All Pairs Shortest Paths: Randomization and Replacement

Erin Davis erin.davis@wellesley.edu Eugenio Fortanely erf@mit.edu

December 11, 2013

Abstract

We present a survey of results in the field of graph algorithms. We begin with an All Pairs Shortest Path algorithm with some constraints that runs in $O(n^2)$ by Peres. We continue with a Replacement Paths algorithm by Williams and conclude with a smoothed analysis of a Single Source Shortest Paths algorithm by Banderier. Each of these problems has connections to randomization in graph algorithms.

1 Introduction

In this paper we will compare and contrast three related graph algorithms, with All Pairs Shortest Path algorithm as the primary. All Pairs Shortest Path is the computation of the shortest path between each pair of vertices in a graph. We will be relating this to the shortest replacement path and single source shortest paths with smoothed analysis. Shortest replacement path is the problem of finding the shortest path if a certain edge in your already shortest path is eliminated. Smoothed analysis turns an arbitrary graph, possibly a hard instance, into a semi-randomized graph, and therefore has tighter analysis.

In each section, we will discuss a problem and one solution: in section 2, the All Pairs Shortest Path problem; in section 3 the shortest replacement path problem; and in section 4, the smoothed analysis of a single source shortest path.

2 All Pairs Shortest Path

2.1 Introduction to All Pairs Shortest Path

The All Pairs Shortest Path (APSP) problem is an important and frequently studied problem [1][4][8]. Given a graph $G = (V, E)$ and a non-negative cost function $c : E \to [0, \infty)$, we want to compute the smallest distance between each pair of vertices in the graph. Throughout this paper, the words cost, weight, and distance are used interchangeably. Solving this problem allows you to rapidly find the shortest path between any two vertices. If you know ahead of time that you are doing many shortest path computations, then you can solve the APSP to cut some computation out of the total computational cost.

The brute force approach to APSP is to compute shortest path on each of the vertices using Dijkstra's algorithm. This would run in $O(mn + n^2 \log n)$ time. However, many of these computations are redundant, so algorithms have been developed that compute the APSP in less time. In 2004, an

$O(mn + n^2 \log\log n)$ was proven by Pettie[11]. Karger et al. and McGeoch developed algorithms that run in $O(m^*n + n^2 \log n)$, where m^* is the number of edges in the graph that are in shortest paths [4][12].

Another approach to simplifying the APSP problem is to only study graphs that exhibit nice properties. For instance, many researchers have examined complete directed graphs, graphs with random edge weights, and complete directed graphs with random edge weights. The simplest such graphs are the set of random graphs with edge weights drawn independently and at random from the uniform distribution on $[0, 1]$ [2]. These are the set of graphs that Peres et al. have been analyzing and that we will explain. We will see later graphs that don't necessarily fit this model, mostly in the shortest replacement path and smoothed analysis sections.

Others proved bounds on m^*, the number of edges expected to be in the APSP. Hassin and Zemel [13] noted that only the $O(\log n)$ shortest edges emanating from a single vertex will be included in shortest paths. Using that knowledge, we can then run Karger's or Dijkstra's algorithm on only that set of edges to achieve a runtime of $O(n^2 \log n)$ [2].

Peres et al. [2] proved an $O(n^2)$ algorithm for computing APSP on a complete directed graph with edge weights chosen independently and uniformly from $[0, 1]$. Their research did not introduce a new algorithm, but rather combined a static version of Demetrescu and Italiano's dynamic algorithm with a bucketing data structure and their analysis yielded an APSP solution in $O(n^2)$ runtime with high probability [1].

2.2 The Algorithm of Demetrescu and Italiano

Demetrescu and Italiano worked on a dynamic All Pairs Shortest Path problem [1]. The dynamic version maintains APSP subject to dynamic changes such as edge insertion, deletion and modification. They used *Local Shortest Paths* to solve this problem in $O(n^2 \log^3 n)$ amortized time per update on any sequence of operations.

