All-Pairs Shortest Paths for Unweighted Undirected Graphs in $o(mn)$ Time*

Timothy M. Chan†

March 31, 2008

Abstract

We revisit the all-pairs-shortest-paths problem for an unweighted undirected graph with n vertices and m edges. We present new algorithms with the following running times:

$$\begin{cases} O(mn/\log n) & \text{if } m > n\log n\log\log\log n \\ O(mn\log\log n/\log n) & \text{if } m > n\log\log n \\ O(n^2\log^2\log n/\log n) & \text{if } m \le n\log\log n. \end{cases}$$

These represent the best time bounds known for the problem for all $m \ll n^{1.376}$. We also obtain a similar type of result for the diameter problem for unweighted directed graphs.

1 Introduction

In this paper, we address one of the most familiar and often-taught problems in the subject of discrete algorithms: the *all-pairs-shortest-paths* (APSP) problem—namely, given a graph, find the shortest path between every pair of vertices. We mostly focus here on the simplest case of *unweighted, undirected* graphs. The textbook solution of running breadth-first search (BFS) from every vertex has an $O(mn)$ running time, where n and m denote respectively the number of vertices and edges. In terms of n, this bound is cubic. Through the years, improvements have been obtained using more advanced techniques, which can be grouped into two categories briefly summarized below:

- *Algorithms that use fast matrix multiplication*: Galil and Margalit [15, 16] and Seidel [23] gave algorithms that run in $O(n^{2.376})$ time for the unweighted undirected case, using Coppersmith and Winograd's matrix multiplication result [9]. (The running time is $O(n^{2.81})$ if Strassen's more implementable method [26] is used instead.)

  Fast matrix multiplication has also been applied to solve APSP in the unweighted directed case [4, 34] (currently the best time bound is $O(n^{2.575})$ by Zwick [34]), as well as the weighted undirected/directed cases [4, 16, 24, 28, 34] when the weights are small integers (subcubic time bounds hold when weights are at most $n^{0.634}$ [24, 34]).

*A preliminary version appeared in *Proc. 17th ACM-SIAM Sympos. Discrete Algorithms*, pages 514–523, 2006. This work has been supported by NSERC.

†School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada (tmchan@uwaterloo.ca).

- *Algorithms that achieve logarithmic-factor speedups*: At STOC'91, Feder and Motwani [12] obtained (among other things) an algorithm that runs in $O(n^3/\log n)$ time for APSP in the unweighted undirected case, using a graph compression technique.

  Although this algorithm is theoretically inferior to Galil and Margalit's and Seidel's algorithms, log-factor-like speedups yield the currently best (and only) type of results known for APSP in the general case with real-valued weights (the first subcubic time bound was $O(n^3(\log\log n/\log n)^{1/3})$ by Fredman [14], and this was improved in a series of papers [10, 18, 27, 29, 35, 8, 19], culminating in an $O(n^3/\log n)$ time bound by this author [7]). We should also mention that one of the earliest and most well-known examples of log-factor speedup was Arlazarov *et al.*'s (a.k.a. the "four-Russians") algorithm [5] for the simpler Boolean matrix multiplication problem, which originally runs in time $O(n^3/\log n)$ (or, upon closer examination, $O(n^3/\log^2 n)$).

The summary above ignores analysis in one crucial parameter, the number of edges m. In the unweighted undirected case, the more sophisticated methods beat the naive $O(mn)$ method only when $m \gg n^{1.376}$, but arguably the more important case is when the graph is sparse. So what is known about the complexity of the problem for small m besides the naive bound?

Essentially nothing! Many results have been published on shortest paths in sparse weighted graphs under various models (for example, see [1, 17, 21, 22, 30, 31]), and these results improve over the standard Fibonacci-heap implementation of Dijkstra's algorithm, which runs in time $O(n\log n + m)$ for the single-source problem (with positive weights), or $O(n^2\log n + mn)$ for the all-pairs problem. However, the improvements all lie in the first term, not the $O(mn)$ term for APSP. There have also been many average-case results (for one early example, see [25]) and many approximation results (for one particularly relevant example, see Aingworth *et al.*'s and Dor *et al.*'s algorithms [2, 11] for unweighted undirected graphs, which can approximate all shortest-path distances in $O(m^{1/2}n^{3/2}\operatorname{polylog} n)$ time up to an additive error of 2, or faster for larger additive errors). However, we are primarily interested in worst-case exact algorithms in this paper.

Regarding matrix-multiplication-based methods, Yuster and Zwick [33] have recently observed a nontrivial bound for Boolean matrix multiplication for two sufficiently sparse matrices (roughly $O(m^{0.7}n^{1.2} + n^2)$ for $n \times n$ matrices with m nonzero elements), by using known results for dense rectangular matrix multiplication. However, the approach falls short of giving new results for multiplying four or more matrices, let alone, computing powers of matrices as required to solve the APSP problem.

Regarding methods with log-factor speedups, Feder and Motwani [12] actually reported a time bound that is sensitive to both m and n for APSP in unweighted undirected graphs. The bound is $O(mn\frac{\log(n^2/m)}{\log n})$, which, for $m \ll n^{2-\varepsilon}$, offers no asymptotic improvement at all! The known slightly subcubic results for APSP in the general weighted case [8, 14] do not adapt well either for sparse graphs. Generally speaking, log-factor speedups may be possible when there is some amount of redundancy or repetition in the input or computational process (Feder and Motwani [12] exploited the compressibility of dense graphs, while Fredman [14] exploited the reoccurrences of certain small subproblems). The main difficulty seems to be that as the input graph becomes sparser, it is harder to find such repeated "patterns".

We should be careful not to overstate the difficulty, though. For the simpler problem of Boolean matrix multiplication for two (or any constant number of) matrices, it is easy to obtain an $O(mn/\log n)$-time algorithm. It is also not hard to devise an $O(mn/\log n)$ algorithm for the related transitive closure problem for an unweighted directed graph, as noted previously [8]. However, APSP is more involved and it is not immediately clear how one might obtain an $o(mn)$ algorithm. For example, a standard "repeated squaring" approach fails because it could create dense intermediate matrices all too quickly.

In this paper, we present two new APSP algorithms for unweighted undirected graphs. The first runs in $O(mn/\log n)$ time for all $m \gg n\log^2 n$, and the second runs in $O(mn\log\log n/\log n + n^2\log^2\log n/\log n)$ time for all m (see Sections 3 and 4 respectively). These results beat the previous $O(n^{2.376})$ algorithms when $m \ll n^{1.376}$, and subsume the previous $O(n^3/\log n)$ algorithm, as well as the naive $O(mn)$ algorithm, for all m. Both our algorithms are derived from nontrivial combinations of a number of old and new ideas.

Although log-factor-type improvements may appear slight, the first algorithm has the advantage of simplicity: it is very much implementable, avoids matrix multiplication, and has reasonable constant factors (see the experimental data in Section 7). The second algorithm is surprising and shows that speedup is possible even for the sparsest graphs with $m = O(n)$: of course, we cannot break the obvious $\Omega(n^2)$ lower bound if we want to report all shortest-path distances, but what we show is that within the stated subquadratic time bound, it is possible to compute a succinct representation of all shortest-path distances so that each distance can be reported in constant time and each path can be retrieved in time linear in its length.

As a further refinement, we show that a combination of the two algorithms can guarantee $O(mn/\log n)$ running time for a slightly wider range $m \gg n\log n\log\log\log n$.

We also consider the unweighted directed case. Although we are unable to obtain new results for APSP here, we still manage to solve some related problems, like the computation of the *diameter* (largest shortest-path distance), in $O(mn\log^2\log n/\log n)$ randomized time for $m \gg n\log^2 n$ (see Section 6). For unweighted directed planar graphs, we can solve APSP in $O(n^2\log\log n/\log n)$ time.

We can easily modify our algorithms to handle graphs with very small ($O(1)$) positive integer weights. Our algorithms work under the standard RAM model with logarithmic word size. (In Section 7, we note how the algorithms may be adapted in a more powerful word-RAM model or a weaker pointer-machine model.)

