cup X^{[a, b)}}(s, v)$ for all $v \in X^{[a, b)}$. Now we split into two cases. First suppose that $a + t \geq b$.

Then $[a + t, b)$ is an empty interval, so $S \cup X^{[a, b)} = S_k = S_k \cup X^{[a + t, b)}$.

Suppose instead that $a + t < b$. Then

$S \cup X^{[a, b)} = S \cup X^{[a, a + t)} \cup X^{[a + t, b)} = S_k \cup X^{[a + t, b)}$. So in any case

$S \cup X^{[a, b)} = S_k \cup X^{[a + t, b)}$, which means that X is $(S_k, [a + t, b))$ -independent.

Then for part 2:

For X_{i+1} to be (S_i, I) -independent, it must be true that

$d(s, v) = d_{S_i \cup X_{i+1}^I}(s, v) = d_{S_{i+1}}(s, v)$ for all $v \in X_{i+1}^I$. We prove this by

contradiction. So suppose there exists a $v \in X_{i+1}^I$ such that the shortest path from s to v is not contained in S_{i+1} . On this path, let w be the last vertex that is not in S_{i+1} .

We know that $d(s, v) \in I \subseteq [a, b)$, so by the $(S, [a, b))$ -independence of X we know that w is either in S or in X . But since $w \notin S_{i+1} \supseteq S$, w must be in X . Now we split into three cases:

If $d(s, w) < a$ then w is neither in S nor in $X^{[a, b)}$. This implies that $d(s, v) < d_{S \cup X^{[a, b)}}(s, v)$ which contradicts the $(S, [a, b))$ -independence of X .

If $a \leq d(s, w) < \min\{a + t, b\}$ then $w \notin S_{i+1}$ implies that $w \in S_k \setminus S_{i+1}$.

Then, by definition of a t -partition, $d(w, v) \geq t$. This, together with $d(s, v) \in I$ gives us that

$\min\{a + t, b\} > d(s, v) = d(s, w) + d(w, v) \geq d(s, w) + t$ which implies that $a > d(s, w)$, a contradiction with the assumption of this case.

If $d(s, w) \geq \min\{a + t, b\}$ then the non-negativity of edge weights implies that $d(s, v) \geq d(s, w) \geq \min\{a + t, b\}$ which contradicts v being in X_{i+1}^I . \square

In the following lemmas we assume that invariants 5 and 4 are automagically updated behind our backs. So when vertices are added to S , edge relaxations take place, and D -values are updated which may in turn cause nodes to be moved to different buckets.

First the lemma that was promised in section 2.7.2 about VISIT fixing the D -value upon entry.

Lemma 30 *Suppose $V(C_x)$ is (S, I) -independent. Then if $\text{VISIT}(x, I)$ is the first VISIT call on x , then $D(x) = d(s, x)$ and $d(s, x) \in I$.*

Proof:

The very first VISIT call is always $\text{VISIT}(\text{Root}, [0, \infty))$. $D(\text{Root})$ is initially 0 and cannot be diminished by relaxations, because we assume that all edge weights are non-negative. We know that $d(s, \text{Root}) = 0$ because s is a descendant of Root . All in all: $D(\text{Root}) = d(s, \text{Root}) = 0 \in [0, \infty)$.

Now for a general VISIT call where $I = [a, b)$. Just before the first $\text{VISIT}(x, I)$ call, $p(x)$ must be active (because the call happens) and x must be inactive (because this is the first call on x). Moreover, x must be in the $p(x)$ -bucket that covers I . Then invariant 5 says that $D(x) \in I$. By assumption, $V(C_x)$ is (S, I) -independent. So, $a \leq d(s, x) \leq D(x) < b$. Because of the (S, I) -independence, $D(x)$ cannot be affected by relaxations from vertices outside the subgraph C_x . And because $D(x)$ is the minimum of all the D -values of x 's descendants, it cannot be affected by relaxations from vertices inside the subgraph. So it must be the case that $D(x) = d(s, x)$. □

Now to a lemma that describes a property of the *Norms* and *Diams* of \mathcal{H} . It will come in handy for the upcoming lemma.

Lemma 31 $\forall x \in \mathcal{H} : \text{Either } \text{Diam}(x) < \text{Norm}(p(x)) \text{ or } \text{Norm}(x) \text{ divides } \text{Norm}(p(x)).$

Proof: Suppose x and $p(x)$ are in the same stratum. Then by definition, $\text{Norm}(p(x)) = 2^k \cdot \text{Norm}(x)$ for some integer k , so $\text{Norm}(x)$ divides $\text{Norm}(p(x))$.

Suppose that x is at a lower stratum than $p(x)$, and let $Norm(p(x)) = l_j^* \cdot 2^{level(p(x))-1}$. Then we know that:

$$\begin{aligned}
Diam(x) &<^1 (|V(C_y)| - 1) \cdot 2 \cdot Norm(x) \\
&< n \cdot 2 \cdot Norm(x) \\
&<^2 n \cdot 2 \cdot \frac{l_j^*}{2n} \\
&= l_j^* \\
&\leq^3 Norm(p(x))
\end{aligned}$$

- 1: By definition of $Diam$.
- 2: By definition of $Norm$ and normalizing edge weights.
- 3: $p(x)$ cannot be a leaf, so $level(p(x)) \geq 1$ by definition of levels.

□

The following lemma shows that in a VISIT call, $Norm$ divides the permission interval we are given, and that our bucket intervals line up properly. The only exception is when VISIT is just about done with this node.

Lemma 32 *Suppose $V(C_x)$ is (S, I) -independent. Then in a call to $VISIT(x, [a, b])$, either $V(C_x)^{[0, b]} = V(C_x)$ or we know that $b - a_x$ is divisible by $Norm(x)$.*

Proof: Suppose this is the first VISIT call on x , then a_x is initialized to t_x (algorithm 2 line 16).

If $D(x) + Diam(x) < b$ then t_x has been set to $D(x)$. Lemma 30 tells us that $D(x) = d(s, x)$, so we know that $\max_{v \in V(C_x)} \{d(s, v)\} \leq D(x) + Diam(x) < b$. This implies that $V(C_x)^{[0, b]} = V(C_x)$.

If $D(x) + Diam(x) \geq b$ then t_x has been set to $b - Norm(x) \lceil \frac{b - D(x)}{Norm(x)} \rceil$. Then $Norm(x)$ divides $b - t_x$ which is initially equal to $b - a_x$. In the while-loops, a_x is only incremented in steps of $Norm(x)$, so $Norm(x)$ still divides $b - a_x$.

Now suppose this is not the first VISIT call on x , then a_x is initialized to a . By lemma 31 either $Norm(p(x)) > Diam(x)$ or $Norm(p(x))$ is divisible by $Norm(x)$.

If $Norm(p(x)) > Diam(x)$: This is not the first VISIT call, so there must have been an earlier call $VISIT(x, [a', b'])$ where $a' < b' \leq a$. By lemma 30, this means that $d(s, x) \in [a', b']$, so $d(s, x) < a$. Then we know that $d(s, x) + Diam(x) < a + Diam(x) < a + Norm(p(x)) = b$. This implies that $V(C_x)^{[0, b]} = V(C_x)$.

If $Norm(x)$ divides $Norm(p(x)) = b - a$ then $Norm(x)$ initially divides $b - a_x$. In the while-loops, a_x is only incremented in steps of $Norm(x)$, so $Norm(x)$ still divides $b - a_x$. \square

And now, finally, we get to the lemma that shows the correctness of VISIT:

Lemma 33 *VISIT is correct: A call to $VISIT(x, [a, b])$ ensures that $V(C_x)^{[a, b]} \subseteq S$.*

Proof: Before we go any further, we make a slight change in VISIT to simplify some steps in the proof. Instead of having the inner while-loop of algorithm 2 (lines 21-30) treat only the child nodes in the current bucket, we let the loop iterate over *all* child nodes of x . For each child y , we check whether y is in the current bucket. If it is, we proceed as usual, if not, we do nothing. Correctness-wise, this is clearly equivalent to what is done in algorithm 2 – only the running time changes.

The correctness proof for VISIT consists of four simultaneous inductions, no less. The four are:

- **Induction over time**

IF $V(C_{p(x)})$ is (S, I_p) -independent when $VISIT(p(x), I_p)$ is called
THEN $V(C_x)$ is $(S, [a, b])$ -independent when $VISIT(x, [a, b])$ is called.

- **Induction over size**

IF $\forall y \in Children(x) : VISIT(y, I_y)$ ensures that $V(C_y)^{I_y} \subseteq S$
THEN $VISIT(x, [a, b])$ ensures that $V(C_x)^{[a, b]} \subseteq S$.

- **Induction over the outer while-loop**

Let a_x, S and a_x^*, S^* be the values before and after an iteration of the outer while-loop, respectively.

IF $V(C_x)^{[0, a_x]} \subseteq S$ and $V(C_x)$ is $(S, [a_x, b])$ -independent.

THEN $V(C_x)^{[0, a_x^*]} \subseteq S^*$ and $V(C_x)$ is $(S^*, [a_x^*, b])$ -independent.

- **Induction over the inner while-loop**

Let S' be the value of S at the beginning of the current outer while-loop. Let $\{y_i\}_i$ be $Children(x)$. Let $I = [a_x, a_x + Norm(x))$ for the current value of a_x .

IF $S = S' \cup V(C_{y_1})^I \cup \dots \cup V(C_{y_{j-1}})^I$ before the j^{th} iteration of the (rewritten) inner while-loop

THEN $S = S' \cup V(C_{y_1})^I \cup \dots \cup V(C_{y_j})^I$ after the iteration.

Clearly, the second of the four is the one we are after, but to get there, we need the other three. First we prove the base case for each induction. Gradually, we start assuming the IF-parts and finally, we prove the THEN-parts.

Base case for time: The base case for the time of the call is the very first call $VISIT(Root, [0, \infty))$. If $x = Root$, then $p(x)$ does not exist, so the IF-part is empty. Then we must prove that under all circumstances, V is (\emptyset, R_+) -independent, meaning that $\forall v \in v : d(s, v) = d_{V, R_+}(s, v)$. This follows from the fact that all edge weights are in R_+ .

Assumption 1: From now on we assume that $V(C_x)$ is $(S, [a, b])$ -independent when $VISIT(x, [a, b])$ is called.

