Chapter 1

Fast Estimation of Diameter and Shortest Paths (without Matrix Multiplication)

D. Aingworth[*] C. Chekuri[†] R. Motwani[‡]

1 Introduction

Consider the problem of computing all-pairs shortest paths (APSP) in an unweighted, undirected graph G with n vertices and m edges. The recent work of Alon, Galil, and Margalit [AGM91], Alon, Galil, Margalit, and Naor [AGMN92], and Seidel [Sei92] has led to dramatic progress in devising fast algorithms for this problem. These algorithm are based on formulating the problem in terms of matrices with *small integer* entries and using fast matrix multiplications. They achieve a time bound of $\tilde{O}(n^{\omega})$[1] where ω denotes the exponent in the running time of the matrix multiplication algorithm used. The current best matrix multiplication algorithm is due to Coppersmith and Winograd [CW90] and has $\omega = 2.376$. In contrast, the naive algorithm for APSP performs breadth-first searches from each vertex, and requires time $\Theta(nm)$.

Given the fundamental nature of this problem, it is important to consider the desirability of implementing the algorithms in practice. Unfortunately, fast matrix multiplication algorithms are far from being practical and suffer from large hidden constants in the running time bound. Consequently, we adopt the view of treating these results primarily as indicators of the existence of efficient algorithms and consider the question of devising a purely *combinatorial algorithm* for APSP that runs in time $O(n^{3-\epsilon})$. The (admittedly vague) term "combinatorial algorithm" is intended to contrast with the more algebraic flavor of algorithms based on fast matrix multiplication. To understand this distinction, we believe it is instructive to try and interpret the "algebraic" algorithms in purely graph-theoretic terms even with the use of the simpler matrix multiplication algorithm of Strassen [Str69]. Currently, the best known combinatorial algorithm is due to Feder and Motwani [FM91] that runs in time $O(n^3/\log n)$, yielding only a marginal improvement over the naive algorithm.

We take a step in the direction of realizing the goals outlined above by presenting an algorithm which solves the APSP problem with an *additive* error of 2 in time $O(n^{2.5}\sqrt{\log n})$. This algorithm returns actual paths and not just the distances. Note that the running time is better than $\tilde{O}(n^{\omega})$ when the more practical matrix multiplication algorithm of Strassen [Str69] is used ($\omega = 2.81$) in the algorithms described earlier. Further, as explained below, we also give slightly more efficient algorithms (for *sparse* graphs) for approximating the diameter. Our algorithms are easy to implement, have the desired property of being combinatorial in nature, and the hidden constants in the running time bound are fairly small. While our results are presented only for the case of unweighted, undirected graphs, they can be generalized to the case of undirected graphs with small integer edge weights; the details will be provided in the final version of the paper.

A crucial step in the development of our result was the shift of focus to the problem of computing the diameter of a graph. This is the maximum over all pairs of vertices of the shortest path distance between the vertices. The diameter can be determined by computing all-pairs shortest path (APSP) distances in the graph, and it appears that this is the only known way to solve the diameter problem. In fact, Fan Chung [Chu87] had earlier posed the question of whether there is an $O(n^{3-\epsilon})$ algorithm for finding the diameter without resorting to fast matrix multiplication. The situation with regard to combinatorial algorithms for diameter is only marginally better than in the case of APSP. Basch, Khanna, and Motwani [BKM95] presented a combinatorial algorithm that verifies whether a graph has diameter 2 in time $O\left(n^3/\log^2 n\right)$. A slight adaptation of this algorithm yields a boolean matrix multiplication

[*]Department of Computer Science, Stanford University. Email: donald@cs.stanford.edu. Supported by an NSF Graduate Fellowship and NSF Grant CCR-9357849.

[†]Department of Computer Science, Stanford University. Email: chekuri@cs.stanford.edu. Supported by an OTL grant and NSF Grant CCR-9357849.

[‡]Department of Computer Science, Stanford University. Email: rajeev@cs.stanford.edu. Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Development Award, an OTL grant, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

[1]The notation $\tilde{O}(f(n))$ denotes $O(f(n)\,\mathrm{polylog}(n))$.

algorithm which runs in the same time bound, thereby allowing us to verify that the diameter of a graph is d, for any constant d, in $O\left(n^3/\log^2 n\right)$ time.