2 Preliminaries

In the rest of the paper, $\delta_G(u, v)$ denotes the shortest-path distance from u to v in graph G.

Although our main results are for unweighted undirected graphs, we find it convenient to consider a slightly extended family of graphs. We work with a given undirected graph G whose edges may be unweighted (i.e., have weight 1) or weighted, and all weights and distances are positive integers at most n. Let V be its vertex set, E be the set of all unweighted edges, and V_0 be a subset of "special" vertices. In the next three sections, we assume the following property: for every two vertices s, v, there exists a shortest path π from s to v such that all but at most one edge along π are unweighted, and if π does use a weighted edge, it can only be the first edge su with $u \in V_0$. We call such a path π a *normalized* shortest path, and we call a graph G satisfying this property an *almost unweighted* graph.

Let $n = |V|$, $m = |E|$, and $n_0 = |V_0|$. Without loss of generality, we assume that $m = \Omega(n)$. (At first, the reader may find it helpful to focus on the case of a purely unweighted undirected graph and take $V_0 = \emptyset$ and $n_0 = 0$.)

Note that we can compute a shortest-path tree from each source vertex $s \in V$ in $O(m)$ time:

For an almost unweighted graph, the only weighted edges that need to be considered are $O(n_0)$ weighted edges incident to s. We could apply Dijkstra's algorithm, which runs in linear time if we implement the (monotone) priority queue by bucketing, since distances are small integers. However, a simpler method is to apply BFS: only a slight modification is required for an almost unweighted graph (namely, when generating the i-th level of the shortest-path tree, we just preload all unexplored vertices in $\{u \in V_0 \mid su \text{ has weight } i\}$).

3 First Algorithm

Our first APSP algorithm for unweighted undirected graphs is inspired by the algorithms by Aingworth *et al.* [2] and Dor *et al.* [11] that solved an approximate version of APSP with additive error 2. The basic idea is to treat *high-degree* vertices and *low-degree* vertices separately: roughly speaking, high-degree vertices allow potential for speedup because they are all dominated by a small subset of vertices; low-degree vertices are cheap because the number of incident edges is small. Our treatment of high-degree vertices will differ from Aingworth *et al.*'s, as we want exact distances.

We begin with some easy "word tricks". A subset of $\{1, \ldots, k\}$ can be represented as a bit vector and stored in a sequence of $O(\lceil k/\log n \rceil)$ words each of $\lfloor \alpha \log n \rfloor$ bits for some suitable constant α.

Lemma 3.1 *Given the bit-vector representations of sets $S_1, S_2 \subseteq \{1, \ldots, k\}$, we can*

(i) *compute the bit-vector representations of $S_1 \cup S_2$, $S_1 \cap S_2$, and $S_1 - S_2$ in $O(k/\log n + 1)$ time;*

(ii) *list the elements of S_1 in $O(|S_1| + k/\log n + 1)$ time.*

Proof: (i) is obvious by bitwise or/and/not operations. (ii) follows by repeatedly finding (and turning off) the most significant bit. If these operations are not directly supported, we can precompute the answers to all words (or pairs of words) with $O(n^{2\alpha} \operatorname{polylog} n) = o(n)$ preprocessing time (if $\alpha < 1/2$) and subsequently perform table lookup. □

The key subroutine behind our algorithm is given in the next lemma, which allows us to compute all shortest paths from k starting vertices in time better than the naive $O(km + kn)$ bound. This subroutine however works only under the assumption that the starting vertices are all very close to a common vertex:

Lemma 3.2 *Given a set S of k vertices of distances at most $c = O(1)$ from a fixed vertex s_0, we can compute a representation of a shortest path from every vertex in S to every vertex in V in total time $O(km/\log n + kn + m)$.*

Proof: We first compute the distance from s_0 to every vertex in $O(m)$ time by one BFS. By our assumption about S, $\delta_G(s_0, v) = i$ implies that $i - c \le \delta_G(s, v) \le i + c$. So, we already know approximately the level of each vertex v in the shortest-path tree from any starting vertex $s \in S$. The idea is to use these level estimates to run BFS simultaneously from all starting vertices in S.

More precisely, we compute the sets $S_i[v] = \{s \in S \mid \delta_G(s, v) = i\}$ iteratively as follows:

1. for every i do $A_i = \{v \in V \mid \delta_G(s_0, v) = i\}$
2. for $s \in S$ do $S_0[s] = \{s\}$
3. for $i = 1$ to n do
4.   for $v \in A_{i-c} \cup \cdots \cup A_{i+c}$ do {
5.   if $v \in V_0$ then $S_i[v] = \{s \in S \mid sv \text{ has weight } i\} - \textsc{old}[v]$
6.   for every u adjacent to v in (V, E) do {
7. $\textsc{new} = S_{i-1}[u] - \textsc{old}[v]$
8. $S_i[v] = S_i[v] \cup \textsc{new}$, $\textsc{old}[v] = \textsc{old}[v] \cup \textsc{new}$
9.   for $s \in \textsc{new}$ do $\textsc{ans}[s, v] = (i, u)$

        }

Here, NEW and OLD$[v]$ are subsets of S (OLD$[v]$ contains all $s \in S$ for which $\delta_G(s, v)$ has been found). All sets are initialized to the empty set by default. The table ANS encodes the shortest-path trees—ANS$[s, v]$ holds the distance $\delta_G(s, v)$ as well as the predecessor of v in a shortest path from s to v. Correctness easily follows by induction (since $\delta_G(s, v) = i$ implies that $s \in S_{i-1}[u]$ for some u adjacent to v in (V, E), or sv has weight i).

For the analysis, observe that each vertex plays the role of v at most $2c + 1 = O(1)$ times in the loop in line 4. The total cost of line 5 is $O(kn_0)$. The total number of set-union/difference operations in lines 7 and 8 is thus $O(m)$, and their total cost is $O(m(k/\log n + 1))$ by Lemma 3.1(i) using bit vectors. The total additional cost of line 9 is $O(kn)$ by Lemma 3.1(ii), since each of the $O(kn)$ entries of ANS is set once. The lemma follows. $\square$

We now show that every high-degree vertex is close to some vertex in a small subset, thereby enabling the previous lemma to find shortest paths from all high-degree vertices effectively:

Lemma 3.3 *Consider the unweighted undirected graph (V, E). Given a number d, let H be the subset of all vertices of degree at least d. We can find a subset R of $O(n/d)$ vertices, in $O(m)$ time, such that each vertex in H has distance at most 2 from some vertex of R.*

Proof: Let $N(v)$ denote the set of all vertices adjacent to v. We use a simple greedy strategy: initially unmark all vertices; for each $v \in H$, if all vertices of $N(v)$ are unmarked, then insert v to R and mark all vertices in $N(v)$.

To show correctness, observe that for each $v \in H$, $N(v)$ must intersect $N(r)$ for some $r \in R$, and thus v and r have distance at most 2. Furthermore, $\{N(r) \mid r \in R\}$ is a collection of disjoint subsets each of size at least d, and so $|R| \leq n/d$. $\square$

Remark 3.4 In the original algorithm by Aingworth *et al.* [2], the above lemma was proved by simply letting R be a random sample; this yields actually a better distance guarantee (1 instead of 2) but the size of R is larger by a logarithmic factor.

With the preceding two lemmas, we can put together a complete APSP algorithm:

Theorem 3.5 *We can solve the APSP problem for an almost unweighted, undirected graph G in time $O(mn/\log n + n^2 \log n + mn_0)$.*

Proof: We compute the shortest-path distance $\delta_G(s, v)$ of every pair $s, v \in V$ as follows:

1. Let H and R be as in Lemma 3.3. Add all vertices in V_0 to both H and R.

2. For each $v \in H$, let $r_v \in R$ be a vertex of distance at most 2 from v.

3. Compute the shortest-path distance from every vertex in H to every vertex in V by calling Lemma 3.2 on subsets of the form $S = \{v \in H \mid r_v = r\}$.