Definition 2.1: A path π_{xy} is *locally shortest* in G if either
i. π_{xy} consists of a single vertex
ii. every proper subpath of π_{xy} is a shortest path in G [1].

An alternate definition of *Locally Shortest Path* from Peres et al. is:

Definition 2.2: A path is a *locally shortest path* if the path obtained by deleting its first edge and the path obtained by deleting its last edge are both shortest paths [2].

With these definitions, we can introduce some notation. Let $u \to u' \rightsquigarrow v' \to v$ denote a path composed of the edge $u \to u'$, the shortest path from u' to v' and the edge $v' \to v$. This path is considered *locally shortest* iff $u \to u' \rightsquigarrow v'$ is the shortest path from u to v' and $u' \rightsquigarrow v' \to v$ is the shortest path from u' to v. Any shortest path is a locally shortest path, but any locally shortest path is not necessarily a shortest path.

Let `apsp` be Demetrescu and Italiano's algorithm for finding APSP, for each $u, v \in V$ we will keep track of:

1. The current shortest path, $d(u, v)$
2. The second vertex on the shortest path from u to v, $p[u, v]$

3. The penultimate vertex on the shortest path from u to v, $q[u, v]$
4. The set of *left extensions* of the shortest path from u to v, $L[u, v]$
5. The set of *right extensions* of the shortest path from u to v, $R[u, v]$

A *left extension* is a vertex w so that $w \to u \rightsquigarrow v$ is known to be a shortest path, and a *right extension* is a vertex w so that $u \rightsquigarrow v \to w$ is known to be a shortest path.

For each $u, v \in V$ there is a $dist(u, v)$ which is initialized to 0 if $u = v$, $c(u, v)$ if $(u, v) \in E$, and ∞ otherwise. We add each of these pairs into a priority queue with the distance as the key. While the heap is not empty, we consider each pair and add the potential left extensions for this shortest path to $L[u, v]$ and the potential right extensions to $R[u, v]$ to consider whether these new paths are shortest paths. If they are, then they are locally shortest paths. At the end of this algorithm, $dist(u, v)$ will be the shortest path from u to v. Pseudocode for this algorithm can be found in the Appendix [2]. With `apsp` now defined, we can assert the following theorem.

Theorem 2.1: If all edge weights are positive and all shortest paths are unique, then `apsp` correctly finds all the shortest paths in the graph. Algorithm `apsp` runs in $O(n^2(T_{ins}(n^2) + T_{ext}(n^2)) + |LSP|T_{dec}(n^2))$ time, where $T_{ins}(n)$, $T_{dec}(n)$, and $T_{ext}(n)$ are the amortized time of inserting an element, decreasing the key and extracting the minimum element from a priority queue of size n, and $|LSP|$ is the number of locally shortest paths in the graph. This uses only $O(n^2)$ space [2].

This algorithm only takes $O(n^2)$ space since we are considering $dist[u, v]$ for every vertex pair and the lists for left and right extensions cannot take up more than $O(1)$ space each.

Using Fibonacci heaps as the priority queue, T_{ext} is $O(\log n)$ and T_{ins} and T_{dec} are both $O(1)$. Thus applying this theorem asserts a time bound of $O(n^2 \log n + |LSP|)$. To prove an $O(n^2)$ bound, we need to show both that $|LSP|$ is bounded by $O(n^2)$ and that we need a faster way of implementing heaps to get the $O(n^2 \log n)$ down to $O(n^2)$.

2.3 Bounding the number of Locally Shortest Paths

We want to demonstrate that the expected number of locally shortest paths is $O(n^2)$. We are asserting this claim on K_n, the complete graph on n vertices with random edge weights chosen independently from $[0, 1]$. To do this, we will combine two lemmas from Peres et al. [2] to assert the theorem that $|LSP| = O(n^2)$.

Lemma 2.1: Let a, b, c be three different vertices. The probability that $a \to b \to c$ is an LSP is $O(\frac{\ln^2 n}{n^2})$.

