Speeding Up Dijkstra's Algorithm for All Pairs Shortest Paths *

Camil Demetrescu †
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza", Roma, Italy

Giuseppe F. Italiano ‡
Dipartimento di Informatica, Sistemi e Produzione
Università di Roma "Tor Vergata", Roma, Italy

Abstract

We present a technique for reducing the number of edge scans performed by Dijkstra's algorithm for computing all pairs shortest paths. The main idea is to restrict path scanning only to locally shortest paths, i.e., paths whose proper subpaths are shortest paths. On a directed graph with n vertices and m edges, the technique we discuss allows it to reduce the number of edge scans from $O(mn)$ to $O(lsp)$, where $lsp \leq mn$ is the number of locally shortest paths in the graph. Using Fibonacci heaps, this method can be implemented in $O(lsp + n^2 \log n)$ worst-case time in a graph with non-negative real edge weights. Experimental evidence suggests that $lsp << mn$ for many classes of graphs that arise in practical applications, resulting in substantial speedups especially in the case of dense graphs.

1 Introduction

All Pairs Shortest Path (APSP) is perhaps one of the most fundamental graph problems. The fastest static algorithm for APSP on graphs with arbitrary real weights is achieved by Dijkstra's algorithm [5] with the Fibonacci heaps of Fredman and Tarjan [7] and has a running time of $O(mn + n^2 \log n)$, where m is the number of edges and n is the number of

*This work has been partially supported by the Sixth Framework Programme of the EU under contract number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet"), and number 001907 ("DELIS : Dynamically Evolving, Large Scale Information Systems"), and by the Italian Ministry of University and Research (Project "ALGO-NEXT: Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments").

†Email: demetres@dis.uniroma1.it. URL: http://www.dis.uniroma1.it/~demetres.

‡Email: italiano@disp.uniroma2.it. URL: http://www.disp.uniroma2.it/users/italiano.

vertices in the graph. This is $\Omega(n^3)$ in the worst case. Fredman [6] and later Takaoka [11] showed how to break this cubic barrier: the best asymptotic bound is by Takaoka, who showed how to solve APSP in $O(n^3\sqrt{\log\log n/\log n})$ time.

The quest for faster subcubic algorithms has led to a surge of interest in APSP. Alon, Galil and Margalit [1] were the first to show that fast matrix multiplication can be effectively used for solving APSP on directed graphs with small integer weights: they gave an algorithm whose running time is $\tilde{O}(n^{(3+\omega)/2})$ for APSP on directed graphs whose weights are integers in the set $\{-1, 0, 1\}$, where ω is the best known exponent for matrix multiplication: currently, $\omega < 2.376$ [2]. Galil and Margalit [8] and Seidel [9] gave $\tilde{O}(n^\omega)$ algorithms for solving APSP for undirected graphs whose weights are integers in the set $\{-1, 0, 1\}$. Shoshan and Zwick [10] and Zwick [12] achieve the best known bound for APSP with positive integer edge weights less than C: $\tilde{O}(Cn^\omega)$ for undirected graphs [10] and $O(C^{0.681}n^{2.575})$ for directed graphs [12]. All these subcubic algorithms are based on clever reductions to fast matrix multiplication.

Our contributions. In this paper we present a technique for reducing the number of edge scans performed by Dijkstra's algorithm for computing all pairs shortest paths in a directed graph with non-negative edge weights. The main idea is to visit only locally shortest paths, i.e., paths whose proper subpaths are shortest paths. On a directed graph with n vertices and m edges, the technique allows it to reduce the number of edge scans from $O(mn)$ to $O(lsp)$, where $lsp \leq mn$ is the number of locally shortest paths in the graph. Using Fibonacci heaps, the method can be implemented in $O(lsp + n^2 \log n)$ worst-case time. Experimental evidence [4] suggested that $lsp << mn$ for many classes of graphs that arise in practical applications, resulting in substantial speedups especially in the case of dense graphs.