4. Form a graph G' by taking the edges in E incident to the vertices of $V - H$, and adding an edge su of weight $\delta_G(s, u)$ for every $s \in V, u \in H$.

5. Compute all shortest-path distances in G' naively by repeated BFSs.

6. For every $s, v \in V$, set $\delta_G(s, v) = \delta_{G'}(s, v)$.

To prove correctness, first note that $\delta_{G'}(s, v) \leq \delta_G(s, v)$ is obvious. For the reverse, let u be the last vertex of $H \cup \{s\}$ on a normalized shortest path from s to v in G. Since the portion of the path from u to v avoids intermediate vertices from H and thus uses only unweighted edges in G', we indeed have $\delta_{G'}(s, v) \leq \delta_G(s, u) + \delta_G(u, v) = \delta_G(s, v)$. Incidentally, this also proves that G' is almost unweighted (with special vertices H).

For the analysis, it suffices to examine steps 3 and 4. Step 3 requires $O(n/d + n_0)$ calls to Lemma 3.2, with a total running time of $O(mn/\log n + n^2 + m[n/d + n_0])$. Step 4 requires n BFSs, with a total running time of $O(dn^2)$, since G' has only $O(dn)$ unweighted edges. Setting $d = \lceil \log n \rceil$ yields the theorem. Note that the shortest-path trees of G themselves can be easily pieced together from the shortest-path trees from steps 3 and 4. $\square$

4 Second Algorithm

Our second algorithm is more complicated but can yield nontrivial running time even when $m \ll n \log^2 n$ (and subquadratic running time when $m = O(n)$). Like in some previous APSP algorithms such as Alon *et al.*'s [4] and Zwick's [34], the basic idea is to consider the case of *short* shortest paths and *long* shortest paths separately: roughly speaking, short shortest paths are easier to compute because fewer steps are involved and operations are done to smaller numbers; long shortest paths can also be found quickly because they all pass through a small subset of vertices. Our treatment of the short case requires a careful analysis (together with a more sophisticated set-representation scheme) that is sensitive to the size of the intermediate sets generated. Our treatment of the long case also requires a new interesting "mod" trick, to keep intermediate numbers small.

We begin with some fancier word tricks. We describe a different scheme to represent a set $S \subseteq \{1, \ldots, k\}$ that has the advantage of input-size sensitivity. Instead of using a bit vector (which always requires k bits, even if S is sparse), we list the elements of S in sorted order, where each element uses $\lceil \log k \rceil$ bits; we then divide the list into $O(\lceil |S| \log k / \log n \rceil)$ words, where each word holds $\lfloor \alpha \log n / \lceil \log k \rceil \rfloor$ elements. We call the resulting sequence of words the "packed-sorted-list" representation of S. (Similar word tricks have been used before; e.g., see [3].)

Lemma 4.1 *Given the packed-sorted-list representations of sets $S_1, \ldots, S_i \subseteq \{1, \ldots, k\}$, we can*

(i) *compute the packed-sorted-list representations of $S_1 \cup S_2$, $S_1 \cap S_2$, and $S_1 - S_2$ in $O((|S_1| + |S_2|) \log k / \log n + 1)$ time;*

(ii) *compute the packed-sorted-list representation of* $S_1 \cup \cdots \cup S_i$ *in* $O((|S_1|+\cdots+|S_i|)\log^2 k/\log n+i)$ *time.*

Proof:

(i) To compute $S_1 \cup S_2$, we imitate the standard linear-time algorithm for merging two sorted lists. Let $B = \lfloor \alpha \log n / \lceil \log k \rceil \rfloor$. We first define some nonstandard word operations: given two sorted sequences w_1, w_2 of B elements, let $M[w_1, w_2]$ be the sequence of the $B-1$ smallest elements of $w_1 \cup w_2$ after duplicates are removed; also let $N_j[w_1, w_2]$ ($j \in \{1, 2\}$) be the number of elements of w_j that appear in the sequence $M[w_1, w_2]$. If these operations are not supported, we can precompute the answers to all pairs of w_1, w_2 with $O(n)$-time preprocessing (if $\alpha < 1/2$) and subsequently perform table lookup.

In each iteration, we grab the next B elements w_1 of S_1 and the next B elements w_2 of S_2, append $M[w_1, w_2]$ to the output sequence, and then skip over $N_1[w_1, w_2]$ elements of S_1 and $N_2[w_1, w_2]$ elements of S_2. Each iteration requires just a constant number of operations and shifts (if shifting is not directly supported, we can again use table lookup). The number of iterations, and thus the running time for union, is $O(\lceil (|S_1| + |S_2|)/B \rceil)$ as desired. Intersection and difference are similar.

(ii) We repeatedly take binary unions in the form of a complete binary tree with i leaves. By part (i), the cost of each level of the tree is $O((|S_1| + \cdots + |S_i|)\log k/\log n)$ plus the number of nodes in the level. So, the total cost of the bottommost $\log k$ levels is $O((|S_1| + \cdots + |S_i|)\log^2 k/\log n + i)$. The remaining levels involve $O(i/k)$ merges only and require a total cost of $O((i/k) \cdot (k \log k/\log n + 1))$, which is subsumed by the earlier cost. $\square$

The following lemma solves the short case:

Lemma 4.2 *Given a number ℓ, we can compute a representation of a shortest path for every pair of distance at most ℓ, in total time $O(mn \log(\ell \log n)/\log n + n^2 \log^2(\ell \log n)/\log n + nn_0)$.*

Proof: Consider any set S of k vertices. Like before, the idea is to run BFS simultaneously from all starting vertices $s \in S$, thereby computing the sets $S_i[v] = \{s \in S \mid \delta_G(s, v) = i\}$ iteratively. This time, we only need to generate the first ℓ levels of the shortest-path trees, but to generate each level, we have to go through all vertices of the graph. The pseudocode is as follows (where as usual, all sets are initially empty):

1. for $s \in S$ do $S_0[s] = \{s\}$
2. for $i = 1$ to ℓ do
3.   for $v \in V$ do {
4.   if $v \in V_0$ then $S_i[v] = \{s \in S \mid sv \text{ has weight } i\} - \textsc{old}_0[v]$
5.   for the j-th vertex u adjacent to v in (V, E) do {
6. $\textsc{new} = S_{i-1}[u] - (S_{i-2}[v] \cup S_{i-1}[v] \cup S_i[v])$
7. $S_i[v] = S_i[v] \cup \textsc{new}$
8. $\textsc{ans}_{ij}[v] = \{(s, i, j) \mid s \in \textsc{new}\}$
9.   if $v \in V_0$ then $\textsc{old}_0[v] = \textsc{old}_0[v] \cup \textsc{new}$

            }

        }

10. for $v \in V$ do $\textsc{ans}[v] = \bigcup_{i,j} \textsc{ans}_{ij}[v]$

Line 6 here takes advantage of the undirectedness of (V, E): if $\delta_G(s,u) = i-1$ and $uv \in E$, then $\delta_G(s,v) \geq i-2$; so $s \in S_{i-1}[u] - (S_{i-2}[v] \cup S_{i-1}[v])$ indeed implies that $\delta_G(s,v) = i$. Also, ANS is now in a different format: if the set $\text{ANS}[v]$ contains the triple (s,i,j), then $\delta_G(s,v) = i$ and the j-th vertex u adjacent to v is a predecessor of v in a shortest path from s to v. Correctness again easily follows by induction.

For the analysis, first note that the total cost of lines 4 and 9 is $O(kn_0)$ by trivially implementing the OLD_0's using bit vectors. Lines 6 and 7 involve a total of $O(\ell m)$ set-union/difference operations, and the total size of the sets involved in these operations is $O(\sum_{i=1}^{\ell} \sum_{uv \in E}(|S_{i-1}[u]| + |S_{i-2}[v]| + |S_{i-1}[v]| + |S_i[v]|)) = O(km)$, since for each v, $\sum_i |S_i[v]| \leq k$ by disjointness. The total cost of lines 6 and 7 is thus $O(km \log k / \log n + \ell m)$ by Lemma 4.1(i) using packed sorted lists.