A sketch of the proof: $a \to b \to c$ is an LSP if and only if $a \to b$ and $b \to c$ are both shortest paths. But, unfortunately, the probability that $a \to b$ is a shortest path and $b \to c$ is a shortest path are not independent events. Thus, with some fiddling of probabilities and events, the authors find that the probability that $a \to b \to c$ is at most $(\frac{\ln(n/2)}{n/2} + O(\frac{1}{n}))^2$, which is $O(\frac{\ln^2 n}{n^2})$

Lemma 2.2: Let a, b, c, d be four distinct vertices. The probability that $a \to b \rightsquigarrow c \to d$ is an LSP is $O(\frac{1}{n^2})$.

The proof of this can be found in Peres et al. [2], but is too long to be included here. The gist of it is that with the randomization, we can bound the distance between a and c that avoids b and d as well as the distance between b and d that avoids a and c. Then, using the fact that each of these is the distance between 2 vertices on a randomly weighted complete graph on $n-2$ vertices, the authors conclude the $O(\frac{1}{n^2})$ bound. This encompasses all locally shortest paths of length ≥ 3 since $b \rightsquigarrow c$ means the shortest path which could be a path of 2 vertices or a path of $n-2$ vertices.

Theorem 2.2: $\mathbb{E}[\mid LSP \mid] = \Theta(n^2)$.

Proof: The number of LSPs of length 1 is exactly $n(n-1)$ since each edge in the complete graph is an LSP of length 1. By Lemma 2.1, the expected number of LSPs of length 2 is $O(n^3 \cdot \frac{\ln^2 n}{n^2}) = O(n \ln^2 n)$. By Lemma 2.2, the expected number of LSPs of length greater than 2 is $O(n^4 \cdot \frac{1}{n^2}) = O(n^2)$ ■ [2].

These results were proved with directed graphs with random assigned weights, but the same analysis could be used on undirected graphs with randomly assigned weights as well. The paper then goes on to prove that using approximation techniques, the expected number of LSPs is $O(n^2)$ with high probability.

2.4 A Better Approach to Heaps for APSP

Consider $\delta = min\{c(u,v) \mid (u,v) \in E\}$. The difference between any two shortest paths $d(u,v)$ and $d(u',v')$ in this graph must be $\geq \delta$. Suppose that there were two shortest paths $d(u,v)$ and $d(u',v')$ whose difference was $< \delta$, then they must differ by at least one edge, and the minimum possible difference there is δ. This is a contradiction. Therefore, the difference between any two shortest paths must be $\geq \delta$.

Using this discovery, we can assert that the `extract-min` function for the priority queue does not actually need to return the absolute minimum of the keys in the queue. Instead, the heap simply needs to return $d(u,v)$ such that $d(u,v) \leq d(u',v') + \delta$ for every $u', v' \in V$, or the smallest distance so that the difference between that distance and everything else is more than δ.

Instead of using a normal heap data structure, we can bucket each distance that is entered into the queue based on δ. We can create a new measure of distance $dist'(u,v) = \lfloor dist(u,v)/\delta \rfloor$. This integer can now be used as an approximation of the actual distance function that preserves the absolute ordering of distances. Therefore, each distance can easily be sorted into buckets.

Let us assume that $\delta \geq n^{-2.5}$, because if $\delta < n^{-2.5}$ then the probability of that happening is $n^{-0.5}$ and then the Fibonacci heap implementation will still be $O(n^2)$ [2]. Thus we are only going to consider the case where $\delta \geq n^{-2.5}$. We now need $n^2 = L$ buckets to fit all of our possible edges, named $B_1, B_2, \ldots, B_L$.

The `heap-insert` operation is easy, since we just insert (u,v) into B_i where $i = \min(dist'(u,v), L)$. This way, all of the distances that are ∞ will be in the L^{th} bucket. The `decrease-key` operation now just removes the item from its current bucket and places it in the appropriate new bucket.

Now, `extract-min` is implemented by scanning each bucket until you find the first non-empty bucket and returning an arbitrary element from that bucket if that bucket isn't B_L. If the bucket is B_L, then we will enter all of those elements into a comparison-based heap and remove the minimum.