Consider the problem of devising a fast algorithm for *approximating* the diameter. It is easy to estimate the diameter within a ratio $1/2$ in $O(m)$ time: perform a breadth-first search (BFS) from any vertex v and let d be the depth of the BFS tree obtained; clearly, the diameter of G lies between d and $2d$. No better approximation algorithm was known for this problem; in fact, it was not even known how to distinguish between graphs of diameter 2 and 4. Our first result is an $O(m\sqrt{n \log n})$ algorithm for distinguishing between graphs of diameter 2 and 4, and this is later extended to obtaining a ratio $2/3$ approximation to the diameter in time $O(m\sqrt{n \log n} + n^2 \log n)$.

Our work suggests several interesting directions for future work, the most elementary being: Is there a combinatorial algorithm running in time $O(n^{3-\epsilon})$ for distinguishing between graphs of diameter 2 and 3? It is our belief that the problem of efficiently computing the diameter can be solved given such a decision algorithm, and our work provides some evidence in support of this belief. In fact, it is our view that the bottleneck in obtaining a faster combinatorial APSP algorithm is precisely the problem of distinguishing graphs of diameter 2 and 3. This also raises the question of whether there is some strong equivalence between the diameter and APSP problems, e.g., that their complexity is the same within poly-logarithmic factors. Finally, of course, removing the additive error from our results remains a major open problem.

The rest of this paper is organized as follows. We begin by presenting some definitions and useful observations in Section 2. In Section 3, we describe the algorithms for distinguishing between graphs of diameter 2 and 4, and the extension to obtaining a ratio $2/3$ approximation to the diameter. Then, in Section 4, we apply the ideas developed in estimating the diameter to obtain the promised algorithm for an additive approximation for APSP. Finally, in Section 5 we present an empirical study of the performance of our algorithm for all-pairs shortest paths.

2 Preliminaries and a Basic Algorithm

We present some notation and a result concerning dominating sets in graphs that underlies all our algorithms. All definitions are with respect to some fixed undirected graph $G(V, E)$ with n vertices and m edges.

Definition 2.1. *The* distance $d(u, v)$ *between two vertices u and v is the length of the shortest path between them.*

Definition 2.2. *The* diameter *of a graph G is defined to be* $\max_{u,v\in G} d(u, v)$.

We will denote the diameter of the graph G by Δ.

Definition 2.3. *The k-neighborhood $N_k(v)$ of a vertex v is the set of all vertices other than v that are at distance at most k from v, i.e.,*

$$N_k(v) = \{u \in V \mid 1 \leq d(u, v) \leq k\}.$$

The degree *of a vertex v is denoted by $d_v = |N_1(v)|$. Finally, we will use the notation $N(v) = N_1(v) \cup \{v\}$ to denote the set of vertices at distance at most 1 from v.*

It is important to keep in mind that the set $N(v)$ contains not just the neighbors of v, but also includes v itself.

Definition 2.4. *For any vertex $v \in V$, we denote by $b(v)$ the depth of a BFS tree in G rooted at the vertex v.*

Throughout this paper, we will working with a parameter s to be chosen later that will serve as the threshold for classifying vertices as being of *low* degree or *high* degree. This threshold is implicit in the following definition.

Definition 2.5. *We define $L(V) = \{u \in V \mid d_u < s\}$ and $H(V) = V \setminus L(V) = \{u \in V \mid d_u \geq s\}$.*

The following is a generalization of the standard notion of a dominating set.

Definition 2.6. *Given a set $A \subseteq V$, a set $B \subseteq V$ is a* dominating set for *A if and only if for each vertex $v \in A$, $N(v) \cap B \neq \emptyset$. That is, for each vertex in $A \setminus B$, one of its neighbors is in B.*

The following theorem underlies all our algorithms.