We use the packed-sorted-list representation also for the sets $\text{ANS}_{ij}[v]$ and $\text{ANS}[v]$; each triple (s,i,j) is encoded as an integer bounded by $O(k\ell \deg(v))$, ordered lexicographically. For each v, line 8 involves a multiple-set union operation for $O(\ell \deg(v))$ sets with total size $\sum_{i,j} |\text{ANS}_{ij}[v]| \leq k$; the cost of this operation is thus $O(k \log^2(k\ell \deg(v)) / \log n + \ell \deg(v))$ by Lemma 4.1(ii). The total cost of line 10 is therefore $O(kn \log^2(k\ell m/n)/\log n + \ell m)$ (by concavity of polylogarithms).

At the end, we need to convert each set $\text{ANS}[v]$ to a vector indexed by s, stored over $O(k \log(\ell \deg(v))/\log n)$ words. This way, given s, we can indeed look up s's entry in $\text{ANS}[v]$ to identify the distance $\delta_G(s,v)$ as well as the predecessor u of v in constant time by simple address calculations. Using the appropriate word operations (like in the proof of Lemma 4.1), the conversion can be accomplished in time linear in the number of words in $\text{ANS}[v]$, i.e., $O(k \log(k\ell \deg(v))/\log n)$. The total additional cost is $O(kn \log(k\ell m/n)/\log n)$.

We conclude that the shortest paths from every vertex $s \in S$ to every vertex of distance at most ℓ can be computed in time $O(km \log k/\log n + \ell m + kn \log^2(\ell m/n)/\log n + kn_0)$. To obtain the lemma, we call this procedure $\lceil n/k \rceil$ times for different size-k subsets S, for a total running time of $O(mn \log k / \log n + \ell mn/k + n^2 \log^2(\ell m/n)/\log n + nn_0)$. The time bound follows by setting $k = \lceil \ell \log n \rceil$ (and noting that $\log^2(m/n) = o(m/n)$). □

We now focus our attention on the long case. The following lemma is well known and has been used in previous papers, e.g., [11, 20, 32, 34].

Lemma 4.3 *Consider the unweighted undirected graph (V,E). Given a number ℓ, we can find a subset R of $O((n/\ell)\log n)$ vertices, in $O(n)$ time, such that with high probability (w.h.p.), for every pair of distance at least ℓ, a shortest path passes through some vertex of R.*

Proof: Basically, we seek a hitting set R for a collection of $O(n^2)$ subsets (paths) where each subset has size at least ℓ. (In the terminology of set systems, R is an "ε-net" with $\varepsilon = \ell/n$.) We can simply let R be a random sample of $c(n/\ell)\ln n$ vertices for $c \gg 2$. Then the probability that a fixed subset is not hit by R is about $[1 - c(1/\ell)\ln n]^{\ell} \leq n^{-c}$. □

We now put together our second APSP algorithm, by using modular arithmetic to map long distances into short distances in a constant number (8) of modified graphs:

Theorem 4.4 *We can solve the APSP problem for an almost unweighted, undirected graph G w.h.p. in time $O(mn \log\log n/\log n + n^2 \log^2 \log n/\log n + mn_0)$.*

Proof: We introduce some shorthand notation: let $[\,x\,]_a := \lfloor (x+a)/8\ell \rfloor$ and $\{\,x\,\}_a := (x+a) \bmod 8\ell$. Our algorithm works as follows:

1. Let R be as in Lemma 4.3. Add all vertices in V_0 to R.
2. Compute the shortest-path distance from every vertex in R to every vertex in V naively by repeated BFSs.
3. For each $v \in V$, let $r_v \in R$ be a vertex of distance less than ℓ from v (if it exists).
4. For each value $a \in \{0, \ell, 2\ell, \ldots, 7\ell\}$:
   (a) Form an almost unweighted graph G'_a (with special vertices R), by taking the edges in E, adding an extra vertex s' for each $s \in V$, and adding a directed edge (s', u) of weight $\{\, \delta_G(s,u) \,\}_a$ for every $s \in V,\ u \in R$.
   (b) Compute the shortest-path distance for every pair of distance at most 4ℓ in G'_a by Lemma 4.2 (which is still applicable even though the weighted edges in G'_a are directed).
5. To compute $\delta_G(s,v)$: if $\delta_{G'_0}(s,v) < \ell$, then set $\delta_G(s,v) = \delta_{G'_0}(s,v)$; otherwise, if $2\ell \leq \{\, \delta_G(s,r_v) \,\}_a < 3\ell$ and $\delta_{G'_a}(s',v) < 4\ell$ for some a, then set $\delta_G(s,v) = [\, (\delta_G(s,r_v) \,]_a \cdot 8\ell + \delta_{G'_a}(s',v) - a$.

We analyze the cost of the algorithm first. Step 2 takes $O(m|R|) = O(m[(n/\ell)\log n + n_0])$ time. Step 4 takes $O(mn\log(\ell\log n)/\log n + n^2\log^2(\ell\log n)/\log n + (n^2/\ell)\log n)$ time. Setting $\ell = \lceil \log^2 n \rceil$ yields the desired time bound. Note that each shortest-path distance can indeed be reported in constant time, and the shortest path itself can be easily retrieved in time linear in its length from the shortest-path trees from steps 2 and 4.

To prove correctness, suppose that $\delta_G(s,v) \geq \ell$ (otherwise, $\delta_G(s,v) = \delta_{G_0}(s,v)$ is clearly correctly found). Let $a \in \{0, \ell, \ldots, 7\ell\}$ be such that $2\ell \leq \{\, \delta_G(s,r_v) \,\}_a < 3\ell$. We need to show that $\delta_{G'_a}(s',v) < 4\ell$ and $\delta_G(s,v) + a = [\, \delta_G(s,r_v) \,]_a \cdot 8\ell + \delta_{G'_a}(s',v)$. In other words, we want to establish the following three facts:

- $[\, \delta_G(s,v) \,]_a = [\, \delta_G(s,r_v) \,]_a$.

  Proof: This follows since $|\delta_G(s,v) - \delta_G(s,r_v)| \leq \delta_G(v,r_v) < \ell$.

- $\{\, \delta_G(s,v) \,\}_a = \delta_{G'_a}(s',v)$.

  Proof of $\geq$: Let u be the last vertex from $R \cup \{s\}$ on a normalized shortest path from s to v in G (if the path is not unique, take the one with the most vertices from R). Then $\delta_G(s,v) = \delta_G(s,u) + \delta_G(u,v)$. Because any shortest path from u to v avoids intermediate vertices from R and uses only edges in E, we have $\delta_G(u,v) < \ell$ by Lemma 4.3. Incidentally, this confirms the existence of r_v (as $\delta_G(s,v) \geq \ell$ implies $s \neq u \in R$). Since $|\delta_G(s,v) - \delta_G(s,r_v)| < \ell$, it follows that $\ell \leq \{\, \delta_G(s,v) \,\}_a < 4\ell$ and $0 \leq \{\, \delta_G(s,u) \,\}_a < 4\ell$. So, $\delta_{G'_a}(s',v) \leq \{\, \delta_G(s,u) \,\}_a + \delta_G(u,v) = \{\, \delta_G(s,v) \,\}_a$.

  Proof of $\leq$: Let $\hat{u} \in R$ be such that $\delta_{G'_a}(s',v) = \{\, \delta_G(s,\hat{u}) \,\}_a + \delta_G(\hat{u},v)$. We already know from above that $\delta_{G'_a}(s',v) \leq \{\, \delta_G(s,v) \,\}_a < 4\ell$. Thus, we also have $\{\, \delta_G(s,\hat{u}) \,\}_a < 4\ell$ and $\delta_G(\hat{u},v) < 4\ell$. Since $|\delta_G(s,v) - \delta_G(s,\hat{u})| \leq \delta_G(\hat{u},v) < 4\ell$, it follows that $\{\, \delta_G(s,v) \,\}_a, \{\, \delta_G(s,\hat{u}) \,\}_a < 4\ell$ implies $[\, \delta_G(s,v) \,]_a = [\, \delta_G(s,\hat{u}) \,]_a$. So, $\delta_G(s,v) \leq \delta_G(s,\hat{u}) + \delta_G(\hat{u},v)$ yields $\{\, \delta_G(s,v) \,\}_a \leq \{\, \delta_G(s,\hat{u}) \,\}_a + \delta_G(\hat{u},v) = \delta_{G'_a}(s',v)$.