This implementation works since all of the inputs are *monotone*, as each input or decrease-key operation does not decrease the min key to a value below the latest element removed.

Thus the overall time spent on all heap operations is $O(N + L)$ where N is the total number of heap operations and L is n^2. We know that $N = O(|LSP| + n^2)$ and that $\mathbb{E}[|LSP|] = O(n^2)$, and so the number of operations becomes $O(n^2)$ both in expectation and with high probability [2].

2.5 Conclusion of All Pairs Shortest Path

We can now conclude an $O(n^2)$ algorithm on a complete graph with random edge weights selected independently from $[0, 1]$. This conclusion is important for the APSP problem because if a graph can be related to a complete graph with random edge weights within a range, then its APSP can be solved in $O(n^2)$ time. Unfortunately, not all graphs fit this model, but this analysis is an improvement on previous work done on the APSP problem.

3 Shortest Replacement Paths

3.1 Introduction to Shortest Replacement Path

Now that we have solved the All Pairs Shortest Path problem, we will move onto solving other graph problems that relate to the APSP problem. One such problem is the shortest replacement path problem. Let's suppose we have already run the APSP algorithm on a graph, but now we want to have a backup. If an edge were to be removed, what is now the shortest path between two vertices?

The shortest replacement path problem is, given a pair of vertices s and t and every edge e on the shortest path between them, calculate the best replacement path that avoids e. The solution to this problem is important in routing network traffic, or regular traffic, to immediately know which detour is best if a router or road suddenly went out or was under repair.

Let $P = \{s = v_1 \to v_2 \to \ldots \to v_k = t\}$ be the shortest path from s to t. For $j > i$, a detour $D(v_i, v_j)$ between v_i and v_j is a simple path from v_i to v_j which does not use any of the other nodes on the path P.

Definition 3.1: Thus the `shortest replacement path` for any edge $e = (v_i, v_{i+1}) \in E(P)$ is the minimum of all paths $s \to v_2 \to \ldots v_j \odot D(v_j, v_k) \odot v_k \to \ldots t$ where $j \leq i$, $i + 1 \leq k$, and $\odot$ denotes concatenation.

To solve the shortest replacement path problem, we just need to calculate all such $D(v_j, v_k)$ for every possible j, k pair satisfying our constraints and compare the resulting path lengths to find the shortest one. The shortest replacement path would not take multiple detours, instead just one extended detour since it is not required to actually visit v_i or v_{i+1}.

3.2 Connection to APSP

The algorithm given by Williams [3] provides one reduction to the APSP problem. We will create a new graph $G' = (V', E')$ where

1. V' includes $V \setminus V(P)$ as well as
   (a) a v_i^{in} and a v_i^{out} for each $v_i \in P$
2. E' includes $E \setminus E(P)$ as well as
   (a) an edge (u, v_i^{in}) for each edge $(u, v_i) \mid u \in G \setminus P$
   (b) an edge (v_i^{out}, u) for each edge $(v_i, u) \mid u \in G \setminus P$

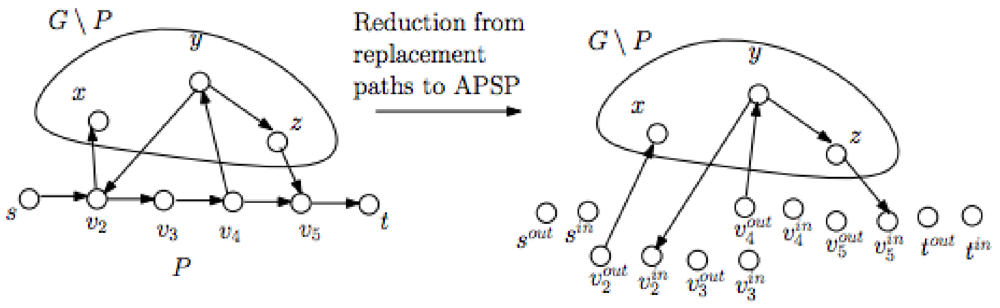

Figure 1: G and G' example [3]

If we compute APSP on G', the shortest path between v_i^{out} and v_j^{in} is exactly the optimal detour $D(v_i, v_j)$ in G. See Figure 1 for an example of such a transformation.