Theorem 2.1. *There exists a dominating set for $H(V)$ of size $O(s^{-1} n \log n)$ and such a dominating set can be found in $O(m + ns)$ time.*

Remark 2.1. *It is easy to see that choosing a set of $\Theta(s^{-1} n \log n)$ vertices uniformly at random gives the desired dominating set for $H(V)$ with high probability. This theorem is in effect a derandomization of the resulting randomized algorithm.*

Proof. Suppose, to begin with, that $H(V) = V$; then, we are interested in the standard dominating set for the graph G. The problem of computing a minimum dominating set for G can be reformulated as a set cover problem, as follows: for every vertex v create a set $S_v = N(v)$. This gives an instance of the set cover problem $\mathcal{S} = \{S_v \mid v \in V\}$, where the goal is to find a minimum cardinality collection of sets whose union is V. Given any set cover solution $\mathcal{C} \subseteq \mathcal{S}$, the set of vertices corresponding to the subsets in $\mathcal{C}$ forms a dominating set for G of the same size as $\mathcal{C}$. This is because each vertex v occurs in one of the sets $S_w \in \mathcal{C}$, and thus is either in the dominating set itself or has a neighbor therein. Similarly, any dominating set for G corresponds to a set cover for $\mathcal{S}$ of the same cardinality.

The greedy set cover algorithm repeatedly chooses the set that covers the most uncovered elements, and it is known to provide a set cover of size within a factor $\log n$ of the *optimal fractional solution* [Joh74, Lov75]. Since every vertex has degree at least s and therefore the corresponding set S_v has cardinality at least s, assigning a weight of $1/s$ to every set in $\mathcal{S}$ gives a fractional set cover of total weight (fractional size) equal to $s^{-1}n$. Thus, the optimal *fractional* set cover size is $O(n/s)$, and the greedy set cover algorithm must then deliver a solution of size $O(s^{-1}n \log n)$. This gives a dominating set for G of the same size. If we implement the greedy set cover algorithm by keeping the sets in buckets sorted by the number of uncovered vertices, the algorithm can be shown to run in time $O(m)$.

Consider now the case where $H(V) \neq V$. Construct a graph $G' = (V', E')$, adding a set of dummy vertices $X = \{x_i \mid 1 \leq i \leq s\}$, as follows: define $V' = V \cup X$ and $E' = E \cup \{(x_i, x_j) \mid 1 \leq i < j \leq s\} \cup \{(u, x_i) \mid u \in L(V)\}$. Every vertex in this new graph has degree s or higher, so by the preceding argument we can construct a dominating set for G' of size $O(s^{-1}(n+s)\log(n+s)) = O(s^{-1}n \log n)$. Since none of the new vertices in X are connected to the vertices in $H(V)$, the restriction of this dominating set to V will give a dominating set for $H(V)$ of size $O(s^{-1}n \log n)$. Finally, the running time is increased by the addition of the new vertices and edges, but since the total number of edges added is at most $ns + s^2 = O(ns)$, we get the desired time bound. □

In the rest of this paper, we will denote by D a dominating set for $H(V)$ (or, V) constructed as per this theorem.

3 Estimating the Diameter

In this section we will develop an algorithm to find an estimator E such that $2\Delta/3 \leq E \leq \Delta$. We first present an algorithm for distinguishing between graphs of diameter 2 and 4. It is then shown that this algorithm generalizes to the promised approximation algorithm.

3.1 Distinguishing Diameter 2 from 4

The basic idea behind the algorithm is rooted in the following lemma whose proof is straight-forward.

LEMMA 3.1. *Suppose that G has a pair of vertices a and b with $d(a,b) \geq 4$. Then, the BFS tree rooted at a vertex $v \in N(a) \cup N(b)$ will have depth at least 3.*

The algorithm, called Algorithm 2-vs-4, computes BFS trees from a small set of vertices that is guaranteed to contain such a vertex, and so one of these BFS trees will certify that the diameter is more than 2.

Algorithm 2-vs-4

1. **if** $L(V) \neq \emptyset$ **then**
   (a) **choose** $v \in L(V)$
   (b) **compute** a BFS tree from each of the vertices in $N(v)$
2. **else**
   (a) **compute** a dominating set D for $H(V) = V$
   (b) **compute** a BFS tree from each of the vertices in D
3. **endif**
4. **if** all BFS trees have depth 2 **then return** 2 **else return** 4.

We are assuming here (and in all other algorithms) that the sets $L(V)$ and $D(V)$ are provided as a part of the input; otherwise, they can be computed in $O(m+ns)$ time.