- $\delta_{G'_a}(s', v) < 4\ell$.

  PROOF: Already shown above. □

Remark 4.5 The above description technically gives only a Monte Carlo algorithm, but it can be easily made Las Vegas because we can verify whether a given subset R satisfies the requirement of Lemma 4.3: form an almost unweighted graph G' by taking the edges in E, adding an extra vertex s' for each $s \in V$, and adding an edge (s', u) of weight $\delta_G(s, u)$ for every $s \in V$, $u \in R$; then we check, by Lemma 4.2, that for every $s, v \in V$, $\delta_G(s, v) = \ell$ implies $\delta_{G'}(s', v) = \ell$ (so as to guarantee that $\min_{u \in R}(\delta_G(s, u) + \delta_G(u, v)) = \ell$).

We can also derandomize Lemma 4.3, as in [34], by using the greedy algorithm for hitting sets, although this would require an extra $O(n^2 \ell)$ cost. See the appendix for a simpler derandomization of Lemma 4.3, specialized for undirected graphs, that requires only linear time.

We mention one quick application—computing the diameter of sparse unweighted undirected graphs in subquadratic time:

Corollary 4.6 *We can compute the diameter of an unweighted undirected graph G in time $O(mn \log\log n / \log n + n^2 \log^2 \log n / \log n)$.*

Proof: For each $a \in \{0, \ell, \ldots, 7\ell\}$ and $v \in V$, it suffices to search for the maximum $\rho_{a,v}$ of $[\,\delta_G(s, r_v)\,]_a \cdot 8\ell + \delta_{G'_a}(s', v)$ over all $s \in V$ such that $2\ell \leq \{\,\delta_G(s, r_v)\,\}_a < 3\ell$ and $\delta_{G'_a}(s', v) < 4\ell$.

For each $r \in R$, we first generate a bit vector for the set $S_{a,r}$ of all $s \in V$ such that $2\ell \leq \{\,\delta_G(s, r)\,\}_a < 3\ell$ and $[\,\delta_G(s, r)\,]_a$ is maximal. This takes $O(n|R|) = O((n^2/\ell) \log n)$ total time. For each $v \in V$, Lemma 4.2 gives us a vector storing all values $\delta_{G'_a}(s', v)$ at most 4ℓ over $s \in V$. By scanning this vector and $S_{a,r}$ using appropriate word operations, we can determine $\rho_{a,v}$ in time linear in the number of words $O(n \log \ell / \log n)$. The total additional time is thus $O(n^2 \log\log n / \log n)$. □

5 A Hybrid Algorithm

We now show how the algorithms in the previous two sections can be combined.

Theorem 5.1 *We can solve the APSP problem for an unweighted undirected graph G in time $O(mn/\log n + n^2 \log\log\log n)$.*

Proof: Observe that in the first algorithm, we can perform step 5 by recursion. The running time of the two algorithms can be summarized as follows:

$$\begin{aligned} T(m, n, n_0) &= O(mn/\log n + n^2 + mn/d + mn_0) \;+\; T(dn, n, n/d + n_0) \\ T(m, n, n_0) &= O(mn \log\log n / \log n + n^2 \log^2 \log n / \log n + mn_0). \end{aligned}$$

Let $T(m) := T(m, n, 2n^2/m)$. By choosing $d = m/(2n)$, we then get

$$\begin{aligned} T(m) &= O(mn/\log n + n^2) \;+\; T(m/2) \\ T(n \log n / \log\log n) &= O(n^2). \end{aligned}$$

By running the first algorithm multiple times and then finishing with the second algorithm, we therefore obtain $T(m) = O\left(mn/\log n + n^2 \log\left\lceil \frac{m}{n \log n / \log\log n} \right\rceil\right)$. If $m > n \log n \log\log\log n$, the first term dominates; otherwise, the bound is at most $O(n^2 \log\log\log n)$. □

6 The Directed Case?

We next examine the case when the given graph G is unweighted and directed. Our first algorithm cannot be adapted here because there is no adequate version of Lemma 3.3 for directed graphs. Therefore, we consider modifying the second algorithm. We show that speedup is possible in the short case by a variation of Lemma 4.2. Lemma 4.3 clearly still works. However, the mod trick from Theorem 4.4 doesn't, because it heavily relies on symmetry of the distance function. So, we cannot solve APSP but are able to solve some APSP-related problems, like diameter, using a subtraction trick in place of mod.

We begin with the substitute for Lemma 4.2:

Lemma 6.1 *For an almost unweighted, directed graph G, given a number ℓ, we can compute a representation of the shortest paths for all pairs of distances at most ℓ in G', in $O(mn\log^2(\ell\log n)/\log n + \ell n^2\log(\ell\log n)/\log n + n^2)$ time.*

Proof: Consider any set S of k vertices. We compute the sets $S_i[v] = \{s \in S \mid \delta_G(s,v) = i\}$ as follows:

1. for $s \in S$ do $S_0[s] = \{s\}$
2. for $i = 1$ to ℓ do
3.   for $v \in V$ do {
4. $\text{NEW} = \left(\bigcup_{u:\,(u,v)\in E} S_{i-1}[u]\right) \cup \{s \in S \mid (s,v) \text{ has weight } i\} - \text{OLD}[v]$
5. $S_i[v] = \text{NEW}$, $\text{OLD}[v] = \text{OLD}[v] \cup \text{NEW}$
6.   for every u adjacent to v in (V,E) do {
7.   for $s \in \text{NEW} \cap S_{i-1}[u]$ do $\text{ANS}[s,v] = (i,u)$
8. $\text{NEW} = \text{NEW} - S_{i-1}[u]$

        }

    }

The main difference with the algorithm of Lemma 4.2 is that each $S_i[v]$ is now computed using a single multiple-set union operation (line 4) instead of several binary unions; the reason is that in the directed case, we need to maintain the $\text{OLD}[v]$ sets and and their sizes could be large ($O(k)$). The ANS table is also computed differently.

The total sizes of the $O(\ell m)$ sets involved in the unions in line 4 is $O(\sum_{i=1}^{\ell}\sum_{(u,v)\in E}|S_{i-1}[u]|) = O(km)$, since for each v, $\sum_i |S_i[v]| \le k$ by disjointness. The total cost of these unions is thus $O(km\log^2 k/\log n + \ell m)$ using packed sorted lists.

The $O(\ell n)$ set-difference operations in line 4 and set-union operations in line 5 take $O(\ell n \cdot k\log k/\log n)$ time. In lines 7 and 8, the total sizes of the sets involved in the $O(\ell m)$ set-intersection/difference operations is $O(\sum_{i=1}^{\ell}\sum_{(u,v)\in E}(|S_{i-1}[u]| + |S_i[v]|)) = O(km)$; the cost of these operations is thus $O(km\log k/\log n + \ell m)$. The total cost of the loop in line 7 is $O(kn)$, since each ANS entry is set once.

We conclude that the shortest paths from all $s \in S$ to all vertices of distances at most ℓ can be computed in $O(km\log^2 k/\log n + \ell m + \ell n \cdot k\log k/\log n + kn)$ time. To obtain the lemma, we multiply the time bound by n/k and set $k = \lceil \ell\log n\rceil$, as before. $\square$

We now put together a diameter algorithm, using subtraction to reduce long distances to short distances.

Theorem 6.2 *We can compute the diameter of an unweighted directed graph w.h.p. in time $O(mn \log^2 \log n / \log n + n^2 \log n / \log \log n)$.*

Proof: We solve a slightly more general problem: for each $s \in V$, compute its *eccentricity* $\rho_s = \max_{v \in V} \delta_G(s, v)$.