3.3 Analysis of this APSP

Zwick proved that APSP could be solved in undirected graphs with integer edge weights in the range $\{0, 1, \ldots M\}$ in $Mn^{2.38}$ and in directed graphs with integer edge weights in $\{-M, \ldots, M\}$ in $M^{0.68}n^{2.58}$ [8].

Using these APSP conclusions, we assert that the shortest replacement path problem can be solved in $O(\text{APSP} + n^2 \log n)$ and that the APSP problem can be solved in $O(M^{0.68}n^{2.58})$ using rectangular matrix multiplication. The $M^{0.68}n^{2.58}$ dominates the $n^2 \log n$. Thus, there exists a deterministic algorithm for the replacement paths problem that runs in $O(M^{0.68}n^{2.58})$.

Williams' paper continued to discuss methods such as bucketing vertices together, classifying the right path, left path and combinations of them, and considering long detours separately from short detours [3]. Using all of these techniques together, we can find the shortest replacement path. Since we can solve these smaller problems in shorter time, our challenge becomes combining the results.

By bucketing the vertices on the path, Williams reduces the number of vertices in the APSP calculations and along the final replacement path. For instance, all of the vertices along the original path can eliminated except for the origin of the path, the replacement vertices and the destination, or s, v_i, v_{i+1}, and t.

For future reference, ω is the smallest number for which $n \times n$ matrix multiplication can be done in n^ω time, in this paper. Its official definition is the infimum of the set of all x where $n \times n$ matrix multiplication can be done in n^x time, so n^ω $n \times n$ matrix multiplication is not necessarily achievable. We know that $\omega < 2.38$ [9].

The Williams paper concluded that directed graphs with integer weights in $\{-M \ldots M\}$ has an $Mn^{\omega+o(1)}$ algorithm.

Fortunately, advancements in solving the APSP problem can improve the run time of shortest replacement paths. If a $Mn^{\omega+o(1)}$ exists for APSP, then this shortest replacement path algorithm improves on that.

3.4 Conclusion of Shortest Replacement Paths

The Shortest Replacement Path problem is important both in theory and in practice. Williams stated and proved an $Mn^{\omega+o(1)}$ algorithm for finding the shortest replacement path on a directed graph with integer edge weights between $\{-M, \ldots, M\}$. Unfortunately, not all graphs satisfy these constraints, but many do. For instance, graphs with rational edge weights can be modified to fit these constraints, and graphs can be rounded or approximated to fit these constraints. Work within the field of shortest replacement paths is still exciting.

4 Smoothed Analysis of Shortest Paths Algorithms

4.1 Introduction to Smoothed Analysis

We now transition from introducing new algorithms to introducing a type of algorithmic analysis with applications for shortest paths algorithms. Smoothed analysis is a hybrid of worst-case and average-case analysis, introduced by Spielman and Teng for the simplex algorithm [10]. Smoothed analysis addresses why so many algorithms with poor worst-case runtimes are quite performant in practice. Real data is noisy. Worst-case runtimes are often not a reflection of how an algorithms runs on real inputs, and can be constructed from quite contrived inputs that may rarely be given from real data. Average-case runtimes may also not be representative of real runtimes, as a random input may not resemble an actual input to the algorithm.

Smoothed analysis looks at runtime with input subject to slight random Gaussian perturbation. Often the hard input instances are sparse. If the hard instances of a particular problem are surrounded by easier instances, these easier instances can be solved faster and may approximate the hard instances with a corresponding decrease in runtime. If the space of hard input instances is dense, then the smoothed analysis will be the same as the worst-case analysis.