THEOREM 3.1. *Algorithm 2-vs-4 distinguishes graphs of diameter 2 and 4, and it has running time $O(ms^{-1}n \log n + ms)$.*

Proof. It is clear that the algorithm outputs 2 for graphs of diameter 2 since in such graphs no BFS tree can have depth exceeding 2. Assume then that G has diameter 4 and fix any pair of vertices $a, b \in V$ such that $d(a,b) \geq 4$. We will show that the algorithm does a BFS from a vertex $v \in N(a) \cup N(b)$ and since, by Lemma 3.1, the depth of the BFS tree rooted at v is at least 3, the algorithm will output 4.

We consider the two cases that can arise in the algorithm.

Case 1: $[L(V) \neq \emptyset]$

If either a or b belong to $N(v)$, then there is nothing to prove. If $b(v) > 2$, then again we have nothing to prove. Therefore, the only case that remains is when $b(v) = 2$ and both a and b are in $N_2(v)$ (see Figure 1). Since $d(a,b) \geq 4$, all paths between a and b have to go through a vertex in $N_1(v)$ which implies that $N(v) \cap (N(a) \cup N(b)) \neq \emptyset$. Further, since we compute a BFS tree from each vertex in $N(v)$, we are guaranteed to have a BFS from a neighbor of a or b, completing the proof. The size of $N(v)$ is at most s, therefore the time to compute the BFS trees is bounded by $O(ms)$.

Case 2: $[L(V) = \emptyset]$

Since D is a dominating set for V, it follows immediately that $D \cap (N(a) \cup N(b)) \neq \emptyset$, establishing the proof of correctness. From Theorem 2.1, we have

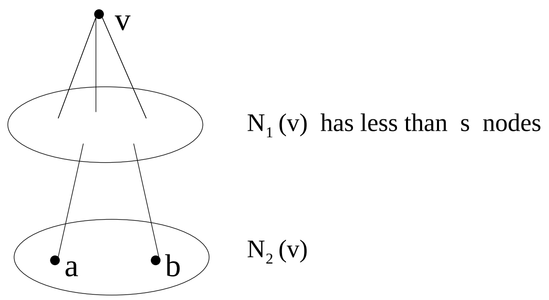

Figure 1: Case 1 in Algorithm 2-vs-4.

$|D| = O(s^{-1} n \log n)$ and this implies a bound of $O(ms^{-1} n \log n)$ on the cost of computing the BFS trees in this case. □

Choosing $s = \sqrt{n \log n}$, we obtain the following corollary.

COROLLARY 3.1. *Graphs of diameter 2 and 4 can be distinguished in $O(m\sqrt{n \log n})$ time.*

3.2 Approximating the Diameter

The basic ideas used in Algorithm 2-vs-4 can be generalized to estimate the diameter in general. Fix any two vertices a and b for which $d(a, b) = \Delta$, where Δ is the diameter of the graph. Suppose we can a find a vertex v in $N_{\Delta/3}(a) \cup N_{\Delta/3}(b)$, then it is clear that $b(v) \geq 2\Delta/3$ and we can use $b(v)$ as our estimator. As before, we will find a small set of vertices which is guaranteed to have a vertex in $N_{\Delta/3}(a) \cup N_{\Delta/3}(b)$. Then, we can compute the BFS tree from each of these vertices and use the maximum of the depths of these trees as our estimator E. The reason for choosing the fraction 1/3 will become apparent in the analysis of the algorithm. In what follows, it will simplify notation to assume that $\Delta/3$ is an integer; in general though, our analysis needs to be modified to use $\lfloor \Delta/3 \rfloor$. Also, we assume that $\Delta \geq 3$, and it is easy to see that the case $\Delta \leq 2$ is easy to handle separately.

A key tool in the rest of our algorithms will be the notion of a *partial-BFS* defined in terms of a parameter k. A k-partial-BFS tree is obtained by performing the usual BFS process up to the point where exactly k vertices (not including the root) have been visited.

LEMMA 3.2. *A k-partial-BFS tree can be computed in time $O(k^2)$.*

Proof. The number of edges examined for each vertex visited is bounded by k since the k-partial-BFS process is terminated when k distinct vertices have been examined. This implies that the total number of edges examined is $O(k^2)$, and that dominates the running time. □

Note that a k-partial-BFS tree contains the k vertices closest to the root, but that this set is not uniquely defined due to ties. Typically, k will be clear from the context and not mentioned explicitly.