1. Let R be as in Lemma 4.3.
2. Compute the shortest-path distance from every vertex in V to every vertex in R naively by repeated BFSs (in the transposed graph).
3. For each $s \in V$, let $a_s = \max_{u \in R} \delta_G(s, u)$.
4. Build a weighted graph G' from G, by adding an extra vertex s' for each $s \in V$, and adding an edge (s', u) of weight $\max\{\delta_G(s, u) - (a_s - \ell), 0\}$ for every $s \in V$, $u \in R$.
5. Compute the shortest-path distance for every pair of distance at most 2ℓ in G' by Lemma 6.1.
6. For every $s, v \in V$: if $\delta_{G'}(s, v) < \ell$, then set $d[s, v] = \delta_{G'}(s, v)$; otherwise, if $\delta_{G'}(s', v) < 2\ell$, then set $d[s, v] = \delta_{G'}(s', v) + (a_s - \ell)$. For each $s \in V$, set $\rho_s = \max_{v \in V} \max\{d[s, v], a_s\}$.

We analyze the running time first. Step 2 takes $O(m|R|) = O((mn/\ell) \log n)$ time. Step 5 takes $O(mn \log^2(\ell \log n)/ \log n + \ell n^2 \log(\ell \log n)/ \log n + n^2)$ time. Setting $\ell = \lceil \log^2 n / \log^2 \log n \rceil$ yields the desired time bound.

To prove correctness, suppose that $\delta_G(s, v) \geq \ell$ (otherwise $\delta_G(s, v) = \delta_{G'}(s, v)$ is clearly correctly found). It suffices to establish the following two facts (since $\rho_s \geq a_s$ obviously implies $\rho_s = \max_{v \in V} \max\{\delta_G(s, v), a_s\}$):

- $\max\{\delta_G(s, v), a_s\} = \max\{\delta_{G'}(s', v) + (a_s - \ell), a_s\}$.

  Proof of $\leq$: Let $\hat{u} \in R$ be such that $\delta_{G'}(s', v) = \max\{\delta_G(s, \hat{u}) - (a_s - \ell), 0\} + \delta_G(\hat{u}, v)$. Then $\delta_G(s, v) \leq \delta_G(s, \hat{u}) + \delta_G(\hat{u}, v) \leq \delta_{G'}(s', v) + (a_s - \ell)$.

  Proof of $\geq$: Let u be the last vertex from R on a shortest path from s to v in G (if the path is not unique, take the one with the most vertices from R). Then $\delta_G(s, v) = \delta_G(s, u) + \delta_G(u, v)$, and $\delta_G(u, v) < \ell$ by Lemma 4.3. So, $\delta_{G'}(s', v) \leq \max\{\delta_G(s, u) - (a_s - \ell), 0\} + \delta_G(u, v) \leq \max\{\delta_G(s, v) - (a_s - \ell), \ell\}$, implying $\delta_{G'}(s', v) + (a_s - \ell) \leq \max\{\delta_G(s, v), a_s\}$.

- $\delta_{G'}(s', v) < 2\ell$.

  Proof: As above, $\delta_{G'}(s', v) \leq \max\{\delta_G(s, u) - (a_s - \ell), 0\} + \delta_G(u, v) < 2\ell$, since $\delta_G(s, u) \leq a_s$ and $\delta_G(u, v) < \ell$. □

We can also solve another special case of the APSP problem—that of finding the shortest paths of selected pairs instead of all pairs. Here, the algorithm is much more straightforward:

Theorem 6.3 *Given $O(m)$ pairs in an unweighted directed graph, we can compute a representation of the shortest path for every such pair w.h.p. in total time $O(mn \log^2 \log n / \log n + n^2 \log n / \log \log n)$.*

Proof: We first compute all distances at most ℓ by Lemma 6.1. Let R be the subset from Lemma 4.3. We compute the shortest path from every vertex in V to every vertex in R, and from every vertex in R to every vertex in R, by repeated BFSs in $O(m|R|) = O((mn/\ell)\log n)$ time. For each given pair (s, v) with $\delta_G(s, v) > \ell$, set its distance to be $\min_{u \in R}(\delta_G(s, u) + \delta_G(u, v))$; this step costs $O(m|R|)$ as well. The theorem follows by setting $\ell = \left\lceil \log^2 n / \log^2 \log n \right\rceil$. $\square$

Remark 6.4 As in Remark 4.5, we can make the above algorithms Las Vegas or, with an extra $O(n^2 \ell)$ cost, deterministic.

We can also consider APSP for other special cases, for example, planar graphs in the directed case. Here, we note a different approach to beating $O(mn)$, which works for any graph families that possess good separators.

Theorem 6.5 *We can solve the APSP problem for an unweighted directed planar graph G in time $O(n^2 \log\log n / \log n)$.*

Proof: By a multiple-cluster version of the planar separator theorem (e.g., [13]), we can partition V into a *separator* subset R of size $O(n/\sqrt{\ell})$ and *cluster* subsets $V_1, \ldots, V_{O(n/\ell)}$ each of at most size ℓ, such that vertices of V_i can only be adjacent to vertices of $V_i \cup R_i$ for some subset $R_i \subseteq R$ of size $O(\sqrt{\ell})$, in the undirected version of G. The subsets can be constructed in near-linear time.

We first compute the shortest-path distance from every vertex in R to every vertex in V, and vice versa, by repeated BFSs in $O(n|R|) = O(n^2/\ell)$ time.

For every $s, v \in V - R$, we next compute the distance of the shortest path from s to v that avoids intermediate vertices from R. Since such a path must stay entirely inside one cluster (in particular, it exists only if s and v are in the same cluster), this step can be carried out by solving an APSP subproblem in the subgraph induced by each V_i. Each subproblem is solvable naively in $O(\ell^2)$ time, for a total of $O(n/\ell \cdot \ell^2) = O(n\ell)$.

For every $s, v \in V - R$, it remains to compute the distance of the shortest path from s to v that passes through some vertex of R. Say $v \in V_i$. This distance is equal to $\min_{u \in R_i : \delta_G(u,v) \le \ell}(\delta_G(s, u) + \delta_G(u, v))$, as we can take u to be the last vertex in R along the shortest path from s to v (so that the portion of the path from u to v stays inside one cluster and thus has length at most ℓ). Let $\textsc{ans}[s, v]$ denotes the vertex u attaining the minimum.

Fix a vertex s and a cluster index i. Let $\textsc{ans}_i[s]$ be a vector holding $\textsc{ans}[s, v]$ for all $v \in V_i$; the vector can be encoded in $O(\ell \log \ell)$ bits, i.e., $O(\ell \log \ell / \log n)$ words. Permute the vertices $u_1, \ldots, u_{O(\sqrt{\ell})}$ of R_i so that $\delta_G(s, u_1) \le \delta_G(s, u_2) \le \delta_G(s, u_3) \le \cdots$, and let $d_j = \min\{\delta_G(s, u_{j+1}) - \delta_G(s, u_j), \ell\}$. Let $\textsc{code}_i[s]$ be an encoding of the permutation along with the numbers $d_1, \ldots, d_{O(\sqrt{\ell})}$; the encoding requires only $O(\sqrt{\ell} \log \ell)$ bits, which can be made smaller than $\alpha \log n$ by setting $\ell = \Theta(\log^2 n / \log^2 \log n)$.

We claim that $\textsc{ans}_i[s]$ is completely determined by $\textsc{code}_i[s]$ and the distances between R and V_i. Indeed, if $v \in V_i$ and $\delta_G(u_j, v), \delta_G(u_k, v) \le \ell$ with $j < k$, we have $\delta_G(s, u_j) + \delta_G(u_j, v) < \delta_G(s, u_k) + \delta_G(u_k, v)$ iff $\delta_G(u_j, v) - \delta_G(u_k, v) < \delta_G(s, u_k) - \delta_G(s, u_j)$, iff $\delta_G(u_j, v) - \delta_G(u_k, v) < d_j + \cdots + d_{k-1}$, since the left-hand side is known to be at most ℓ.