Base case for size: The base case for the size of a graph is a leaf. If x is a leaf, then $Children = \emptyset$, so using only Assumption 1, we have to prove that $VISIT$ adds a leaf x to S iff $d(s, x) \in [a, b)$. The $VISIT$ algorithm itself (line 2) gives us that $D(s, x) \in [a, b)$, and the $(S, [a, b])$ -independence from Assumption 1, implies that $D(s, x) = d(s, x)$.

Assumption 2: From now on we assume that $\forall y \in Children(x) : VISIT(y, I)$ ensures that $V(C_y)^I \subseteq S$.

Base case for the outer loop: The base case for the outer while-loops is the very first outer loop. Before the first loop, a_x has been set to either a or t_x . If $a_x = a$ then we must prove that $V(C_x)$ is $(S, [a, b])$ -independent and that $V(C_x)^{[0, a)} \subseteq S$. The former follows from Assumption 1. The latter follows from the assumption on x 's parent which must exist because if a_x is set to a , this is not the first $VISIT$ call on x . On the other hand, if a_x was set to t_x , then by definition of t_x , $a_x \leq D(x)$. Again, we know that $D(x) = d(s, x)$, so $V(C_x)^{[0, a_x)} = \emptyset \subseteq S$. This implies that $V(C_x)^{[a_x, b)} = V(C_x)^{[a, b)}$, so Assumption 1 gives us that $V(C_x)$ is $(S, [a_x, b))$ -independent.

Assumption 3: From now on we assume that at the beginning of an iteration of the outer while-loop, $V(C_x)^{[0, a_x]} \subseteq S$ and $V(C_x)$ is $(S, [a_x, b])$ -independent.

Base case for the inner loop: The base case for the inner while-loop is the first loop. This means that we are at the beginning of the outer while-loop. So we need to prove that $S = S'$ when $S = S'$. We leave this to the reader.

Assumption 4: From now on we assume that $S = S' \cup V(C_{y_1})^I \cup \dots \cup V(C_{y_{j-1}})^I$ before the j^{th} iteration of the (rewritten) inner while-loop.

THEN-part for time:

Let $I = [a_x, a_x + \text{Norm}(x))$. Suppose we are just about to call $\text{VISIT}(y_j, I)$, then we want to show that $V(C_{y_j})$ is (S, I) -independent.

By Assumption 3, $V(C_x)$ is $(S, [a_x, b])$ -independent. By lemma 3 $\text{Children}(x)$ is a Norm -partition of $V(C_x)$. By Assumption 4, $S = S' \cup V(C_{y_1})^I \cup \dots \cup V(C_{y_{j-1}})^I$. All this put together satisfies the requirements for lemma 29 part 2 to give us that

$$V(C_{y_j}) \text{ is } (S, [a, \min\{a + \text{Norm}(x), b\}])\text{-independent} \quad (2.5)$$

Statement 2.5 is not quite what we wanted, so we continue. By lemma 32, either $V(C_x)^{[0, b]} = V(C_x)$ or we know that $b - a_x$ is divisible by $\text{Norm}(x)$.

Suppose that $V(C_x)^{[0, b]} = V(C_x)$. If $a + \text{Norm}(x) > b$ then $V(C_x)^{[0, b]} = V(C_x)^{[0, a + \text{Norm}(x))}$, so $V(C_{y_j})$ is (S, I) -independent by statement 2.5. If $a + \text{Norm}(x) \leq b$ then we have (S, I) -independence from statement 2.5 right away.

Suppose that $b - a_x$ is divisible by $\text{Norm}(x)$. Since we are about to call $\text{VISIT}(y_j, I)$ we must be in the while-loops which means that $a_x < b$. Then the only way the Norm can divide $b - a_x$ is if $a_x + \text{Norm}(x) \leq b$. This combined with statement 2.5 implies that $V(C_{y_j})$ is (S, I) -independent. This concludes the proof for the induction over time.

THEN-part for the inner loop:

Let $I = [a_x, a_x + \text{Norm}(x))$. We must show that after the j^{th} iteration of the inner loop, $V(C_{y_j})^I$ has been added to S . In the (rewritten) inner loop, either $\text{VISIT}(y_j, I)$ is called or nothing happens. If the call is made, Assumption 2 gives us the desired result directly. If no call is made, we

must show that $V(C_{y_j})^I$ is already in S . If no call is made then by definition y_j is not in bucket I . Then invariant 5 says that either $V(C_{y_j}) \subseteq S$ or $D(y) \geq a_x + \text{Norm}(x)$, which implies that $V(C_{y_j})^I = \emptyset$. Either way, we get the desired result. This concludes the proof of the induction over the inner loop.

THEN-part for the outer loop:

We need to show that $V(C_x)^{[0, a_x^*]} \subseteq S^*$ and that $V(C_x)$ is $(S^*, [a_x^*, b])$ -independent.

By assumption 3, we have $V(C_x)^{[0, a_x]} \subseteq S$. The induction on the inner loop gives us that $S^* = S \cup V(C_x)^{[a_x, a_x + \text{Norm}(x)]}$. Then we have that $V(C_x)^{[0, a_x^*]} \subseteq S^*$, because we get $a_x^* = a_x + \text{Norm}(x)$ from the last line of the loop.

Just like in the proof for the induction over time, we use Assumption 3, lemma 3, and Assumption 4 to fulfill the conditions of lemma 29. Part 1 of that lemma then gives us directly that $V(C_x)$ is $((S^*, [a_x^*, b]))$ -independent. This concludes the proof of the induction over the outer loop.

THEN-part for size: We need to show that $\text{VISIT}(x, [a, b])$ ensures that $V(C_x)^{[a, b]} \subseteq S$. When the outer while-loop terminates, we know that either $a_x \geq b$ or $\text{NotDone}(x) = \emptyset$. If $a_x \geq b$ then we know from the induction over the outer loop that $V(C_x)^{[0, b]} \subseteq S$, implying $V(C_x)^{[a, b]} \subseteq S$. If $\text{NotDone}(x) = \emptyset$ then $V(C_x) \subseteq S$, which also implies that $V(C_x)^{[a, b]} \subseteq S$. This completes the proof for the induction over size, and we are done. \square

Chapter 3

Practice

In this chapter, we turn our attention to actual implementation. We will examine the practicality of the Pettie algorithm and compare it to its natural competitor, Dijkstra. We discuss implementation issues of both algorithms, and run tests on various types of input graphs. We interpret this collection of empirical measurements to find out whether or not the Pettie algorithm has practical value. First, some remarks that apply to the implementations of both SP algorithms.

3.1 Common Factors

No matter which SP algorithm we are implementing, there are some issues that we must always consider. They are discussed in this section.

3.1.1 Graph Representation

For the SSSP runs of both Pettie and Dijkstra, we use the *forward-star* graph representation. It is an adjacency list representation described in [17] which was found by Cherkassky et al. [4] to be the most efficient graph representation technique for shortest paths algorithms.

The idea of forward-star is very simple: we have two arrays of integers A_e and A_v . Array A_e is of size m and contains the ID number of the destination vertex of each input graph edge. The array is ordered by source

vertex so that the first entries in the array correspond to the outgoing edges of vertex 1, then come vertex 2's outgoing edges etc. A_v is of size n and contains indices into A_e . If $A_v[i] = j$ and $A_v[i + 1] = k$ it means that the entries $A_e[j], \dots, A_e[k - 1]$ contain the destination vertices of vertex i 's outgoing edges.

This representation makes for instance the loop in the edge relaxation very efficient, because iterating through neighboring entries of an array is usually fast in practice.

However, for convenience we have used a slower linked-list implementation of the adjacency lists in the *build-graph* of the Pettie algorithm. This is not necessary, as there are other ways to build the hierarchy than the one we have used. This is of little importance, however, since building the hierarchy will turn out to take up only a microscopic part of the total execution time for solving APSP (see section 3.4.2).

3.1.2 Programming Language

We have chosen to implement the algorithms in C++, which is a natural choice given that we focus on performance aspects of the algorithms. It makes our programs highly effective, with very little of the performance overhead found in e.g. Java. This effectiveness in turn results in accurate, transparent timing measurements with no interference from virtual machines etc.

Choosing C++ also means that we can use the excellent code optimization of the gcc compiler to produce fast executables, while having access to the Standard Template Library (STL) data structures, and allowing an object-oriented style of programming.

To ensure comparability between algorithms, we strive to make all implementations as effective as possible. For instance, we use as little dynamic memory allocation as possible during the SSSP runs.

3.1.3 Weight Data Type

Real-weighted algorithms such as Dijkstra and Pettie are designed to be very flexible with regard to the data type of the edge weights. We are

therefore free to use just about any types we see fit. In the experiments presented here, we use the basic C++ type `double`.

In preliminary experiments, we have measured the effect of using simpler types by comparing performance on `doubles`, `floats`, `long longs`, and `ints`. Our findings are very similar to those of Pettie et al. [33] in that using different basic types does not change the asymptotic behaviour of an algorithm, it only adds a fixed amount of time to the computations (or subtracts it).

Our implementations are constructed so that they could easily be made to accept custom data types such as rationals or extremely large numbers. However, we have conducted no experiments on such types.

3.2 The Dijkstra Algorithm

When implementing the standard Dijkstra algorithm, there are two main issues to consider: what type of priority queue to use, and whether it should contain all the vertices right from the beginning, or add them as their D -value falls below ∞ . In our preliminary experiments, adding the vertices to the priority queue as we go along has proven unequivocally superior, so this is the method of choice in all the Dijkstra results we present.

Regarding the type of priority queue there are many options available, not least because the theoretically fastest Fibonacci heap is usually not very efficient in practice.

In a study by Moret and Shapiro [28], different priority queues for the Prim-Jarnik minimum spanning tree algorithm [22, 34] are evaluated experimentally. This algorithm is very similar to the Dijkstra SP algorithm, so we can reasonably expect their findings to apply in our setting, too. They compare the performance of binary heaps, pairing heaps [15], splay trees [36], Fibonacci heaps [14] and rank-relaxed heaps [11]. In their experiments, the pairing heap is found to be the most efficient in almost all cases. Only on a dense graph type with $m = n\sqrt{n}$ is it beaten by the binary heap.