DEFINITION 3.1. *Let $PBFS_k(v)$ be the set of vertices visited by a k-partial-BFS from v. Denote by $pb(v)$ the depth of the tree constructed in this fashion.*

The approximation algorithm for the diameter is as follows.

Algorithm Approx-Diameter

1. **compute** an s-partial-BFS tree from each vertex in V
2. **let** w be the vertex with the maximum depth ($pb(w)$) partial-BFS tree
3. **compute** a BFS tree from each vertex in $PBFS_s(w)$
4. **compute** a new graph $\widehat{G}$ from G by adding all edges of the form (u, v) where $u \in PBFS_s(v)$
5. **compute** a dominating set D in $\widehat{G}$
6. **compute** a BFS tree from each vertex in D
7. **return** estimator E equal to the maximum depth of all BFS trees from Steps 3 and 6.

The following lemmas constitute the analysis of this algorithm.

LEMMA 3.3. *The dominating set D found in Step 5 is of size $O(s^{-1} n \log n)$.*

Proof. In $\widehat{G}$, each vertex $v \in V$ is adjacent to all vertices in $PBFS_s(v)$ with respect to the graph G. Since $|PBFS_s(v)| = s$ for every vertex v, the degree of each vertex in $\widehat{G}$ is at least s. From Theorem 2.1, it follows that we can find a dominating set of size $O(s^{-1} n \log n)$. □

LEMMA 3.4. *If $|N_{\Delta/3}(v)| \geq s$ for all $v \in V$, then $D \cap (N_{\Delta/3}(v) \cup \{v\}) \neq \emptyset$ for each vertex $v \in V$.*

Proof. Consider any particular vertex $v \in V$. If v is in D, then there is nothing to prove. Otherwise, since D is a dominating set in $\widehat{G}$, there is a vertex $u \in D$ such that (u, v) is an edge in $\widehat{G}$. If (u, v) is in G, then again we are done since $u \in N(v) \subset N_{\Delta/3}(v)$. The other possibility is that u is not a neighbor of v in G, but then it must be the case that $u \in PBFS_s(v)$. The condition $|N_{\Delta/3}(v)| \geq s$ implies that $PBFS_s(v) \subset N_{\Delta/3}(v)$, which in turn implies that $u \in N_{\Delta/3}(v)$, and hence $u \in D \cap N_{\Delta/3}(v)$. □

The reader should notice the similarity between the preceding lemma and Case 2 in Theorem 3.1. Lemma 3.4 follows from the more general set cover ideas used in the proof of Theorem 2.1 and as such it holds even if we replace $\Delta/3$ by some other fraction of Δ. The more crucial lemma is given below.

LEMMA 3.5. *Let S be the set of vertices v such that $|N_{\Delta/3}(v)| < s$. If $S \neq \emptyset$ then the vertex w found in Step 2 belongs to S. In addition if $b(w) < 2\Delta/3$, then for every vertex v, $PBFS_s(w) \cap N_{\Delta/3}(v) \neq \emptyset$.*

Proof. It can be verified that for any vertex $u \in S$, $pb(u) > \Delta/3$; conversely, for any vertex v in $V \setminus S$, $pb(v) \leq \Delta/3$. From this we can conclude that if S is nonempty, then the vertex of largest depth belongs to S.

Also, for each vertex $u \in S$, we must have $N_{\Delta/3}(u) \subset PBFS_s(u)$. If $b(w) < 2\Delta/3$ then every vertex is within a distance $2\Delta/3$ of w. From this and the fact that $N_{\Delta/3}(w) \subset PBFS_s(w)$, it follows that $PBFS_s(w) \cap N_{\Delta/3}(v) \neq \emptyset$. □

The proof of the above lemma makes clear the reason why our estimate is only within 2/3 of the diameter. Essentially, we need to ensure that the Δ/k neighborhood of w intersects the Δ/k neighborhood of every other vertex. This can happen only if $b(w)$ is sufficiently small. If it is not small enough, we want $b(w)$ itself to be a good estimator. Balancing these conditions gives us $k = 3$ and the ratio 2/3.