As initialization, we precompute the mapping from $\textsc{code}_i$ to $\textsc{ans}_i$ for each i; this preprocessing takes $O(n^{1+\alpha} \operatorname{polylog} n)$ time. For each $s \in V - R$, we can sort $\delta_G(s, u)$ over all $u \in R$ in $O(n/\sqrt{\ell})$ time by radix sort (assuming $n/\sqrt{\ell} \gg n^{\varepsilon}$), and subsequently compute $\textsc{code}_i[s]$ for all indices i in $O(n/\ell \cdot \sqrt{\ell})$ time. We can then determine $\textsc{ans}_i[s]$ for all i by table lookup in $O(n/\ell \cdot \ell \log \ell / \log n)$ time.

The total time per vertex s is $O(n/\sqrt{\ell}+n\log\ell/\log n)$. For the above choice of ℓ, the overall running time becomes $O(n^2\log\log n/\log n)$. (As before, the shortest paths themselves can be retrieved easily.) $\square$

7 Discussion

Other models. Although we have used bitwise-logical or table-lookup operations, they can be avoided in some of our algorithms. Often, we can simulate a batch of table lookups on a pointer machine: we gather the arguments involved in the lookups, perform a radix-sort variant to detect duplicates, and compute the answer directly for each distinct argument, skipping over duplicates. This idea is formalized in the lemma below. (The idea is not exactly new; e.g., see [6] for a more sophisticated form of this approach.)

We say that a list is a *purged* list if items of the list whose values are identical are made to point to a common record holding that value.

Lemma 7.1 *On a pointer machine:*

(i) *We can purge any list of n ℓ-bit strings in $O(\ell n+2^\ell)$ time.*

(ii) *Given a purged list $\langle x_1,\ldots,x_n\rangle$ of ℓ-bit strings and a purged list $\langle y_1,\ldots,y_n\rangle$ of m-bit strings, we can form a purged list $\langle x_1y_1,\ldots,x_ny_n\rangle$ in $O(n+2^{\ell+m})$ time.*

(iii) *Given a purged list $\langle x_1,\ldots,x_n\rangle$ of ℓ-bit strings and a function $f:\{0,1\}^\ell\rightarrow\{0,1\}^m$ evaluable in c time, we can form a purged list $\langle f(x_1),\ldots,f(x_n)\rangle$ in $O(n+(c+m)2^\ell+2^m)$ time.*

Proof:

(ii) We use a form of a 2-pass bucket/radix sort. For each i, we create an item i pointing to x_i and y_i. For each $x\in\{0,1\}^\ell$, we create a bucket L_x containing all items i with $x_i=x$. This step takes $O(n+2^\ell)$ total time. For each L_x and each $y\in\{0,1\}^\ell$, we create a bucket $L_{x,y}$ containing all items i in L_x with $y_i=y$. This step takes $O(|L_x|+2^m)$ time per x, for a total time of $O(n+2^{\ell+m})$. We can then make all items in $L_{x,y}$ point to a common record representing xy.

(i) Just apply the concatenation routine from (i) $\ell-1$ times, starting with lists of 1-bit strings.

(iii) Let A be the list of all strings in $\{0,1\}^\ell$. Build a list B of the same cardinality where if item j of A holds x, item j of B holds $f(x)$. The list can be formed in $O(c2^\ell)$ time. We purge B by (i) in $O(m2^\ell+2^m)$ time. By pointer redirections, we can then make x point to $f(x)$ and obtain the purged list $\langle f(x_1),\ldots,f(x_n)\rangle$ in $O(n)$ time. $\square$

As a simple illustration of the above approach, we can obtain a pointer-machine version of the four-Russians Boolean matrix multiplication algorithm [5]. We are not aware of any previous mention of this observation.

Observation 7.2 *The four-Russians algorithm for multiplying two $n\times n$ Boolean matrices can be implemented in $O(n^3/\log^2 n)$ time on a pointer machine.*

Proof: We first give a quick reinterpretation of the four-Russians algorithm on the traditional RAM. Let $w = \lfloor \alpha \log n \rfloor$ for a sufficiently small α. Note that a $w \times n$ submatrix can be stored in n words, each holding a column. The key ideas can be summarized by the following two subroutines:

1. We can multiply a $w \times w$ matrix A with a $w \times n$ matrix B in $O(n)$ time: For each possible w-bit column vector x, we precompute Ax; this preprocessing takes $O(n^\alpha \operatorname{polylog} n)$ time. Afterwards, we can determine the i-th column of the output, i.e., the product of A with the i-th column of B, by table lookup in $O(1)$ time for each i.

2. We can compute the bitwise-and of two $n \times w$ matrices in $O(n)$ time: We simply perform n bitwise-and operations on n pairs of words in parallel.

Applying subroutines 1 and 2 $O(n/w)$ times, we can immediately multiply a $w \times n$ matrix with an $n \times n$ matrix in $O(n^2/w)$ time. Repeating this $O(n/w)$ times, we can then multiply two $n \times n$ matrices in $O(n^3/w^2)$ time on the RAM.

Now, to implement the above algorithm on a pointer machine, we first represent each $w \times n$ submatrix as a purged list of n w-bit columns viewed as w-bit strings; this preprocessing requires $O(nw)$ time by Lemma 7.1(i) for each of the $O(n/w)$ submatrices of the second matrix. Subroutine 1 can be implemented in $O(n)$ time by Lemma 7.1(iii) (with f mapping w-bit columns x to Ax, $\ell = m = \lfloor \alpha \log n \rfloor$, and $c = O(\operatorname{polylog} n)$). Subroutine 2 can be implemented in $O(n)$ time by applying Lemma 7.1(ii) to purge the n pairs of words and then applying Lemma 7.1(iii) (with f mapping xy to the bitwise-and of x and y, $\ell = 2\lfloor \alpha \log n \rfloor$, $m = \lfloor \alpha \log n \rfloor$, and $c = O(\log n)$). $\square$

We can implement at least our first algorithm on the pointer machine by a similar approach. Observe that the table-lookup/bitwise-logical operations in this algorithm occurs in $O(m)$ rounds each of $O(n/\log n)$ operations, since the pseudocode in Lemma 3.2 requires $O(m)$ sequential steps but the $O(n/d)$ applications of Lemma 3.2 may be executed in parallel. In each round, we gather and purge the $O(n/\log n)$ word pairs involved in these operations by Lemma 7.1(ii), and obtain the answers to the operations by Lemma 7.1(iii) in $O(n/\log n)$ time. The total time remains $O(mn/\log n)$ for $m \gg n \log^2 n$.

Although the above might be messy to do in practice, the point we want to emphasize is that our log-factor-like speedups are obtained not because we "cheat" using the bit-level parallelism afforded by the word RAM, but rather because there are indeed repetitions of small subproblems.

On the other hand, if we are allowed to exploit the full power of the word RAM, we can get better time bounds for a larger word size $w > \log n$. Our first algorithm actually runs in $O(mn/w)$ time for all $m \gg nw^2$, using commonly available operations (bitwise-logical and most-significant-bit). Our second algorithm runs in $O((mn \log w)/w + (n^2 \log^2 w)/w)$ time but requires a constant number of nonstandard (non-AC^0) operations.

Experimental results. We have implemented one version of the first algorithm, to illustrate that it is indeed simple enough to be of practical value. This implementation is done in C++ (on a Sun Ultra 10), using operations on `long long int` (with $w = 64$) and the simpler sampling strategy noted in Remark 3.4 (with a sample size of $0.015n$). In Figure 1, we compare the running times of our algorithm and the naive $O(mn)$ algorithm for "random" unit-disk graphs with $n = 1800$ (vertices are uniformly distributed points inside a square, and two vertices are adjacent if their distance is at most one). The new algorithm works well especially for larger densities; for $m \approx 170000$, it is more than 6 times faster than the naive algorithm.