2 An all pairs variant of Dijkstra's algorithm

Figure 1 shows an all pairs variant of Dijkstra's algorithm that computes distances from all vertices simultaneously. The algorithm maintains an upper bound D_{xy} to the distances between each pair of vertices x, y in the input graph, rather than from a single source, and uses a priority queue H that contains pairs of vertices, rather than single vertices. The priority queue is initially loaded with all pairs of vertices (x, y) connected by an edge, with a priority equal to the weight of the edge (line 5). The main loop of the algorithm extracts at each iteration the pair (x, y) with the minimum priority (line 7). As discussed in Claim 1 below, the distance D_{xy} is correct at that time. The algorithm then scans all extensions of the shortest path from x to y by one edge (y, z) (line 8), and checks whether the obtained path from x to z has a weight $D_{xy} + w_{yz}$ smaller than the current distance upper bound D_{xz} (line 9). In this case, the pair (x, z) is either added to H with priority $D_{xy} + w_{yz}$, if this is the first path considered between x and z (line 10), or its priority is decreased to $D_{xy} + w_{yz}$ otherwise (line 11). Finally, the distance upper bound D_{xz} is decreased to $D_{xy} + w_{yz}$. The algorithm ends when the priority queue gets empty (line 7).

```
    Dijkstra-APSP( G = (V, E, w) ) → distance matrix:
1.      H ← ∅                                                    {H is a priority queue}
2.      for each x, y ∈ V do                                     {initialization}
3.          let D_xy ← 0 if x = y and D_xy ← +∞ otherwise
4.      for each (x, y) ∈ E do                                   {priority queue loading}
5.          let D_xy ← w_xy and add (x, y) to H with priority D_xy
6.      while H ≠ ∅ do                                           {main loop}
7.          extract (x, y) from H with minimum D_xy
8.          for each z ∈ V : (y, z) ∈ E do                       {shortest path extension}
9.              if D_xy + w_yz < D_xz then
10.                 if D_xz = +∞ then add (x, z) to H with priority D_xy + w_yz
11.                 else decrease the priority of (x, z) in H to D_xy + w_yz
12.                 let D_xz ← D_xy + w_yz
13.     return D
```

Figure 1: All pairs variant of Dijkstra's algorithm.

Analysis. The correctness of the algorithm can be proved by observing that, if we only consider the iterations that extract pairs with a given starting point s, the algorithm performs exactly the same tasks as the classical single-source Dijkstra's algorithm run with source s. Therefore, the algorithm interleaves the distance computations from each vertex in the graph:

Claim 1 *If edge weights are non-negative, every time algorithm* `Dijkstra-APSP` *executes line 7, D_{xy} is equal to the correct distance from x to y in the graph.*

Notice that each pair is inserted and extracted only once from the priority queue. Moreover, each edge (y, z) is scanned only once for a given source x, which yields a total number of $O(mn)$ edge scans. Using a priority queue that supports each priority decrease operation in $O(1)$ time and each insertion/extraction operation in $O(\log N)$ time on a collection of N elements, and observing that $O(\log n^2) = O(\log n)$, we obtain the following time bound:

Claim 2 *If edge weights are non-negative and H is maintained using Fibonacci heaps [7], algorithm* `Dijkstra-APSP` *runs in $O(mn + n^2 \log n)$ worst-case time.*

3 Reducing the number of edge scans

We now show how to modify algorithm `Dijkstra-APSP` with the aim of reducing the number of edge scans. The idea is to exploit a necessary condition for a path to be a shortest path: namely, that each of its proper subpaths must be shortest paths. Notice that at each iteration, algorithm `Dijkstra-APSP` scans all possible extensions by one edge of a shortest path, even if they do not satisfy the necessary condition mentioned above. Our goal is to modify the algorithm in order to scan only paths whose proper subpaths

are shortest paths. These paths, which are good candidates for being shortest paths, are known as *locally shortest paths* [3].

As we will see in Claim 5, the number of locally shortest paths in a graph can never exceed mn. However, experimental evidence [4] shows that in practice this number can be much smaller than mn.

Additional data structures. In addition to the distance matrix D and the priority queue H, we keep the following extra information for each pair of vertices x, y:

1. the vertex, denoted by ℓ_{xy}, that immediately follows x on the current shortest path from x to y. Notice that $\ell_{xy} = y$ if this path is made of a singe edge;
2. the vertex, denoted by r_{xy}, that immediately precedes y on the current shortest path from x to y. Notice that $r_{xy} = x$ if this path is made of a singe edge;
3. a list L_{xy} that contains all vertices x' such that $\langle x', x, \ldots, y\rangle$ is a shortest path, where $\langle x, \ldots, y\rangle$ is the shortest path from x to y found by the algorithm.
4. a list R_{xy} that contains all vertices y' such that $\langle x, \ldots, y, y'\rangle$ is a shortest path, where $\langle x, \ldots, y\rangle$ is the shortest path from x to y found by the algorithm.