4.2 Smoothed Analysis of SSSP

Closely related to the all pairs shortest paths problem is the single source shortest path problem. What follows is a smoothed analysis of the single source shortest paths problem with directed, non-negative integer weights proposed by Banderier [5]. A common algorithm for this problem is Dijkstra's Algorithm, which is $O(m + n \log n)$. An algorithm by Meyer [6] has been shown to have, with high probability, runtime linear in the problem size $O(m + n)$. An alternative algorithm by Goldberg [7] is the basis for this section, which is worst case $O(m + nK)$, where the input edge weights are K bit integers (in the range $[0, 2^K - 1]$).

Theorem 3.1 (Shortest Paths under Limited Randomness) *Let G be an arbitrary graph, let $c : E \mapsto [0, ..., 2^K - 1]$ be an arbitrary cost function, and let k be such that $0 \leq k \leq K$. Let $\bar{c}$ be obtained by making the last k bits of each edge cost random. Then the single source shortest path problem can be solved in expected time $O(m + n(K - k))$.*

The expected runtime varies linearly from $O(m+n)$ to $O(m+nK)$ corresponding to full randomess and zero randomness respectively.

Proof: Goldberg has shown that his algorithm has runtime

$$O(n + m + \sum_v (K - \log[min\_in\_weight(v)] + 1))$$

where $min\_in\_weight(v)$ is the minimal weight of all incoming edges to node v, $inedges(v)$. Note that all $indeg(v)$ weights of the $inedges(v)$ have their last k bits chosen at random. For an edge e, let $r(e)$ be the number of leading zeros in the random part of e. The expectation of $r(e)$ is

$$E[r(e)] = \sum_{i=1}^{k} i \left(\frac{1}{2}\right)^i \leq 2$$

The smallest possible weight in of the $inedges(v)$ will have a bit representation of all zeros in the nonrandom portion (the first $K - k$ bits), with the random k bit portion having either of all zeros or all zeros with a single 1. This allows a lower bound on $\log[min\_in\_weight(v)]$:

$$\log[min\_in\_weight(v)] \geq k - max\{r(e); e \in inedges(v)\}$$

Assuming a small edge weight of this form, a larger $r(e)$ corresponds to a smaller edge weight, since there highest 1 bit occurs after $K - k + r(e)$ zeros. All edge weights are positive, so

$$max\{r(e); e \in inedges(v)\} \leq \sum_{e \in inedges(v)} r(e)$$

It follows that

$$K - \log[min\_in\_weight(v)] \leq K - k + max\{r(e); e \in inedges(v)\}$$

$$K - \log[min\_in\_weight(v)] \leq K - k + \sum\{r(e); e \in inedges(v)\}$$

Taking the expectation

$$E[K - \log[min\_in\_weight(v)]] \leq K - k + O(indeg(v))$$

gives a time bound of

$$O(n + m + n(K - k) + m) = O(m + n(K - k))$$

4.3 Conclusion to Smoothed Analysis

The All Pairs Shortest Paths algorithm presented earlier had constraints (such as independent, random edge weights with a uniform distribution of the range $[0, 1]$) that may not necessarily model real-world inputs that require fast runtime. The smoothed analysis presented here has much looser constraints, namely integers in a bounded range, that can be determined at runtime. This allows for a shortest paths algorithm that can execute on more realistic input graphs and still have proper guarantees from the runtime analysis.

5 Conclusion

We have presented three main interesting results in graphs algorithms: a fast All Pairs Shortest Paths algorithm, a Shortest Replacement Paths algorithm, and smoothed analysis of a Single Source Shortest Paths algorithm. The first two have practical applications, while the third serves to explain runtimes seen in practice. These graph algorithms work to incorporate randomization and shortest path concepts.

At time of writing, no smoothed analysis of an All Pairs Shortest Path algorithm has been published. Such analysis would constitute interesting future work, as would improvements on certain bounds given by Peres and Williams which are not known to be tight.

References

[1] Camil Demetrescu, Giuseppe Italiano. A New Approach to Dynamic All Pairs Shortest Paths In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing* emph(TOC-03), pages 159-166. ACM Press, 2003.