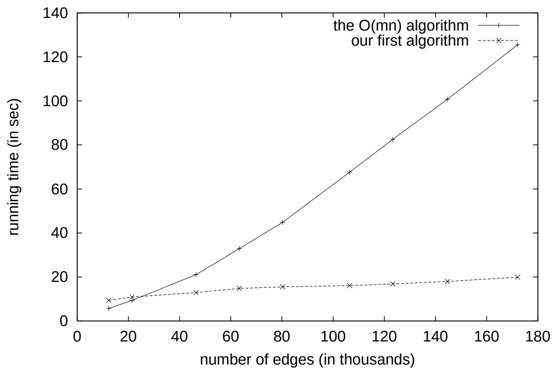

Figure 1: Experimental results for graphs with 1800 vertices.

Our implementation is not optimized; fine-tuning of the sample-size parameter or alternative strategy for choosing R may speed up the algorithm a little. We should mention that for many families of random graphs, the diameter Δ tends to be low (the "small-world phenomenon"), and we have found that a simpler algorithm with $O(mn\Delta/w)$ running time also works well (though it is still slower than our algorithm for the above unit-disk graphs). Of course, this simpler algorithm has very poor worst-case performance (as Δ could be as large as $\Omega(n)$).

Open problems. It seems difficult to beat the $O(mn/\log n)$ time bound. Even for the simpler Boolean matrix multiplication problem, without using "algebraic" techniques (like Strassen's), we don't know of a $o(mn/\log n)$ algorithm for any range of values of $m \ll n^2/\log n$. With or without algebraic techniques, no $o(mn/\log n)$ algorithm is known, for any value of $m \ll n^2/\log n$, to compute, say, the fifth power of an $n \times n$ Boolean matrix with m nonzero entries (a problem that reduces to APSP).

Remaining open problems include finding $o(mn)$ APSP algorithms for general unweighted directed graphs, or weighted undirected graphs with large integer or real positive weights. We can also consider further special cases, for example, finding $o(n^2)$ APSP algorithms for real-weighted planar graphs.

References

[1] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan. Faster algorithms for the shortest path problem. *J. ACM*, 37:213–223, 1990.

[2] D. Aingworth, C. Chekuri, P. Indyk, and R. Motwani. Fast estimation of diameter and shortest paths (without matrix multiplication). *SIAM J. Comput.*, 28:1167–1181, 1999.

[3] S. Albers and T. Hagerup. Improved parallel integer sorting without concurrent writing. *Inf. Comput.*, 136:25–51, 1997.

[4] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *J. Comput. Sys. Sci.*, 54:255–262, 1997.

[5] V. L. Arlazarov, E. C. Dinic, M. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Math. Dokl.*, 11:1209–1210, 1970.

[6] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proc. 30th ACM Sympos. Theory Comput.*, pages 279–288, 1998.

[7] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proc. 39th ACM Sympos. Theory Comput.*, pages 590–598, 2007.

[8] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50:236–243, 2008.

[9] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symbolic Comput.*, 9:251–280, 1990.

[10] W. Dobosiewicz. A more efficient algorithm for the min-plus multiplication. *Int. J. Computer Math.*, 32:49–60, 1990.

[11] D. Dor, S. Halperin, and U. Zwick. All pairs almost shortest paths. *SIAM J. Comput.*, 29:1740–1759, 2000.

[12] T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. Sys. Sci.*, 51:261–272, 1995.

[13] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM J. Comput.*, 16:1004–1022, 1987.

[14] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM J. Comput.*, 5:49–60, 1976.

[15] Z. Galil and O. Margalit. All pairs shortest distances for graphs with small integer length edges. *Inf. Comput.*, 134:103–139, 1997.

[16] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *J. Comput. Sys. Sci.*, 54:243–254, 1997.

[17] T. Hagerup. Improved shortest paths on the word RAM. In *Proc. 27th Int. Colloq. Automata, Languages, and Programming*, Lect. Notes Comput. Sci., vol. 1853, Springer-Verlag, pages 61–72, 2000.

[18] Y. Han. Improved algorithm for all pairs shortest paths. *Inform. Process. Lett.*, 91:245–250, 2004.

[19] Y. Han. An $O(n^3(\log\log n/\log n)^{5/4})$ time algorithm for all pairs shortest paths. In *Proc. 14th European Sympos. Algorithms*, Lect. Notes Comput. Sci., vol. 4168, Springer-Verlag, pages 411–417, 2006.

[20] V. King. Fully dynamic algorithms for maintaining all-pairs shortest paths and transitive closure in digraphs. In *Proc. 40th IEEE Sympos. Found. Comput. Sci.*, pages 81–91, 1999.

[21] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoret. Comput. Sci.*, 312:47–74, 2004.

[22] S. Pettie and V. Ramachandran. A shortest path algorithm for real-weighted undirected graphs. *SIAM J. Comput.*, 34:1398–1431, 2005.

[23] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *J. Comput. Sys. Sci.*, 51:400–403, 1995.

[24] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In *Proc. 40th IEEE Sympos. Found. Comput. Sci.*, pages 605–614, 1999.

[25] P. M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2\log^2 n)$. *SIAM J. Comput.*, 2:28–32, 1973.

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

[27] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Inform. Process. Lett.*, 43:195–199, 1992.

[28] T. Takaoka. Subcubic cost algorithms for the all pairs shortest path problem. *Algorithmica*, 20:309–318, 1998.

[29] T. Takaoka. A faster algorithm for the all-pairs shortest path problem and its application. In *Proc. 10th Int. Conf. Comput. Comb.*, Lect. Notes Comput. Sci., vol. 3106, Springer-Verlag, pages 278–289, 2004.

[30] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46:362–394, 1999.

[31] M. Thorup. Integer priority queues with decrease key in constant time and the single source shortest paths problem. *J. Comput. Sys. Sci.*, 69:330–353, 2004.

[32] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM J. Comput.*, 20:100–125, 1991.

[33] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1:2–13, 2005.

[34] U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *J. ACM*, 49:289–317, 2002.

[35] U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. *Algorithmica*, 46:181–192, 2006.

Appendix

A deterministic proof of Lemma 4.3 (undirected case): Given an unweighted undirected graph $G = (V, E)$, We pick a starting vertex s and run BFS for k levels to obtain the sets $S_i = \{v \in V \mid \delta_G(s, v) = i\}$ for $i = 0, \ldots, k$, where k is the smallest index with $|S_k| \leq 2((|S_0| + \cdots + |S_k|)/\ell) \ln n$. We then add the vertices of S_k to R, and recursively handle the subgraph G' induced by $V - S_0 - \cdots - S_k$.

Excluding the recursive step, the running time is proportional to the total degree among the vertices in $S_0 \cup \cdots \cup S_k$. So, the overall running time is $O(m)$. The overall size of R is clearly $O((n/\ell) \ln n)$.

We claim that $k < \ell/2$: by letting $n_i = |S_0| + \cdots + |S_i|$, we see that for all $i = 1, \ldots, k-1$, $n_i - n_{i-1} > 2(n_i/\ell) \ln n$. So, if $k \geq \ell/2$, then

$$n_i \ > \ n_{i-1} \left[1 - \frac{2 \ln n}{\ell}\right]^{-1} \qquad \Longrightarrow \qquad n_k \ > \ \left[1 - \frac{2 \ln n}{\ell}\right]^{-\ell/2} \ > \ n,$$

a contradiction.

Correctness is a consequence of the following stronger statement: for every pair u, v of shortest-path distance at least ℓ in G, any path π from u to v in G passes through some vertex of R. To prove this, notice that if both u and v are in $S_0 \cup \cdots \cup S_k$, then $\delta_G(u, v) \leq 2k < \ell$. So assume that $u \notin S_0 \cup \cdots \cup S_k$. If π is entirely within the subgraph G', then the statement follows by induction (as the distance between u and v in G' is surely at least ℓ). Otherwise, π contains a vertex w in $S_0 \cup \cdots \cup S_k$. Now, u has distance more than k to s, and w has distance at most k to s in G. As we traverse through the vertices in the subpath from u to w, the distance to s can only increment, decrement, or stay unchanged. So, the subpath must pass through a vertex in S_k. $\square$