Therefore, in section 3.4.1 we run Dijkstra using a hand-coded pairing heap, a hand-coded binary heap, and the STL `multimap` which is implemented as a red-black search tree. One might think that a red-black

tree would be too slow for our purposes, because it needs to do extra work that a heap does not. However, in the study [7] by Dementiev et al., the integer version of the STL `map` (a red-black tree, too) showed excellent performance. Though their measurements are done on random sequences of operations not induced by an algorithm, we have chosen to give the very similar `multimap` a chance in our tests.

In STL, the `multimap` returns a pointer (`multimap::iterator`) to the place in the tree where a vertex is inserted. This pointer is updated as other vertices are inserted and removed. This means that a `DECREASEKEY` operation on `multimap` consists of a very fast removal (by following the pointer) plus a normal insertion. The STL `priority_queue` (which at first glance appears suitable for our purposes) is implemented as a binary heap in an array, but it does not have this iterator-feature. Therefore, its `DECREASEKEY` has a horrible $O(k)$ running time where k is the number of vertices in the heap. This is the reason the hand-coded binary heap is included in our experiments instead.

3.3 The Pettie Algorithm

This section presents our thoughts and ideas on implementing the Pettie algorithm. An implementor of Pettie has many choices to make: how to construct the hierarchy, how to implement buckets, whether to use Gabow's \mathcal{DSF} , etc. Luckily, though, most of the choices are independent of one another. For instance, how we handle the D -values has no influence on the execution time of the bucket structure. Still, we make no claim of exhausting all possibilities in our experiments.

3.3.1 Building the component hierarchy

In our implementation, we build the hierarchy the way it was described in sections 2.6.1 and 2.7.1. We start out with an empty build-graph, add edges, discover SCCs, perform topological sorts, and add hierarchy nodes as subgraphs are contracted. We simulate the contractions by the standard *up-tree* union-find implementation [6] with path compression. Thus, contracting an SCC is a series of `UNION` operations on its nodes. The

addition of an edge to the build-graph then requires a FIND operation to determine which SCC representatives are to be connected by the edge.

It is quite likely that a faster implementation of the hierarchy creation exists. However, as we have mentioned, the hierarchy phase is very quick compared to the execution time of the entire APSP computation.

Therefore, we have chosen to spend our optimization efforts elsewhere.

We represent the hierarchy in an object-oriented style; a node is an object with field variables, such as D -value, references to its bucket structure and to parent and child node objects. More on this representation later.

3.3.2 The Oracle Function

To calculate the oracle function, we first extend our graph representation to include a forward-star version of the input graph where all edges are reversed. Then we proceed as described in section 2.8.4, running regular Dijkstra on the existing edge weights, computing new edge weights and running Dial-Dijkstra on those. This is done for each hierarchy node x where $\frac{Diam(x)}{Norm(x)} \geq \beta$ for some fixed $\beta \in [0, 2n]$.

The regular Dijkstra run is implemented with the priority queue found to be the best in section 3.4.1. The Dial-Dijkstra run is implemented with a STL `vector` as the array, and hand-coded doubly-linked lists as the buckets. We conduct various experiments to determine the usefulness of the oracle, the results are shown in section 3.4.3.

3.3.3 The VISIT Procedure

The VISIT procedure is implemented very much like the pseudocode in algorithm 2. Some minor changes are due to problems with the machine representation of numbers. Sometimes, especially when the edge weights are very close to 0, or drawn from a very narrow interval, trouble can occur. This is mostly with respect to the bucketing.

One has to be very careful when calculating the interval endpoints for the buckets, and the “permission intervals” $[a, b)$ that are passed as arguments to VISIT. For instance, one cannot be sure that evaluating $2*q$ yields exactly the same result as $q+q$ if q is of type `double`. As a consequence of

this, we make sure to always perform computations on variables like a_x in a consistent way. Another consequence is that we always use one more bucket than is theoretically necessary in our bucket structure. This prevents child nodes with d -values very close to the last endpoint from falling just outside the buckets.

From profiling measurements (using `gprof`) we have found that the `VISIT` function itself (not counting bucketing etc.) is responsible for about 15% of the execution time of an SSSP computation.

The most significant code improvement relating to `VISIT` has actually made the procedure a little slower... At the beginning of each SSSP run, our Pettie implementation originally had a cleanup phase that traversed the hierarchy, setting D -values to ∞ etc. We were able to do away with almost all of this traversing, shaving off over 10% of the total SSSP running time. This is done by e.g. letting `VISIT` reset the D -value of a node to ∞ when we are sure that the value will not be read until the next SSSP run.

So it seems that the very traversing of the hierarchy is quite costly compared to the actual operations that we execute. This observation suggests that there could be something to gain from a no-frills array-based representation of the hierarchy – as opposed to our nicer OO-type representation.

3.3.4 The Bucket Structure

Profiling measurements have revealed that approximately 20-25 % of the total SSSP execution time is spent on bucketing operations, so this is an obvious place to try to optimize. But while other parts of the Pettie algorithm are handled by standard well-known data structures, it is less obvious how the bucket structure is most efficiently implemented. We propose and test four different bucketing schemes:

- **Lazy:** The lazy bucketing scheme from section 2.8.3, implemented very much as it is described there. The super-buckets are hand-coded doubly-linked lists, as we have found the performance of STL `list` to be very poor – a scan through an STL `list` is up to twice as slow as a scan through our extremely simple home-made list.

- **Straight-forward:** This is the straight-forward bucketing scheme described just before the lazy bucketing in section 2.8.3. Here, a node has an array of buckets, each implemented as a hand-coded list. When a child node is to be inserted, we calculate the bucket number i given the D -value and add the node to the end of the list in entry i .

In preliminary experiments, we have tried locating the bucket by a binary search as is done in the theory chapter, but this proved too slow in all cases. We have also done experiments to determine a pattern in the distribution of nodes over buckets. If we let b_i be the number of buckets between the front bucket and some later bucket i , then we have found bucket i to typically contain about $O(\frac{1}{b_i})$ nodes. We have tried exploiting this by first searching linearly through a small number of buckets, and then reverting to the standard method if unsuccessful. Alas, this also proved inferior to simple calculation. However, for advanced weight data types where e.g. multiplication is expensive, the search techniques just described might prove useful.

- **Eager:** This is the same bucketing scheme as the straight-forward above, except that each bucket in the array is now a priority queue, ordering nodes by their (topological) order in the children-array of their parent. In other words, every bucket is acting as if it was the front bucket. This scheme makes rebucketing more expensive, but if a node never moves, it is inserted only once, and does not need to be transferred to a special front bucket. This scheme was designed with the oracle function in mind, which ensures that a node is rebucketed less than three times.
- **Collapsed:** In this scheme there is no array, just one bucket that contains all the child nodes. It is implemented as a priority queue ordering nodes lexicographically by $(bucket, topo)$, where $bucket$ is the index of the logical bucket in which the node belongs, and $topo$ is the node's index in its parent's array of children. We can then simulate the iteration through the buckets by just extracting them from the front of the priority queue.

Now we turn to implementation of the “front bucket” which is needed in all bucketing schemes except the collapsed scheme.

The Front Bucket

The front bucket can be implemented using any standard integer priority queue. The only operations we require are `INSERT`, `GETMIN`, and `DELMIN`. However, while the lazy and straight-forward bucketing schemes have only one front bucket, the eager bucketing scheme needs many “front bucket” instantiations. This means that if we use the eager scheme, we add the requirement that an empty front bucket must take up very little space.

In the recent Dementiev et al. paper [7], integer priority queues are evaluated experimentally. On the operations that we require of a front bucket, the most efficient are the STL `map`, the LEDA `(2, 16)`-tree [25] and the authors’ own `S-tree` structure defined in the article. Two implementations of the van Emde Boas heap (the authors’ and LEDA’s) are found to be 3-6 times slower than the rest. For sizes up to about 60.000 nodes, the `map`, `(2, 16)`-tree, and `S-tree` yield roughly the same performance.

Typically, a front bucket will contain few nodes, though of course it depends heavily on the structure of the hierarchy. But even for very large input graphs, front buckets containing more than 60.000 nodes are extremely uncommon in our experience. Therefore, we include front bucket implementations that use `S-trees` and STL `maps` in our experiments. The LEDA library is no longer freely available, so we do not include the `(2, 16)`-tree. Instead we try the integer version of the STL `priority_queue`. The front bucket needs no `DECREASEKEY` operation, so the `priority_queue` should have a fair chance of performing well.

In preliminary experiments, we tested our own implementation of a van Emde Boas heap, but came to the same conclusion as Dementiev et al., so it is not included in the results presented here.

We should note that the `S-tree` takes up quite a lot of memory, even when empty, because it allocates an array of length 2^{16} upon instantiation. Therefore, it is not suited for our eager bucketing scheme, so that combination is not used.

3.3.5 Dijkstra on Subproblems

Acknowledging that Dijkstra is a very simple, efficient algorithm, we experiment with using it on subproblems up to some fixed size. This means that before the SSSP runs begin, we traverse the hierarchy and recursively count the number of leaves in the subtree of each node. We fix a size threshold T , and then traverse the hierarchy again to identify those nodes whose number of subtree leaves is less than T while their parent has more than T leaves. These nodes are converted to *Dijkstra nodes*. This entails deleting the node's entire hierarchy subtree, except for the leaves. We also assign to such a node x a priority queue $Q(x)$ in which it keeps its leaves, just like a regular Dijkstra computation.

Then in the SSSP runs, if x is a Dijkstra node and we call $\text{VISIT}(x, [a, b])$, we do not execute the usual VISIT procedure. Instead, x starts popping leaves from $Q(x)$, and performing edge relaxations from them. It stops when the D -value of the front leaf in $Q(x)$ is outside $[a, b]$. For these Dijkstra runs on subproblems, we use the priority queue found to be the fastest in section 3.4.1.

Using Dijkstra on subproblems is inspired by the way a van Emde Boas tree is usually implemented, where sorted lists or other simple data structures are used to handle the “bottom” of the vEB-tree.

3.3.6 Handling D -values

Handling the D -values for leaves just consists of the usual relaxation loop. Nevertheless, the sheer number of relaxations that must be performed, means that this takes up over 50% of the Pettie execution time. Therefore, we have been careful to implement the relaxation loop so that it uses as few basic operations as possible. As we have seen, the relaxation loop is the same in Pettie and Dijkstra, so any tinkering that causes a speedup is applied to both algorithms. However, the faster we can relax edges, the clearer we can see the effects of optimizing other subroutines and data structures, so it is worth the while.