We remark that the total space required to store all lists L_{xy} and R_{xy} is $O(n^2)$. Indeed, for each pair (x, y) with $x \neq y$, x is only stored in $L_{\ell_{xy}y}$ and y is only stored in $R_{xr_{xy}}$. Clearly, all ℓ_{xy}'s and r_{xy}'s can also be stored in $O(n^2)$ space. Thus, the additional data structures only increase the space usage by a small constant factor.

We also observe that, at the end of the algorithm execution, ℓ allows us to construct in optimal time any shortest path $\langle v_0, v_1, \ldots, v_k, y\rangle$ by observing that $v_i = \ell_{v_{i-1},y}$ for each i, $1 \leq i \leq k$. Clearly, r can be used in a similar way for the same purpose.

Algorithm variant. The modified algorithm, which we call `Dijkstra-APSP-LSP`, is shown in Figure 2. The only differences with respect to `Dijkstra-APSP` are the following:

Line 2: L_{xy} and R_{xy} are created as empty lists for each pair of vertices x, y;

Line 5: ℓ_{xy} and r_{xy} are initialized for each edge (x, y); they can be later overwritten if paths with smaller weights are found;

Line 8: the endpoints x and y of the shortest path just extracted from the priority queue are added to the list of extensions $L_{\ell_{xy}y}$ and $R_{xr_{xy}}$;

Line 9: this is the crucial step that allows it to reduce the number of edge scans. We enumerate all vertices z such that $\langle \ell_{xy}, \ldots, y, z\rangle$ is a shortest path, with the aim of checking if the resulting scanned path $\langle x, \ell_{xy}, \ldots, y, z\rangle$ has a weight smaller than the current distance upper bound D_{xz}. Notice that, since both $\langle x, \ell_{xy}, \ldots, y\rangle$ and $\langle \ell_{xy}, \ldots, y, z\rangle$ are shortest paths, then all proper subpaths of $\langle x, \ell_{xy}, \ldots, y, z\rangle$ are

```
    Dijkstra-APSP-LSP( G = (V, E, w) ) → distance matrix:
1.      H ← ∅                                                    {H is a priority queue}
2.      for each x, y ∈ V do let D_xy ← +∞ and L_xy ← R_xy ← ∅
3.      for each (x, y) ∈ E do
4.          let D_xy ← w_xy and add (x, y) to H with priority D_xy
5.          let ℓ_xy ← y and r_xy ← x
6.      while H ≠ ∅ do                                           {main loop}
7.          extract (x, y) from H with minimum D_xy
8.          add x to L_{ℓ_xy y} and add y to R_{x r_xy}
9.          for each z ∈ R_{ℓ_xy y} do
10.             if D_xy + w_yz < D_xz then
11.                 if D_xz = +∞ then add (x, z) to H with priority D_xy + w_yz
12.                 else decrease the priority of (x, z) in H to D_xy + w_yz
13.                 let D_xz ← D_xy + w_yz
14.                 let ℓ_xz ← ℓ_xy and r_xz ← y
15.         for each z ∈ L_{x r_xy} do
16.             if w_zx + D_xy < D_zy then
17.                 if D_zy = +∞ then add (z, y) to H with priority w_zx + D_xy
18.                 else decrease the priority of (z, y) in H to w_zx + D_xy
19.                 let D_zy ← w_zx + D_xy
20.                 let ℓ_zy ← x and r_zy ← r_xy
21.     return D
```

Figure 2: Variant of Dijkstra's algorithm based on locally shortest paths.

shortest paths, and so this path is a locally shortest path. Notice that algorithm `Dijkstra-APSP` would have scanned all neighbors z of y at this point.

Line 14: this line is executed if $\langle x, \ell_{xy}, \ldots, y, z\rangle$ becomes the current best path from x to y; in this case, we need to update ℓ_{xz} and r_{xz};

Lines 15–20: these lines do essentially the same steps as lines 9–14, by extending the shortest path $\langle x, \ldots, y\rangle$ on the right rather than on the left. We will discuss why these extra lines are needed in the proof of Claim 3.

Analysis of correctness. To establish the correctness of the method, we need to show that for each pair of vertices x, y, the algorithm scans all locally shortest paths between x and y, taking the minimum of their weights. Since shortest paths are also locally shortest, this minimum must be the distance from x to y.