[2] Yuval Peres, Dmitry Sotnikov, Benny Sudakov, Uri Zwick. All−Pairs Shortest Paths in $O(n^2)$ time with high probability, In *51nd Annual Symposium of Foundations of Computer Science, (FOCS 2010)*, pages 663-672. 2010

[3] Virginia Williams. Faster replacement paths. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms* emph(SODA-11), pages 1337-1346. ACM Press, 2011.

[4] David Karger, Daphne Koller, Steven Phillips. Finding the hidden path: time bounds for all-pairs shortest paths. In *32nd Annual Symposium of Foundations of Computer Science*, *(FOCS 1991)*, pages 560-568. 1991

[5] Cyril Banderier, Kurt Mehlhorn, Rene Beier. Smoothed Analysis of Three Combinatorial Problems. In *Proceedings of the 28th International Symposium Mathematical Foundations of Computer Science 2003*, *MFCS 2003*, pages 198-207. Springer Berlin Heidelberg 2003.

[6] Ulrich Meyer. Shortest-Paths on arbitrary directed graphs in linear Average-Case time. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 797-806. ACM Press, 2001.

[7] Andrew Goldberg. A simple shortest path algorithm with linear average time. In *Proceedings of the 9th European Symposium on Algorithms (ESA '01)*, pages 230-241. Springer Lecture Notes in Computer Science LNCS 2161, 2001.

[8] Uri Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. In *J. ACM*, 49(3):289-317, 2002.

[9] Don Coppersmith, Shmuel Winograd. Matrix Multiplication via Arithmetic Progressions. In *Journal of Symbolic Computation* Volume 9 Issue 3, pages 251-280, 1990.

[10] Daniel Spielman, Shang-Hua Teng. Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing* emph(TOC-01), pages 296-305. ACM Press, 2001.

[11] Seth Pettie, A new approach to all-pairs shortest paths on real-weighted graphs, Theoretical Computer Science, vol. 312, no. 1, pages 47-74, 2004.

[12] Catherine McGeoch, All-pairs shortest paths and the essential subgraph, Algorithmica, vol. 13, pages 426-441, 1995.

[13] Refael Hassin, Eitan Zemel, On shortest paths in graphs with random weights, Mathematics of Operations Research, vol. 10, no. 4, pages 557–564, 1985.

6 Appendix

The All Pairs Shortest Path algorithm of Peres et al. [2]:

```
**Function** apsp (G=(V,E,c))
  init($G$)
  $Q \leftarrow heap()$
  **foreach** $(u, v) \in E$ **do**
      $dist[u, v] \leftarrow c(u, v)$
      $p[u, v] \leftarrow v$
      $q[u, v] \leftarrow u$
      heap-insert($Q, (u, v), dist[u, v]$)
  **while** $Q \neq \emptyset$ **do**
      $(u, v) \leftarrow$extract-min($Q$)
      insert($L[p[u, v], v], u$)
      insert($R[u, q[u, v]], v$)
      **foreach** $w \in L[u, q[u, v]]$ **do**
          examine($w, u, v$)
      **foreach** $w \in R[p[u, v], v]$ **do**
          examine($w, u, v$)
```

```
**Function** init(G=(V,E,c))
  **foreach** $u, v \in V$ **do**
      $dist[u, v] \leftarrow \infty$
      $p[u, v] \leftarrow null$
      $q[u, v] \leftarrow null$
      $L[u, v] \leftarrow \emptyset$
      $R[u, v] \leftarrow \emptyset$
  **foreach** $u \in V$ **do**
      $dist[u, u] \leftarrow 0$
```

```
**Function** examine(u,v,w)
  **if** $dist[u, v] + dist[v, w] < dist[u, w]$ **then**
      $dist[u, w] \leftarrow dist[u, v] + dist[v, w]$
      **if** $p[u, w] = null$ **then**
          heap-insert($Q, (u, w), dist[u, w]$)
      **else**
          decrease-key($Q, (u, w), dist[u, w]$)
      $p[u, w] \leftarrow p[u, v]$
      $q[u, w] \leftarrow q[v, w]$
```