In section 2.8.2, we described how the Pettie algorithm utilizes the Gabow \mathcal{DSF} data structure to handle D -values for internal hierarchy nodes. Unfortunately, it turns out not to be very practical for the graph sizes we

will be considering. This is actually because the Ackermann function grows too fast. When we first initialize the \mathcal{DSF} , on the outermost level, it typically contains an enormous sublist in which each super-element presides over only four leaves. This is because the only permitted next step is to make a super-element preside over 2^{16} leaves. So if we have 60.000 vertices in the input graph, we will initially have a sublist of length 20.000 on the outermost level. This makes the first SPLIT alone very time-consuming in practice.

To let the size of the super-elements grow at a more suitable pace, we have implemented a “degenerate” \mathcal{DSF} which has only one level. So there are no recursive calls, and super-elements are allowed to have size 2^i for any integer i . This gives a theoretical time bound of $O(m + n \log n)$ instead of Gabow’s $O(m\alpha(m, n))$, but we would expect it to be faster in practice.

We have also implemented an alternative approach, where no \mathcal{DSF} is used. Instead, whenever the D -value of a hierarchy leaf decreases, we let the new D -value “bubble up” in the hierarchy, decreasing D -values of the leaf’s ancestors until a parent with an even lower D -value is met. This approach renders SPLIT superfluous, FINDMIN becomes constant-time (we simply read the node’s D -variable), but DECREASEKEY now has worst-case time proportional to the height of \mathcal{H} . In practice, though, the average number of bubbling steps is very low.

In fact, in our initial experiments on many different graph types, the D -bubbling has turned out to be far better than both \mathcal{DSF} versions. This is due to two factors: The cost of \mathcal{DSF} SPLITS is much too high, and the penalty for bubbling compared to DECREASEKEY is very low, often non-existent. Furthermore, when we run Dijkstra on subproblems, the height of the hierarchy decreases, so bubbling becomes even more effective. Therefore, we use D -bubbling instead of \mathcal{DSF} in all the results that we present.

3.3.7 Experimental Setup

We run the algorithms on two different machines to examine possible variances due to hardware, operating system or compiler. We will refer to them as the Windows machine and the Linux machine. Their specifications are as follows:

- **The Windows machine** is a Dell Inspiron 8600 running Microsoft Windows XP 2002. It has a 1.4 GHz Intel Pentium M processor, 1 MB of cache, and 512 MB of DDR RAM. On this machine we compile using `gcc` version 2.95.2 with optimization option `-O6`.
- **The Linux machine** runs Debian Linux 2.4.18 on dual 1.7 GHz Intel Xeon processors¹ each with 256 KB cache and a total of 1 GB DDR RAM. On this machine we compile using `gcc` version 3.3.2, still with `-O6`.

We have run many experiments on both machines, but we have not detected any major platform dependent issues. However, there seems to be a tendency for differences between competing algorithms to stand out more clearly on the Linux machine, whereas programs running on the Windows machine tend to vary less in running times. For this reason, we choose the Linux machine as the default for producing the results in the sections to come.

3.3.8 Timing

We make sure both machines are as idle as possible when we start running our tests. On the Windows machine we are logged in as sole user “Administrator”, and we use this status to kill as many background processes as we can before the experiments begin. To measure the execution time of our programs we use the `clock` function which reports the total number of clock cycles elapsed.

On the Linux machine, we are logged in as a common user, so we have less control over background processes. On the other hand, this machine allows us to use the precise `getrusage` system call to measure the milliseconds spent by our process alone.

In the figures, we report the timing results divided by the number of edges of the input graph for better readability.

¹Though the Linux machine has two processors, our program only uses one of them, freeing up the other to execute background processes.

3.3.9 Measurements

There are four phases in our SP algorithms: Loading, initialization, SSSP run, and results. The loading phase loads the input graph from the disk and builds the graph representation from section 3.1.1. The initialization phase is empty for the Dijkstra algorithm, but for Pettie it consists of building the hierarchy and computing $\hat{\Delta}$ -values (if applicable). The SSSP run phase executes the Dijkstra algorithm or the Pettie VISIT recursion. The results phase extracts the D -values from all the vertices and reports them to the user in some way. The SSSP run phase and the result phase are repeated κ times.

In most of our experiments, we only measure the execution time of the κ SSSP run phases. The duration of the loading phase is irrelevant to our experiments, and the results phase performs the exact same operations on the exact same data structure no matter which algorithm we run. We also devote experiments to measure the execution time of hierarchy-building and matters relating to the $\hat{\Delta}$ -computations.

For the APSP problem, κ is of course n . But to be able to test very large graphs, we run SSSP from κ randomly chosen source vertices, where κ can be less than n , but is always at least 100. If the graph is small, we increase κ (sometimes beyond n) to get more reliable timing results.

3.3.10 Test Data

To perform our experiments, we need to have some input graphs on which to run. We know of no canonical set of test graphs that one can use to test graph algorithms such as SP. Even notable studies on SP implementations such as [4] use only a few different (quite artificial) graph types in their tests.

Below, we detail the kinds of graphs we have chosen to generate. One can of course devise infinitely many, more or less arcane, graph classes, but we divide our test graphs into three different groups: very random, clustered, and very sparse. Each group contains three graph classes, most of which are taken from [4] and [33]. They cover very structured graphs such as grid graphs and (artificial) road maps, as well as graphs where all edges are chosen completely at random.

To ensure that all the graphs are strongly connected, we always finish by adding a Hamiltonian cycle to the graphs we produce. Each weight on that cycle is heavier than all the regular weights combined, so any SP algorithm will use the edges of the cycle only as a last resort.

We use STL's `rand48` functions to provide the random numbers needed to generate the graphs. When we compare algorithms, we of course use the same sequence of random seeds, so that the competing algorithms run on the exact same graphs.

Most of our graph types define only structure, not edge weights, so unless otherwise specified, the graphs are assigned random edge weights uniformly in $[0, 1)$.

Very Random Graphs

- **b-ary:** A b -ary graph is a graph where every node has out-degree b . For each vertex, we randomly choose b vertices and connect the vertex to them, so $m = b \cdot n$. In our tests, $b = \log n$, unless otherwise noted. This makes for a sort of middle road between the very sparse graphs discussed below and more dense graphs where the time to relax edges overly dominates the running time.
- **$G_{n,m}$:** In $G_{n,m}$, we choose uniformly at random among all possible graphs with n vertices and m edges. We run tests with $m = n \log n$ for average out-degrees of about 5 to 20.
- **Geometric:** The geometric graphs are constructed by assigning random 2D-coordinates to the vertices such that they are all in a unit square. Then we add (bi-directional) edges between those vertex pairs whose Euclidian distance is less than some fixed threshold t . In our experiments, we use $t = \frac{5}{2\sqrt{n}}$, which gives an average out-degree of about 10. The Euclidian distance measure is also used as edge weight.

Clustered Graphs

- **Bullseye:** A bullseye graph consists of k consecutive subgraphs called *orbits*. Each orbit is a $G_{n,m}$ -type graph. Only neighboring orbits are connected, and then only by a few edges. We enumerate the orbits

and let orbit i draw its internal edge weights uniformly from $[2^i, 2^{i+1})$. The same weight interval is used for edges from orbit i to $i - 1$. We choose $k = 10$, so each orbit contains $\frac{n}{10}$ vertices. The hierarchy of such a graph is a root with 10 children, each of which is the parent of $\frac{n}{10}$ leaves.

- **Hierarchical:** This graph type consists of a hierarchy of $G_{n,m}$ -type subgraphs. The higher up the hierarchy, the heavier the edges, and the more vertices in the subgraphs. It reflects a sort of reverse engineering on the component hierarchy in that it is tailor-made for the component hierarchy to become a complete b -ary tree for some b . We fix b at 10 in our experiments.
- **Cities:** This graph consists of \sqrt{n} clusters called *cities* spread out randomly in a square whose sides have length 100. Each city is a $G_{n',m'}$ -type graph with $n' = \sqrt{n}$, $m' = n' \log n'$ and edge weights in $[0, 1)$. Two cities are connected by a highway (bi-directional edges) if their Euclidian distance is less than some t . The weight on a highway edge equals the Euclidian distance between its endpoints.

Very Sparse Graphs

- **Spanning Tree:** This graph type just consists of a bi-directional random spanning tree plus a few $(\frac{n}{100})$ edges between random vertex pairs. This makes $m \approx 2n$. The random spanning tree is created using a union-find structure U on the vertices. We perform UNIONS on random vertex pairs until U contains only one set. If $\text{UNION}(v_1, v_2)$ unites two sets, we add bi-directional edges between v_1 and v_2 in the graph.
- **Grid:** A grid is a graph where the vertices are arranged in a 2D-matrix and vertex (i, j) is (bi-directionally) connected to the vertices $(i + 1, j)$ and $(i, j + 1)$. This means that $m = 4n$. We test square grids as well as wide grids with a fixed height of 16.
- **Hypertorus:** This graph is a cubic 3D-grid bent into the shape of a (hard-to-imagine) hypertorus. Each edge in the grid is included with a fixed probability p . We set $p = \frac{5}{6}$ for an average out-degree of 5, which makes $m \approx 5n$.

3.4 Results

When we conduct a test run on a graph class, we run the algorithm on five different graphs of equal size from the graph class and take the average time per edge. Suppose, for instance, that we report that the binary heap version of Dijkstra takes 25 microseconds per edge for square grid graphs with 2^{18} edges. This means that we have run the algorithm on five different square grid graphs each having 2^{18} edges, added up the execution time for all five runs and divided by $5 \cdot 2^{18}$ to get 25.

We start by examining the internal issues of the Dijkstra and Pettie algorithms, finding the best heaps, schemes and so forth. Then we compare the best of Dijkstra to the best of Pettie on various graph types, weight data types and platforms.

3.4.1 The Dijkstra Priority Queue

First, we attempt to determine which priority queue is most effective for the Dijkstra algorithm. We run Dijkstra using a pairing heap, a binary tree, and a red-black tree, the STL `multimap`. We try all three queues on all nine types of graphs ranging in size from a few hundred edges to over a million. Figure 3.1 shows some of the results.