Claim 3 *Each locally shortest path of the graph is scanned by algorithm* `Dijkstra-APSP-LSP` *in either lines 9–14 or lines 15–20.*

Proof. Let $\langle u, a, \ldots, b, v\rangle$ be any locally shortest path in the graph. By definition, both $\langle u, a, \ldots, b\rangle$ and $\langle a, \ldots, b, v\rangle$ are shortest paths. Suppose that pair (u, b) is extracted before

pair (a, v) in line 7. At that time, u is added to L_{ab} (line 8); however, since (a, v) has not yet been extracted, then $v \notin R_{ab}$, and thus path $\langle u, a, \ldots, b, v\rangle$ is not scanned in lines 9–14. Notice that algorithm `Dijkstra-APSP` would have scanned it at this point. The path will instead be scanned by lines 15–20 only at the iteration that extracts (a, v), since $u \in L_{ab}$ at that time.

If pair (a, v) is instead extracted before pair (u, b), then $\langle u, a, \ldots, b, v\rangle$ is only scanned in lines 9–14 at the iteration that extracts (u, b). This is the reason why extensions at both endpoints of a shortest path are performed by algorithm `Dijkstra-APSP-LSP` in lines 9–14 and in lines 15–20. □

Analysis of running time. To bound the number of steps required by the algorithm, we notice that each pair of vertices is inserted and extracted only once from H, and each priority decrease operation on H is made only when a locally shortest path is scanned. Since by Claim 3 each locally shortest path is only scanned once, and observing that each operation on ℓ, r, L and R can be implemented in constant time, we obtain the following bound:

Claim 4 *If edge weights are non-negative and H is maintained using Fibonacci heaps [7], algorithm* `Dijkstra-APSP-LSP` *runs in $O(lsp + n^2 \log n)$ worst-case time, where lsp is the number of locally shortest paths of the graph.*

We now discuss how many locally shortest paths there can be in a graph. Without loss of generality, we assume that, for each pair of vertices x, y there is a unique shortest path connecting them. Indeed, if this is not the case, we can modify the algorithm by using a simple tie-breaking strategy [3]: we assign to each edge a unique numerical ID and we let the ID of a path be the minimum ID of its edges; if there are two shortest paths by the same weight, we consider as shortest only the one with the minimum ID. Under this assumption, we can bound the number of locally shortest paths in a graph as follows:

Claim 5 *If shortest paths are unique, there can be at most mn locally shortest paths in a graph.*

Proof. To construct a locally shortest path $\langle x, a, \ldots, y\rangle$, the first edge (x, a) can be chosen in m different ways, the endpoint y can be chosen in n different ways, and there is a unique shortest subpath $\langle a, \ldots, y\rangle$. □

Notice that the bound of Claim 5 is tight, in the sense that there exists a family of graphs with $\Omega(mn)$ locally shortest paths. For instance, it is easy to see that in a graph made of two complete bipartite subgraphs sharing a common level of vertices there are $\Omega(n^3)$ locally shortest paths. However, in a previous work [4] we have shown that in random graphs and in some relevant families of real-world graphs they tend to be very close to n^2, i.e., there is typically only one locally shortest path between any pair of nodes, which is also a shortest path. Experiments also showed that in dense graphs, where edge scans are the bottleneck, algorithm `Dijkstra-APSP-LSP` can be orders of magnitude faster than

`Dijkstra-APSP`. On the other hand, for very sparse graphs, `Dijkstra-APSP-LSP` may be slightly slower than `Dijkstra-APSP` mainly due to the overhead of keeping the additional data structures L, R, ℓ and r.

References

[1] N. Alon, Z. Galil, and O. Margalit. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences*, 54(2):255–262, April 1997.

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

[3] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. *Journal of the Association for Computing Machinery (JACM)*, 51(6):968–992, 2004.

[4] C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *To appear in ACM Transactions on Algorithms*, 2005. Special issue devoted to the best papers selected from the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04).

[5] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] M. L. Fredman. New bounds on the complexity of the shortest path problems. *SIAM Journal on Computing*, pages 87–89, 1976.

[7] M.L. Fredman and R.E. Tarjan. Fibonacci heaps and their use in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[8] Z. Galil and O. Margalit. All pairs shortest paths for graphs with small integer length edges. *Journal of Computer and System Sciences*, 54(2):243–254, April 1997.

[9] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, December 1995.

[10] A. Shoshan and U. Zwick. All pairs shortest paths in undirected graphs with integer weights. In IEEE, editor, *40th Annual Symposium on Foundations of Computer Science: October 17–19, 1999, New York City, New York,*, pages 605–614, 1999.

[11] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, September 1992.

[12] U. Zwick. All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49(3):289–317, 2002.