THEOREM 3.2. *Algorithm Approx-Diameter gives an estimate E such that $2\Delta/3 \leq E \leq \Delta$ in time $O(ms + ms^{-1}n\log n + ns^2)$. Choosing $s = \sqrt{n\log n}$ gives a running time of $O(m\sqrt{n\log n} + n^2\log n)$.*

Proof. The analysis is partitioned into two cases. Let a and b be two vertices such that $d(a,b) = \Delta$.

Case 1: [*For all vertices v, $|N_{\Delta/3}(v)| \geq s$.*]

If either a or b is in D, we are done. Otherwise from the proof of Lemma 3.4, the set D has a vertex $v \in N_{\Delta/3}(a) \cup N_{\Delta/3}(b)$. Since in Step 6 we compute BFS trees from each vertex in D, one of these is v and $b(v)$ is the desired estimator.

Case 2: [*There exists a vertex $v \in V$ such that $|N_{\Delta/3}(v)| < s$.*]

Let w be the vertex in Step 2. If $b(w) \geq 2\Delta/3$, $b(w)$ is our estimator and we are done. Otherwise from Lemma 3.5, $PBFS_s(w)$ has a vertex $v \in N_{\Delta/3}(a) \cup N_{\Delta/3}(b)$. Since in Step 3 we compute BFS trees from each vertex in $PBFS_s(w)$, one of these is v and $b(v)$ is the desired estimator.

The running time is easy to analyze. Each partial-BFS in Step 1 takes at most $O(s^2)$ time by Lemma 3.2; thus, the total time spent on Step 1 is $O(ns^2)$. Step 2 can be implemented in $O(n)$ time. In Step 3, we compute BFS trees from s vertices, which requires a total of $O(ms)$ time. The time required in Step 4 is dominated by the time required to compute the partial-BFS trees in Step 1. Theorem 2.1 implies that Step 5 requires only $O(n^2 + ns)$ time (note that the graph $\widehat{G}$ could have many more edges than m). By Lemma 3.3, Step 6 takes $O(ms^{-1}n\log n)$ time. Finally, the cost of Step 7 is dominated by the cost of computing the various BFS trees in Steps 3 and 6. The running time is dominated by the cost of Steps 1, 3, and 6, and adding the bounds for these gives the desired result. □

4 Additive Factor Approximations

It is possible to determine not only the diameter, but the all-pairs shortest path distances to within an additive error of 2. The basic idea is that a dominating set, since it contains a neighbor of every vertex in the graph, must contain a vertex that is within distance 1 of any shortest path. Since we can only find a small dominating set for vertices in $H(V)$, we have to treat $L(V)$ vertices differently, but their low degree allows us to manage with only a partial-BFS, which we can combine with the information we have gleaned from the dominating set.

Algorithm Approx-APSP

Comment: Define $G[L(V)]$ to be the subgraph of G induced by $L(V)$.

1. **initialize** all entries in the distance matrix $\widehat{d}$ to infinity
2. **compute** a dominating set D for $H(V)$ of size $s^{-1}n\log n$
3. **compute** a BFS tree from each vertex $v \in D$, and update $\widehat{d}$ with the shortest path lengths for v so obtained
4. **compute** a BFS tree in $G[L(V)]$ for each vertex $v \in L(V)$, and update $\widehat{d}$ with the shortest path lengths for v so obtained
5. **for all** $u, v \in V \setminus D$ **do**
$$\widehat{d}(u,v) \leftarrow \min\{\widehat{d}(u,v), \min_{w\in D}\{\widehat{d}(w,u) + \widehat{d}(w,v)\}\}$$
6. **return** $\widehat{d}$ as the APSP matrix, and its largest entry as the diameter.

Figure 2 illustrates the idea behind this algorithm.

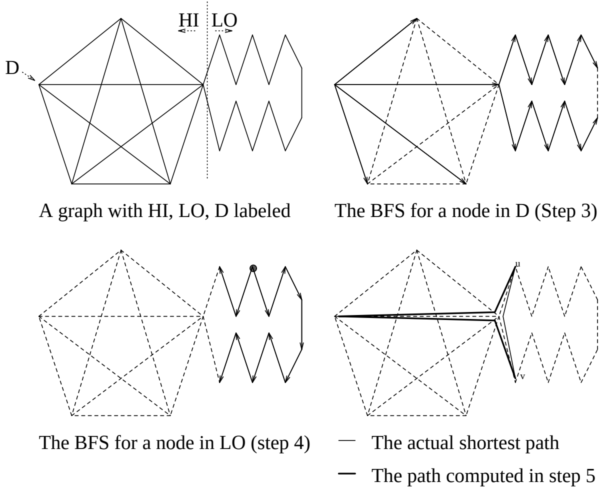

Figure 2: Illustration of Algorithm Approx-APSP.