We observe that the execution time trends are somewhat different on the various graph classes. On $G_{n,m}$ graphs, Dijkstra slows down considerably when m increases beyond roughly 2^{16} meaning that the average out-degree is above 16. Throughout the range of sizes, the binary heap shows the best performance, followed by the pairing heap, leaving the `multimap` in last place.

For geometric graphs, the binary heap is also superior. This time, the execution times of the two other queues follow each other very closely. On the square grid, the binary heap wins again, but here the `multimap` is somewhat faster than the pairing heap for graphs larger than about 20,000 edges. For the bullseye graph, the `multimap` and pairing heap start off far apart, but their asymptotic behavior is very much alike. Once again, the binary heap wins.

Regarding the graph types not shown, the results for d -ary graphs are very close to $G_{n,m}$, all very sparse graphs are similar to the square grid, and the

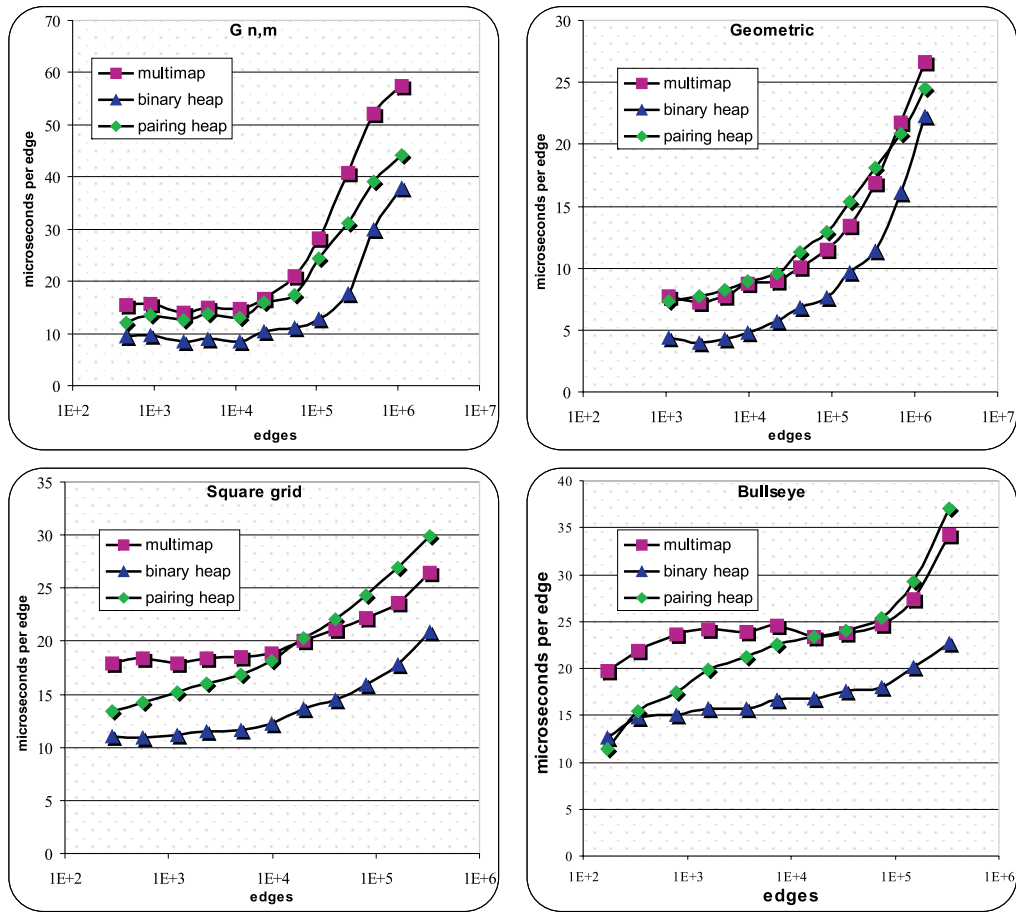


Figure 3.1: Results for Dijkstra using various priority queues.

asymptotical trends of the omitted clustered graphs resemble that of the bullseye graph.

There can be little doubt about the winning priority queue. In almost every single timing measurement performed, the binary heap is the fastest on our set of test graphs. This is contrary to what has been found in other studies where, as we have mentioned, the pairing heap was superior. Of course there can be reasonable explanations for this – for instance, it might be due to platform differences or the pairing heap might not be implemented as efficiently as possible. However, we have tested it against two other pairing heap implementations by other authors, and found it to be the best of the three.

If we look very carefully at the execution times for the largest $G_{m,n}$ and geometric graph types, we may detect a trend that could send the pairing heap into the lead for extremely large graphs with multi-million edges. But this is only speculation, and to test it would take more powerful hardware, or access to the our current hardware over a long period of time. Unfortunately, we have neither.

The fact remains that in our experiments, the binary heap emerges as the priority queue of choice for Dijkstra.

3.4.2 Building the Component Hierarchy

Now we start performing measurements on the Pettie algorithm. We begin by determining how much of the Pettie APSP time is spent building the hierarchy. We have performed various experiments on different graph types, and the following example represents the overall picture very well.

We run on $G_{n,m}$ graphs for varying n . As usual $m = n \log n$ and the weights are uniformly random in $[0, 1)$. We measure the time t_1 to build the hierarchy, as well as the time t_2 for n SSSP runs. Now we set $t = \frac{t_1}{t_1+t_2} \cdot 100\%$ and get the following results:

n	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}
t	1.80%	0.81%	0.40%	0.19%	0.11%	0.06%	0.03%	0.016%

We see that the relative time for our quick-and-dirty hierarchy-building routine rapidly decreases as the graphs grow, and it is negligible already for moderately large graphs. This is only natural, as the number of SSSP computations grows like n^2 and the number of vertices and edges participating in hierarchies grows like $n + m$.

3.4.3 The Oracle Function

Somewhat surprisingly, we have found in preliminary experiments that using the oracle function provides only a very small performance enhancement, if any at all. And this is without counting the time it takes to compute it! So even if we get the oracle for free, and thereby are able to approximately identify the final bucket of a hierarchy node, it does us very little good. This may seem to conflict with the earlier statement that we

use up to 25% of the total time on bucketing, but there is no contradiction as the following experiment shows:

We consider a series of b -ary graphs, where n is fixed at 10.000 vertices and b varies. We execute a number of SSSP runs with straight-forward bucketing on each graph, and along the way we count how many times each node is rebucketed. Common sense tells us that the higher the out-degree in the graph, the more likely rebucket operations become. The results confirm this assumption:

Out-degree b	3	7	11	15	19	...	50	...	100
Rebuckets per node per SSSP run	0.04	0.32	0.67	0.91	1.17	...	1.73	...	2.5

But the real story is not that the figures grow, it is how very few rebuckets are actually needed, even for dense graphs. At $b = 100$ we have $m = n\sqrt{n}$, and still less than three rebuckets per node. Remember, this is in the straight-forward bucketing scheme – for lazy bucketing the number of rebuckets is much smaller. And of course, for the very sparse graph types, the figures are smaller still. Actually, when it comes to rebucketing, the results in the table are from the worst bucketing scheme (along with the eager scheme) on the worst of our graph types.

The claim that actually very little rebucketing is happening during SSSP runs is backed up by `gprof` profiling which tells us that less than 0.5% of the total execution time is spent on the UPDATE calls to the bucket structure.

Given the facts above confirming our preliminary results, we conclude that in our setting, the oracle function has no practical value.

3.4.4 The Bucket Structure

In our Pettie implementation, there are many possible combinations of bucketing schemes and front bucket implementations. Now we perform a number of experiments to determine which combinations are the fastest. We conduct all experiments on the bucket structure without the use of Dijkstra on subproblems. This forces each bucket structure implementation to perform more operations, thus making differences in performance stand out more clearly.

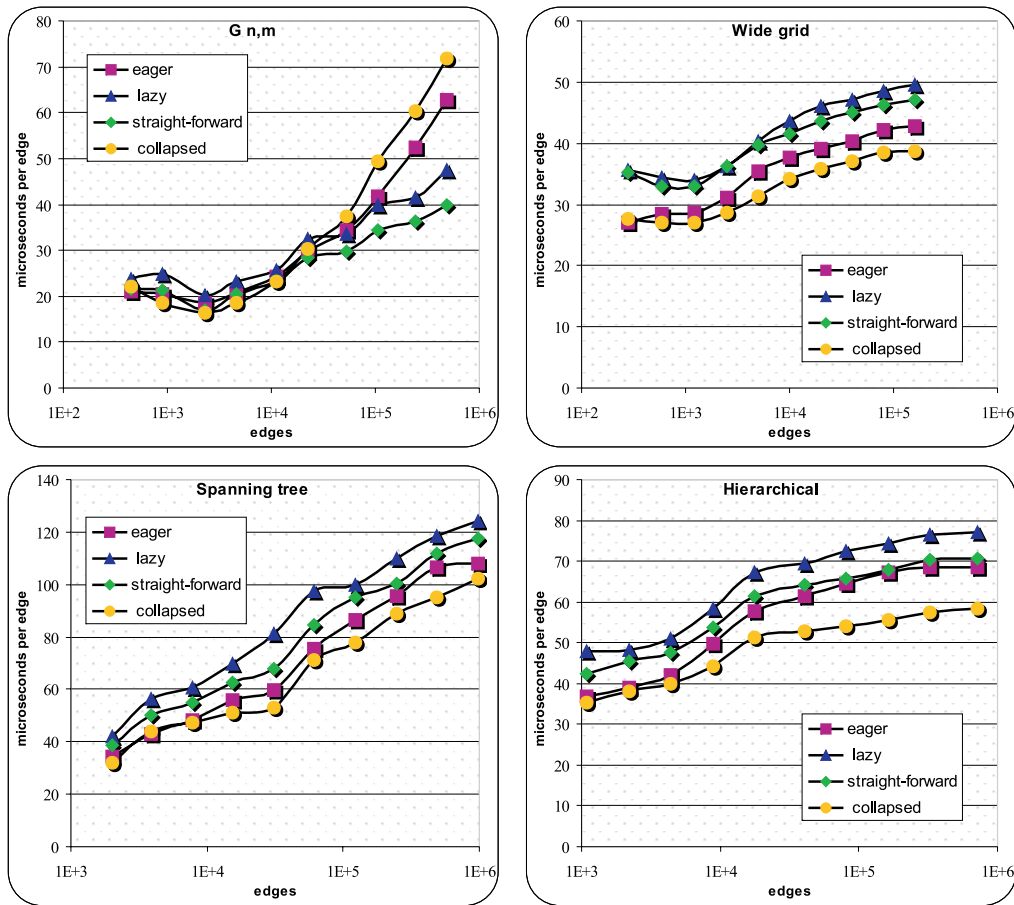


Figure 3.2: Results for Pettie using various bucketing schemes.

Bucketing Scheme

To measure the performance of our four different bucketing schemes described in 3.3.4, we have run the Pettie algorithm on a wide variety of graph types, sizes etc. A few of the obtained results are shown in figure 3.2. As the front bucket, we use the red-black tree, which may or may not be the fastest, but it ensures equal front bucket costs in all cases where front buckets are used.

We observe that for the grid graph, the spanning tree graph and the hierarchical graph, the patterns are almost identical: The collapsed scheme is the best, then comes the eager scheme followed by the straight-forward scheme, and the lazy scheme is in last place. The $G_{n,m}$ graph class stands

out, exhibiting very different execution time trends. Here, the performance on small graphs is very similar for all schemes, but from around 60.000 edges onwards, this changes dramatically. The more edges, the clearer it becomes that the straight-forward scheme is the best, followed by lazy, eager, and collapsed, in that order.

We interpret this to mean that the determining factor for the efficiency of the bucket structure is the average out-degree of the graph vertices. The component hierarchies resulting from the three graph types with similar performance are very different, so hierarchy structure hardly plays a role.

If we count in the heavy cycle that is added to all graphs, then the spanning tree graph has an average degree of about three, the grid has degree five and the hierarchical graph has degree about eight. But the degree of the $G_{n,m}$ graph increases with n , because $m = n \log n$. So at 60.000 edges, the degree is about 16. It seems that if the degree of the input graph is more than 16, then enough bucket operations are performed to let the theoretically superior lazy and straight-forward schemes show their strength. If the degree is low then the simplicity of the collapsed scheme saves it time on initialization – plus it avoids the scanning of empty buckets.

Results on other graph types (not shown) support our conclusion. In some cases the change occurs around degree 15, but the pattern remains the same.

Front Bucket

Exploring which type of front bucket implementation to use is really only relevant to the straight-forward and lazy bucketing schemes. The collapsed scheme has no front bucket at all, and the eager scheme must use the red-black tree `map`. As we have mentioned, this is due to the `S-tree` taking up too much space, and the `priority_queue` being much too slow to be used for all buckets in a bucket structure. In figure 3.3 we display some results for straight-forward and lazy bucketing combined with different front buckets.

We observe that in almost all cases, the `S-tree` is approximately twice as slow as `map` and `priority_queue`. In the results shown, the only exception is for b -ary graphs with a large number of edges, where the `S-tree` performs relatively better. The ratio between the `map` and the

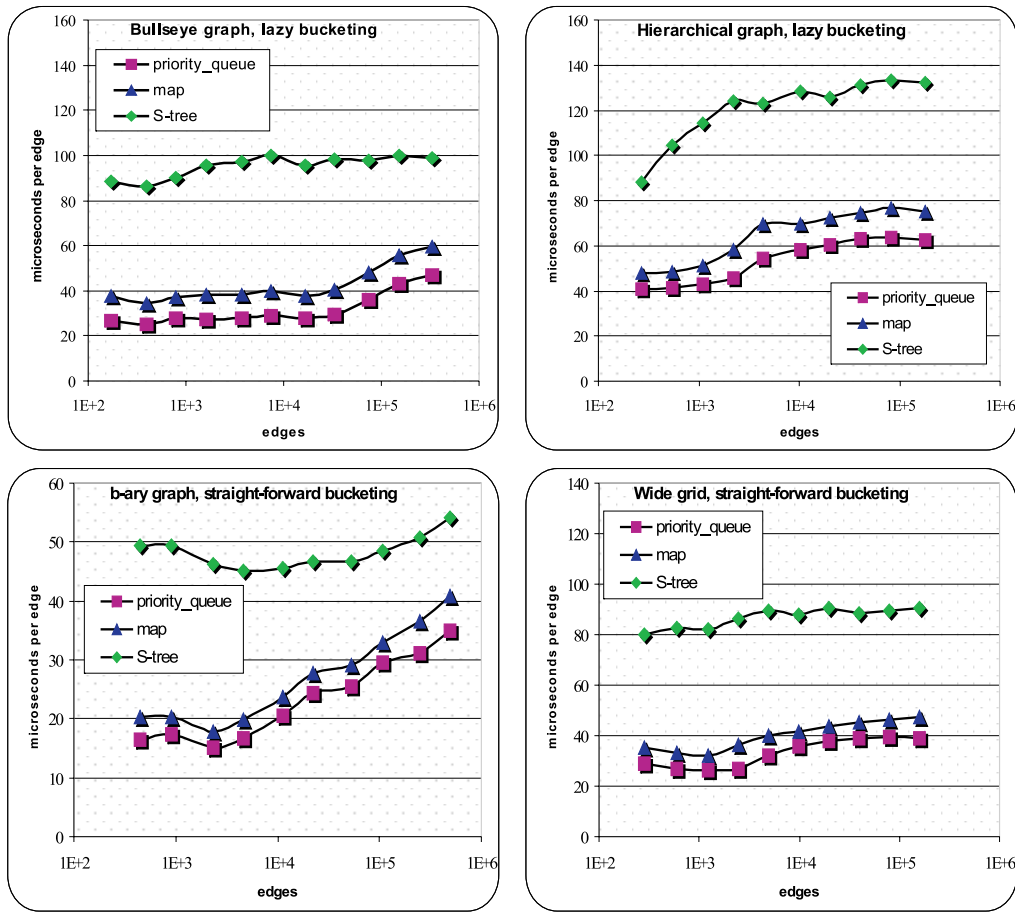


Figure 3.3: Results for Pettie using various front buckets for lazy and straight-forward bucketing.

`priority_queue` is very stable in all experiments; `map` is consistently 15-20% slower than `priority_queue`. There is no noticeable difference in front bucket efficiency between the bucketing schemes, neither in the results shown nor in those not shown.

We interpret the observations to mean that the average number of nodes in a front bucket is too small for the `S-tree` to be competitive. We have also conducted artificial tests (i.e. not as part of an algorithm) in which it was found that the `S-tree` is indeed effective, but only when it contains tens of thousands of elements. This is not the case for our range of graph sizes, so the `S-tree` never gets to display its asymptotical superiority.

As for the `map` and `priority_queue`, things are as could be expected. When the only operations required are `INSERT` and `DELETEMIN`, it is only natural that the binary heap outperforms the red-black tree.

The independence of front bucket performance from our choice of bucketing scheme is no surprise either, because the straight-forward and the lazy bucketing essentially treat the front bucket the same way.

We conclude from this experiment that the best front bucket choice for lazy and straight-forward bucketing is the STL `priority_queue`. This means that we now have four principal variations on the Pettie algorithm: Eager bucketing using STL `map`, collapsed bucketing without a front bucket, and straight-forward and lazy bucketing, both using the STL `priority_queue`. On these four variations, we now experiment with using Dijkstra on subproblems to speed things up.

3.4.5 Dijkstra on Subproblems

As mentioned in section 3.3.5, we explore the use of the Dijkstra algorithm on subproblems in the Pettie `VISIT` recursion. We have tried a number of different values for the size threshold T , from 0 to $n - 1$. When $T = 0$, it means that we do not use Dijkstra nodes at all. $T = n - 1$ means that the Pettie algorithm only orchestrates and coordinates $Deg(Root)$ Dijkstra computations. In Pettie et al.'s article [33] on the practicality of the similar hierarchy-based algorithm for undirected graphs, a T -value of 50 is chosen without further explanation. We have not been successful in determining an optimal value for T . It seems to depend very much on the problem instance.

Not surprisingly, there has been a tendency for slow algorithms to become faster as T increases. But this is mostly of no use to us, as those algorithms would benefit from becoming complete Dijkstra algorithms anyway. In the following section, we set the T -value quite high at \sqrt{n} . In our experience, this gives a good trade-off between the quickness of Dijkstra and the coordination of Pettie.

For this T -value, we have compared the execution times of our four Pettie variants. On almost all graphs, it turns out that the straight-forward and lazy bucketing schemes are now superior. Sometimes only by a hair, other times very convincingly, e.g. on b -ary graphs. The advantage of the collapsed scheme on very sparse graphs observed in section 3.4.4 has now

vanished with the use of Dijkstra. Therefore, we choose the straight-forward and lazy schemes to compete against Dijkstra.

3.4.6 Dijkstra vs. Pettie

Now for the comparison between the Dijkstra and Pettie algorithms. In section 3.1 we found that the binary heap version of Dijkstra was the most efficient, so it is that implementation that we measure the two best Pettie variants against. Figure 3.4 shows some results of the match-ups.

We observe that for small and medium graph sizes, Dijkstra always has the lowest execution times. When we move into the realm of large graphs, the picture is less clear. On the b -ary graph, Pettie catches up to Dijkstra just before the one million edge mark, which makes the degree around 19. For hierarchical graphs, it happens much sooner, at 100.000 edges, and this is the only graph type where Pettie is clearly in the lead for large graphs. On most other graph types, especially the very sparse graphs, Dijkstra has a comfortable lead.

On almost all graphs, the Dijkstra curves behave similarly – with bullseye graph as a notable exception. We also note the almost identical behaviour between the Pettie variants on all graphs except the b -ary.

We interpret the results shown to mean that on very sparse graphs, SP algorithms have very little freedom to visit vertices in the order they see fit. This means that Dijkstra and Pettie will very often make similar choices, and it appears that Pettie’s bucketing overhead is greater than Dijkstra’s penalty for sorting.

The hierarchical graph type always produces a b -tree-like component hierarchy for some b , so this fits the Pettie algorithm very well. On the more random graphs, graphs need to be fairly large for Pettie to gain the upper hand. But then it appears that it does pay off in the long run to coordinate a number of Dijkstra computations rather than perform one big Dijkstra.

So all in all, our version of the Pettie algorithm comes out as a moderately successful implementation, though we are nowhere near the excellent results from [33]. In the last section, we give a few ideas to improve on this.