THEOREM 4.1. *In Algorithm Approx-APSP, for all vertices $u, v \in V$, the distances returned in $\widehat{d}$ satisfy $0 \leq \widehat{d}(u,v) - d(u,v) \leq 2$. Further, the algorithm can be modified to produce paths of length $\widehat{d}$ rather than merely returning the approximate distances. This algorithm runs in time $O(n^2 s + n^3 s^{-1} \log n)$; choosing $s = \sqrt{n \log n}$ gives a running time of $O(n^{2.5}\sqrt{\log n})$.*

Proof. We first show that the algorithm can be easily modified to return actual paths rather than only the distances. To achieve this, in Steps 3 and 4 we can associate with each updated entry in the matrix the path from the BFS tree used for the update. In Step 5, we merely concatenate the two paths from Step 3 the sum of whose lengths determine the minimum value of $\widehat{d}$.

For a vertex u, it is clear that the shortest path distance to any vertex $v \in V$ that is returned cannot be smaller than the correct values, since they correspond to actual paths. To see that they differ by no more than 2, we need to consider three cases:

Case 1: $[u \in D]$

In this case, the BFS tree from v is computed in Step 3 and so clearly the distances returned are correct.

Case 2: $[u \in H(V) \setminus D]$

By the definition of D, it must be the case that u has a neighbor w in D. Clearly, the distances from u and w to any other vertex cannot differ by more than 1, and the distances from w are always correct as per Case 1. The assignment in Step 6 guarantees $\widehat{d}(u,v) \leq \widehat{d}(w,v) + \widehat{d}(w,u) = d(w,v) + 1 \leq d(u,v) + 2$.

Case 3: $[u \in L(V)]$

Fix any shortest path from u to v. Suppose that the path from u to v is entirely contained in $L(V)$; then, $\widehat{d}(u,v)$ is set correctly in Step 4. Otherwise, the path must contain a vertex $w \in H(V)$. If w is contained in D, then the correct distance is computed as per Case 1. Finally, if $w \in H(V) \backslash D$, then D contains a neighbor x of w. Clearly, in Step 6, one of the possibilities considered will involve a path from u to x and a path from x to v. Since the distances involving x are correctly computed in Step 3, this means that $\widehat{d}(u,v) \leq d(x,u) + d(x,v) \leq d(w,u) + d(w,v) + 2 = d(u,v) + 2$.

Finally, we analyze the running time of this algorithm. Step 1 requires only $O(n^2)$ time, and Theorem 2.1 implies that we can perform Step 2 in the stated time bound. Step 3 requires $ms^{-1} n \log n$ for computing the BFS trees. Step 4 may compute as many as $\Omega(n)$ BFS trees, but $G[L(V)]$ only has $O(ns)$ edges and so this requires only $O(n^2 s)$ time. Finally, Step 5 takes all n^2 vertex pairs, and compares them against the $s^{-1} n \log n$ vertices in D. This implies the desired time bound. □

Although the error in this algorithm is 2, it can be improved for the special case of distinguishing diameter 2 from 4 based on the following two observations.

FACT 4.1. *If $u \in H(V)$ is at distance Δ from some vertex v, then $\widehat{d}(u,v) \leq \Delta + 1$.*

Proof. Consider w, the vertex that dominates u. If the algorithm were to have set $\widehat{d}(u,v) > \Delta + 1$ then Step 5 of the algorithm would imply $\widehat{d}(w,v) > \Delta$. Since $\widehat{d}$ is exact for vertices in D, this is not possible. □

FACT 4.2. *Whenever the algorithm reports for some $u \in L(V)$ that $b(u) > 2$, we can verify this in time $O(ns)$ per vertex.*

Thus, by performing a verification for each of the $L(V)$ vertices that report distance over 2, we can improve Algorithm Approx-APSP so that it always performs as well as the diameter approximation algorithms of the previous section. The first fact also appears to be useful in bringing the diameter error down to 1, but unfortunately, the vertices in $L(V)$ cannot be handled as easily for larger diameters.