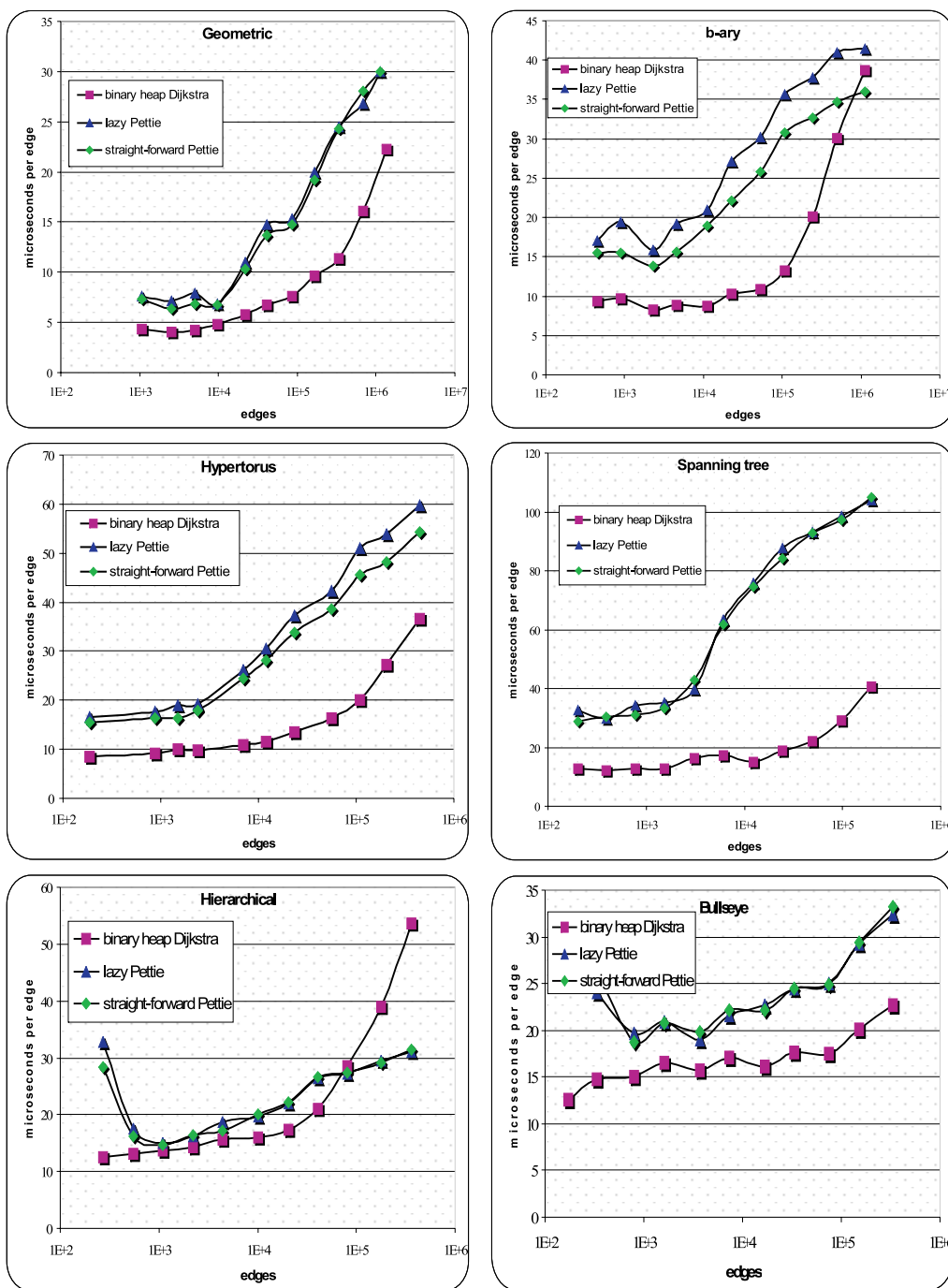


Figure 3.4: Results for the best Pettie variants against the best Dijkstra.

3.5 Conclusions and Future Work

In this chapter, we have conducted, to our knowledge, the first exploration of the practicality of the Pettie algorithm. Though the results are not overwhelming, we do manage to beat Dijkstra on a few graph types, so the road ahead looks promising.

It is quite likely that a better implementation of the Pettie algorithm can be devised, as there are certainly many parameters to adjust and many possibilities to explore. The bucket structure alone can be implemented in any number of ways. One such way could include spending a short initial time gathering information about the graph and/or component hierarchy and then use that knowledge to determine which kind of bucket structure is appropriate for the problem at hand.

Our implementation would probably also benefit from a leaner bare-bones design with emphasis on fast array-like structures and less memory consumption, as opposed to the current, nicely modularized OO-style.

Wrapping up this chapter, we propose the following areas of future work:

It would be interesting to explore how Pettie's algorithm performs in practice if the data type for the edge weights is more advanced than the basic C++ types. This might require another type of implementation where more thought is put into minimizing the number of basic operations such as '+'.

Another avenue of interest could be to examine the performance of Pettie on real-world graphs such as road maps. We suspect that such graphs would create hierarchies similar to those of our clustered graphs, so the Pettie algorithm would have a reasonable chance of doing well. On such graphs, computing the hierarchy would also be a rare occurrence, as traffic networks tend to change less often than, say, computer networks.

Finally, there might be something to gain from making some form of automatic adjustments on the size bound T for Dijkstra nodes in the component hierarchy. It seems quite unsatisfying to just fix it at a certain value regardless of hierarchy structure. A special case of the automated T -adjustment could be to set it to $n + 1$, which means that we just perform a normal Dijkstra run, and do not involve a hierarchy. Such an automation would make the Pettie algorithm a generalization of Dijkstra, capable of

degenerating into standard Dijkstra on graph types unsuited for the component hierarchy approach.

Code Availability

This thesis and the source code of the implementation will be available shortly at <http://www.daimi.au.dk/~karsten/thesis>

Bibliography

- [1] Wilhelm Ackermann. Zum hilbertschen aufbau der reellen zahlen. *Math. Ann.*, 99:118–133, 1928.
- [2] R. E. Bellman. On the routing problem. *Quart. Appl. Math*, 16:87–90, 1958.
- [3] Bernard Chazelle. *The discrepancy method: randomness and complexity*. Cambridge University Press, 2000.
- [4] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1994.
- [5] J. S. Chuang and D. Roth. Gene recognition based on dag shortest paths. *Bioinformatics*, 2001.
- [6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.
- [7] Roman Dementiev, Lutz Kettner, Jens Mehnert, and Peter Sanders. Engineering a sorted list data structure for 32 bit keys. In *Algorithm Engineering and Experiments (ALENEX'04)*, 2004.
- [8] Robert B. Dial. Algorithm 360: shortest-path forest with topological ordering. *Commun. ACM*, 12(11):632–633, 1969.
- [9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [10] E. A. Dinic. Finding shortest paths in a network. In *Transportation Modeling Systems*, pages 36–44. Institute for System Studies, Moscow, 1978.

- [11] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert E. Tarjan. Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31(11):1343–1354, 1988.
- [12] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.
- [13] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.
- [14] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *J. ACM*, pages 338–346, 1984.
- [15] Michael L. Fredman, Robert Sedgwick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.
- [16] H. N. Gabow. A scaling algorithm for weighted matching on general graphs. In *Proc. 26th Annual Symposium on Foundations of Computer Science (FOCS'85)*, pages 90–100, 1985.
- [17] G. Gallo and S. Pallottino. Shortest paths algorithms. *Annals of Oper. Res.*, 13:3–79, 1988.
- [18] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1983.
- [19] Andrew V. Goldberg. Scaling algorithms for the shortest paths problem. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 222–231. Society for Industrial and Applied Mathematics, 1993.
- [20] Torben Hagerup. Improved shortest paths on the word ram. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 61–72, 2000.
- [21] Yijie Han. Improved fast integer sorting in linear space. *Inf. Comput.*, 170(1):81–94, 2001.
- [22] V. Jarník. O jistém problému minimálním. *Práce mor. přírodověd. spol.*, 6:57–63, 1930.

- [23] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.
- [24] David R. Karger, Daphne Koller, and Steven J. Phillips. Finding the hidden path: Time bounds for all-pairs shortest paths. In *IEEE Symposium on Foundations of Computer Science*, pages 560–568, 1991.
- [25] Kurt Mehlhorn and Stefan Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [26] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound on the complexity of the union-split-find problem. *SIAM J. Comp.*, 17(6), 1988.
- [27] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.
- [28] B. M. E. Moret and H. D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. *Lecture Notes in Computer Science*, 519, 1991.
- [29] László G. Nyúl, Alexandre X. Falcão, and Jayaram K. Udupa. Fuzzy-connected 3d image segmentation at interactive speeds. *Graphical Models*, 64(5):259–281, 2002.
- [30] Rózsa Peter. Über der zussammenhang der verschiedene begriffe der rekursiven funktion. *Math. Ann.*, 110:612–632, 1934.
- [31] Seth Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312:47–74, 2004.
- [32] Seth Pettie and Vijaya Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 267–276. Society for Industrial and Applied Mathematics, 2002.
- [33] Seth Pettie, Vijaya Ramachandran, and Srinath Sridhar. Experimental evaluation of a new shortest path algorithm. In *ALLENEX'02*, 2002.
- [34] A. Prim. Shortest connecting networks and some generalizations. *Bell Syst. Tech J.*, 36:1389–1401, 1957.

- [35] Rajeev Raman. Recent results on the single-source shortest paths problem. *SIGACT News*, 28(2):81–87, 1997.
- [36] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
- [37] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comp.*, 1(2):146–160, 1972.
- [38] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, 1999.
- [39] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.
- [40] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.
- [41] Uri Zwick. Exact and approximate distances in graphs - a survey. In *Proceedings of the 9th Annual European Symposium on Algorithms*, pages 33–48. Springer-Verlag, 2001.

Appendix A

Operations in Our Computational Model

In our computational model we allow two types of values: integers and reals. No conversion between types is allowed. The only operations initially available are addition and comparison for all values, and array lookup for integers. All are constant-time. Below we show how to implement other operations.

SUBTRACT. We simulate subtraction by representing each real value a as (a_1, a_2) where $a = a_1 - a_2$. Then

- $\text{ADD}(a, b)$ returns $(a_1 + b_1, b_2 + a_2)$
- $\text{SUBTRACT}(a, b)$ returns $(a_1 + b_2, b_1 + a_2)$
- $\text{LESSTHAN}(a, b)$ returns $(a_1 + b_2 < b_1 + a_2)$

So all are still constant-time.