5 Experimental Results

To evaluate the usefulness of our algorithm, we ran it on two families of graphs and compared the results against a carefully coded algorithm based on breadth-first searches. The algorithm Approx-APSP was tweaked with the following heuristic improvement to Step 5 that avoids many needless iterations: when a node has a neighbor in D, then we copy the distances of its neighbor (since they can differ by at most 1). This algorithm (called Fast Approx-APSP) occasionally has a higher fraction of incorrect entries, but seems to be the fastest way to solve the all-pairs shortest path problem.

	Approx-APSP speedup	Fast Approx-APSP speedup	Approx-APSP accuracy	Fast Approx-APSP accuracy
GB Median	0.59	3.95	0.69	0.53
GB Average	2.44	10.18	0.72	0.47
GB Standard Deviation	0.24	1.73	0.16	0.13
RG Median	0.52	5.30	0.39	0.51
RG Average	0.63	4.75	0.39	0.55
RG Standard Deviation	0.23	1.70	0.14	0.12

Table 1: Summary of Experimental Results

The first family of graphs were random graphs from the $G_{n,m}$ model [Bol85], which are graphs chosen uniformly at random from those with n vertices and m edges. In our experiments, we chose random graphs with n ranging from 10 to 1000, and $2m/n^2$ ranging from 0.03 to 0.90. On these graphs, Fast Approx-APSP runs about 5 times faster than the BFS implementation, and about half of the distances are off by one.

The second family of graphs come from the Stanford GraphBase [Knu93]. We tested all of the connected, undirected graphs from Appendix C in Knuth [Knu93], ignoring edge weights. This is a very heterogeneous family of graphs, including graphs representing highway connections for American cities, athletic schedules, 5-letter English words, and expander graphs, as well as more combinatorial graphs. Thus the results here are quite indicative of practical performance. Although the BFS-based algorithm runs fastest for certain subfamilies of the GraphBase, Fast Approx-APSP outperformed all other algorithms overall.

The results are summarized in Table 1. In the table, GB and RG refer to GraphBase and random graphs, respectively. The speedup numbers indicate the inverse of the ratio of the execution time of the algorithms to that of the carefully coded BFS algorithm. The accuracy refers to the ratio of the total number of *exact* entries in the distance matrix to the total number of entries in the matrix. In both of these families, the accuracy of Approx-APSP could be improved by subtracting 1 in Step 5. This did not seem necessary given that the BFS approach performed about as fast as Approx-APSP, and that Fast- Approx APSP performed faster with roughly 50% accuracy. The numbers indicate that for general graphs where an additive factor error is acceptable, Fast Approx-APSP is the algorithm of choice, and for more specific families of graphs, the parameters can be adjusted for even better performance.

Acknowledgements

We are grateful to Noga Alon for his comments and suggestions, and to Nati Linial for helpful discussions. Thanks also to Michael Goldwasser, David Karger, Sanjeev Khanna, and Eric Torng for their comments.

References

[AGM91] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 569–575, 1991.

[AGMN92] N. Alon, Z. Galil, O. Margalit, and M. Naor.. Witnesses for Boolean Matrix Multiplication and for Shortest Paths. In *Proceedings of the 33nd Annual IEEE Symposium on Foundations of Computer Science*, pages 417–426, 1992.

[BKM95] J. Basch, S. Khanna, and R. Motwani. On Diameter Verification and Boolean Matrix Multiplication. Report No. STAN-CS-95-1544, Department of Computer Science, Stanford University (1995).

[Bol85] B. Bollobás. *Random Graphs.* Academic Press, 1985.

[Chu87] Fan R.K. Chung. Diameters of Graphs: Old Problems and New Results. *Congressus Numerantium*, 60:295–317, 1987.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990.

[FM91] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 123–133, 1991.

[Joh74] D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[Knu93] D.E. Knuth *The Stanford GraphBase: A platform for combinatorial computing.* Addison-Wesley publishing company, 1993.

[Lov75] L. Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13:383–390, 1975

[Sei92] R.G. Seidel. On the all-pairs-shortest-path problem. In *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 745–749, 1992.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.