MULTIPLY. We can multiply a real r by an integer i in $O(\log i)$ time. Given r and i , generate the lists $L = \{r, 2r, 4r, \dots, 2^{\lfloor \log i \rfloor} r\}$ and $L' = \{1, 2, 4, \dots, 2^{\lfloor \log i \rfloor}\}$ by repeatedly adding each new list element to itself. This takes $O(\log i)$ additions total. Then set $r' = 0$, $i' = 0$ and scan L from right to left. At every j , check whether $i' + L'[j] > i$. If yes, do nothing, if no, add $L[j]$ to r' and add $L'[j]$ to i' . When we reach the end of the list,

$r' = i \cdot r$. This uses at most three additions and one comparison per list entry, so the total time is $O(\max\{1, |L|\}) = O(1 + \log i)$.

FLOOR-DIVIDE. Given reals a and b , we can compute the integer $\lfloor \frac{a}{b} \rfloor$ in $O(1 + \log \frac{a}{b})$ time. We generate the lists $L = \{b, 2b, 4b, \dots, 2^{\lfloor \log \frac{a}{b} \rfloor} b\}$ and $L' = \{1, 2, 4, \dots, 2^{\lfloor \log \frac{a}{b} \rfloor}\}$ by repeatedly adding each new list element to itself. This takes $O(\log \frac{a}{b})$ additions total. Then set $r' = 0, i' = 0$ and scan L from right to left. At every j , check whether $r' + L[j] > a$. If yes, do nothing, if no, add $L[j]$ to r' and add $L'[j]$ to i' . When we reach the end of the list $i' = \lfloor \frac{a}{b} \rfloor$. This uses at most three additions and one comparison per list entry, so the total time is $O(\max\{1, |L|\}) = O(1 + \log \frac{a}{b})$.

Appendix B

NP-hard Shortest Paths Problems

The issue is this: We have a graph $G = (V, E, f)$ (integer or real weighted, directed or undirected, it does not matter) that contains negative cycles. To avoid negative infinity-solutions to the SP problem, we simply forbid any utilization of negative cycles. We do this by adding the condition that in a path, no sub-part that starts and ends in the same node can have a negative total weight.

Formally it looks like this:

In a path $P = (v_1, v_2), (v_2, v_3), \dots, (v_i, v_{i+1}), \dots, (v_{j-1}, v_j), \dots, (v_k, v_{k+1})$ we demand that if $v_i = v_j$ then $\sum_{t=i}^{j-1} f((v_t, v_{t+1})) \geq 0$.

We will call the APSP problem with this additional path condition, SIMPLE NEG APSP. We will assume that the APSP algorithm produces both lengths and paths as output.

Lemma 34 SIMPLE NEG APSP is NP-hard.

Proof: We note that whatever the input to a SIMPLE NEG APSP algorithm, there can be no cycles at all in its output. Positive cycles only add to the total length of a path and can therefore not occur in a shortest path. Zero-cycles can be discarded without changing the total length of a path, and negative cycles in paths are expressly forbidden by the condition above. So, any solution to SIMPLE NEG APSP consists of simple paths, meaning paths that do not visit the same vertex twice.

The problem HAMILTONIAN PATH is a well known NP-complete problem [18]. Our claim is that if we can solve SIMPLE NEG APSP then we can solve HAMILTONIAN PATH with only polynomial-time extra work. In other words, we can reduce HAMILTONIAN PATH to SIMPLE NEG APSP.

Construction:

Given an unweighted graph as input to HAMILTONIAN PATH, we assign a weight of -1 to every edge. Now we run SIMPLE NEG APSP on this graph to get the $n \times n$ solution matrix M of shortest path lengths. We then spend $O(n^2)$ time traversing M . If we encounter the value $1 - n$ then we answer YES, and return the path from v_1 to v_2 where $M(v_1, v_2) = 1 - n$. Otherwise we answer NO.

Correctness:

Any Hamiltonian path visits n vertices, so it consists of $n - 1$ edges. Then, by construction of the graph, it must have total weight $1 - n$.

Conversely, if a path in the solution has total length $1 - n$ then, by construction of the graph and by definition of total length, it must consist of $n - 1$ edges. That means that it visits n vertices. As mentioned above, any path in the solution of SIMPLE NEG APSP must be simple. So we have a path that visits each of n nodes exactly once. That is a Hamiltonian path. \square

Note that SIMPLE NEG SSSP is also NP-hard, as we can solve APSP using any SSSP algorithm by just running it on each node in the graph.

Appendix C

Ackermann's Function

The original Ackermann function was introduced by Rózsa Péter in 1934 [30], based on a 1928 observation by Ackermann [1]. Here, we present a variant similar to that of Chazelle [3]. To define this function, we first need an infinite matrix F constructed according to the following recurrence relation:

$$\begin{aligned}
 F(0, j) &= 2j && \text{for } j \geq 0 \\
 F(i, 0) &= 0 && \text{for } i \geq 1 \\
 F(i, 1) &= 2 && \text{for } i \geq 1 \\
 F(i, j) &= F(i - 1, F(i, j - 1)) && \text{for } i \geq 1, j \geq 2
 \end{aligned}$$

The matrix looks like this:

0	2	4	6	8	10	...
0	2	4	8	16	32	...
0	2	4	2^{2^2}	$\underbrace{2^{2^{\dots^2}}_4}$	$\underbrace{2^{2^{\dots^2}}_5}$...
0	2	4	$\underbrace{2^{2^{\dots^2}}_4}$	$\underbrace{2^{2^{\dots^2}}_{2^{16}}}$	$\underbrace{2^{2^{\dots^2}}_{2^{16}}}$...
⋮	⋮	⋮	⋮	⋮	⋮	⋮

We notice that:

$$\begin{aligned} F(0, j) &= 2 \cdot j \\ F(1, j) &= 2^j \\ F(2, j) &= \underbrace{2^{2^{\cdot^{\cdot^2}}}}_j \end{aligned}$$

and for $F(3, j)$ the towers-of-two run completely amok! So the rows grow really fast. But look at the columns. From column 3 onwards, they each grow faster than any primitive recursive function. This is also true for the diagonal. And already at $F(4, 4)$, the number is too huge to be written as a decimal expansion in the physical universe.

We can now define Ackermann's function as simply:

$$A(x, y) = F(x, y)$$

Or, if we only have one parameter, as the diagonal values:

$$A(x) = F(x, x)$$

Similarly, we can define the inverse Ackermann function for one parameter:

$$\alpha(y) = \min\{x \mid A(x) \geq y\}$$

Or, for two parameters:

$$\alpha(x, y) = \min\{r \mid A(r, 3 + \lfloor \frac{x}{y} \rfloor) \geq y\}$$

The following is a lemma showing that the inverse Ackermann function is slow-growing enough for our purposes.

Lemma 35 $m\alpha(m, n)$ is $O(m + n \log \log n)$

Proof: We split the proof into three cases by the number of edges.

First, suppose $m \in [n, n \log \log \log n]$. In this case, we need to prove that $m\alpha(m, n)$ is $O(n \log \log n)$.

$$\begin{aligned} & m \cdot \alpha(m, n) \leq^1 m \cdot \alpha(n, n) \\ \Rightarrow^2 & m \cdot \alpha(m, n) < n \log \log \log n \cdot \alpha(n, n) \\ \Rightarrow^3 & m \cdot \alpha(m, n) = O(n \log \log \log n \cdot \log \log \log n) \\ \Rightarrow^4 & m \cdot \alpha(m, n) = O(n \log \log n) \end{aligned}$$

1: By definition of α and the range of m .

2: By the range of m .

3: Because $A(i, 3 + \lfloor \frac{n}{n} \rfloor) = A(i, 4)$ obviously grows faster than $2^{2^{2^i}}$.

4: Because \log grows like \ln , and if we set

$$f(x) = \ln \ln x \quad \text{and} \quad g(x) = \ln \ln \ln x \cdot \ln \ln \ln \ln x$$

$$\text{then} \quad \lim_{x \rightarrow \infty} f = \infty \quad \text{and} \quad \lim_{x \rightarrow \infty} g = \infty$$

$$\text{and} \quad f'(x) = \frac{1}{x \cdot \ln x} > \frac{\ln \ln \ln \ln x + 1}{x \cdot \ln x \cdot \ln \ln x} = g'(x)$$

Now, suppose that $m \in [n \log \log \log n, n \log \log n]$. Here we again need to prove that $m\alpha(m, n)$ is $O(n \log \log n)$.

$$\begin{aligned} & m \cdot \alpha(m, n) \leq^1 m \cdot \alpha(n \log \log \log n, n) \\ \Rightarrow^2 & m \cdot \alpha(m, n) < n \log \log n \cdot \alpha(n \log \log \log n, n) \\ \Rightarrow^3 & m \cdot \alpha(m, n) = O(n \log \log n) \end{aligned}$$

1: By definition of α and the range of m .

2: By the range of m .

3: Follows from lemma 36

Finally, suppose that $m \geq n \log \log n$. This means that we must prove that $m\alpha(m, n)$ is $O(m)$ when m is in this interval. This follows from lemma 36. \square

Lemma 36 *If $m \geq n \log \log \log n$ then $\alpha(m, n)$ is $O(1)$.*

Proof: If $m \geq n \log \log \log n$ then it follows from the definition of α and F that $m = n \log \log \log n$ maximizes $\alpha(m, n)$ for some fixed n . So it is sufficient to prove that $\alpha(n \log \log \log n, n)$ is less than some constant C for all n greater than some constant N_0 .

By definition:

$$\begin{aligned}\alpha(n \log \log \log n, n) &= \min\{r \mid A(r, 3 + \lfloor \frac{n \log \log \log n}{n} \rfloor) \geq n\} \\ &= \min\{r \mid A(r, 3 + \lfloor \log \log \log n \rfloor) \geq n\}\end{aligned}$$

We choose $C = 2$ and $N_0 = 4$. So we claim that if we set $r = 2$ then $A(r, 3 + \lfloor \log \log \log n \rfloor) \geq n$ for all $n \geq 4$. This follows from the fact that

$$F(2, 3 + j) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^2}}}}}_{3+j} > 2^{2^{2^j}} \text{ for } j \geq 0. \quad \square$$