Engineering Shortest Path Algorithms*

Camil Demetrescu[1] and Giuseppe F. Italiano[2]

[1] Dipartimento di Informatica e Sistemistica,
Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italy. Email: demetres@dis.uniroma1.it,
URL: http://www.dis.uniroma1.it/~demetres/

[2] Dipartimento di Informatica, Sistemi e Produzione,
Università di Roma "Tor Vergata",
via del Politecnico 1, 00133 Roma, Italy. Email: italiano@disp.uniroma2.it,
URL: http://www.disp.uniroma2.it/users/italiano/

Abstract. In this paper, we report on our own experience in studying a fundamental problem on graphs: all pairs shortest paths. In particular, we discuss the interplay between theory and practice in engineering a simple variant of Dijkstra's shortest path algorithm. In this context, we show that studying heuristics that are efficient in practice can yield interesting clues to the combinatorial properties of the problem, and eventually lead to new theoretically efficient algorithms.

1 Introduction

The quest for efficient computer programs for solving real world problems has led in recent years to a growing interest in experimental studies of algorithms. Producing efficient implementations requires taking into account issues such as memory hierarchy effects, hidden constant factors in the performance bounds, implications of communication complexity, numerical precision, and use of heuristics, which are sometimes overlooked in classical analysis models. On the other hand, developing and assessing heuristics and programming techniques for producing codes that are efficient in practice is a difficult task that requires a deep understanding of the mathematical structure and the combinatorial properties of the problem at hand. In this context, experiments can raise new conjectures and theoretical questions, opening unexplored research directions that may lead to further theoretical improvements and eventually to more practical algorithms. The whole process of designing, analyzing, implementing, tuning, debugging and experimentally evaluating algorithms is usually referred to as *Algorithm Engineering*. As shown in Figure 1, algorithm engineering is a cyclic process: designing

* Work 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"), by the IST Programme of the EU under contract number IST-2001-33555 ("COSIN: COevolution and Self-organization In dynamical Networks"), and by the Italian Ministry of University and Scientific Research (Project "ALINWEB: Algorithmics for Internet and the Web").

algorithmic techniques and analyzing their performance according to theoretical models provides a sound foundation to writing efficient computer programs. On the other hand, analyzing the practical performance of a program can be helpful in spotting bottlenecks in the code, designing heuristics, refining the theoretical analysis, and even devising more realistic cost models in order to get a deeper insight into the problem at hand. We refer the interested reader to [4] for a broader survey of algorithm engineering issues.

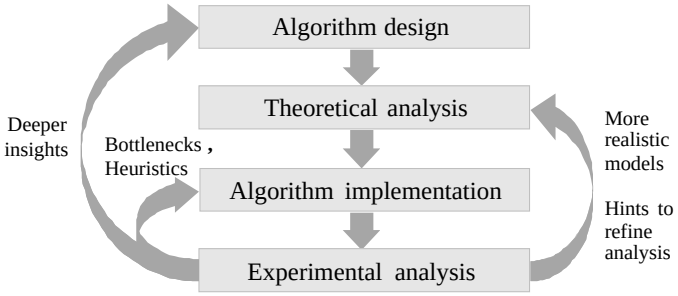


Fig. 1. The algorithm engineering cycle.

In this paper, we report on our own experience in studying a fundamental problem on graphs: the all pairs shortest paths problem. We discuss the interplay between theory and practice in engineering a variant of Dijkstra's shortest path algorithm. In particular, we present a simple heuristic that can improve substantially the practical performances of the algorithm on many typical instances. We then show that a deeper study of this heuristic can reveal interesting combinatorial properties of paths in a graph. Surprisingly, exploiting such properties can lead to new theoretically efficient methods for updating shortest paths in a graph subject to dynamic edge weight changes.

2 From theory to experiments: engineering Dijkstra's algorithm

In 1959, Edsger Wybe Dijkstra devised a simple algorithm for computing shortest paths in a graph [6]. Although more recent advances in data structures led to faster implementations (see, e.g., the Fibonacci heaps of Fredman and Tarjan [7]), Dijkstra's algorithmic invention is still in place after 45 years, providing an ubiquitous standard framework for designing efficient shortest path algorithms. If priority queues with constant amortized time per decrease are used, the basic method computes the shortest paths from a given source vertex in $O(m + n \log n)$ worst-case time in a graph with n vertices and m edges. To compute all pairs shortest paths, we get an $O(mn + n^2 \log n)$ bound in the worst case by simply repeating the single-source algorithm from each vertex.

```
    algorithm DijkstraAPSP(graph G = (V, E, w)) → matrix
1.      let d be an n × n matrix and let H be an empty priority queue
2.      for each pair (x, y) ∈ V × V do                          {initialization}
3.          if (x, y) ∈ E then d[x, y] ← w(x, y) else d[x, y] ← +∞
4.          add edge (x, y) to H with priority d[x, y]
5.      while H is not empty do                                  {main loop}
6.          extract from H a pair (x, y) with minimum priority
7.          for each edge (y, z) ∈ E leaving y do                {forward extension}
8.              if d[x, y] + w(y, z) < d[x, z] then              {relaxation}
9.                  d[x, z] ← d[x, y] + w(y, z)
10.                 decrease the priority of (x, z) in H to d[x, z]
11.         for each edge (z, x) ∈ E entering x do               {backward extension}
12.             if w(z, x) + d[x, y] < d[z, y] then              {relaxation}
13.                 d[z, y] ← w(z, x) + d[x, y]
14.                 decrease the priority of (z, y) in H to d[z, y]
15.     return d
```

Fig. 2. All pairs variant of Dijkstra's algorithm. $w(x, y)$ denotes the weight of edge (x, y) in G.

In Figure 2, we show a simple variant of Dijkstra's algorithm where all pairs shortest paths are computed in an interleaved fashion, rather than one source at a time. The algorithm maintains a matrix d that contains at any time an upper bound to the distances in the graph. The upper bound $d[x, y]$ for any pair of vertices (x, y) is initially equal to the edge weight $w(x, y)$ if there is an edge between x and y, and $+\infty$ otherwise. The algorithm also maintains in a priority queue H each pair (x, y) with priority $d[x, y]$. The main loop of the algorithm repeatedly extracts from H a pair (x, y) with minimum priority, and tries to extend at each iteration the corresponding shortest path by exactly one edge in every possible direction. This requires scanning all edges leaving y and entering x, performing the classical *relaxation* step to decrease the distance upper bounds d. It is not difficult to prove that at the end of the procedure, if edge weights are non-negative, d contains the exact distances. The time required for loading and unloading the priority queue is $O(n^2 \log n)$ in the worst case. Each edge is scanned at most $O(n)$ times and for each scanned edge we spend constant amortized time if H is, e.g., a Fibonacci heap. This yields $O(mn + n^2 \log n)$ worst-case time.

While faster algorithms exist for very sparse graphs [8–10], Dijkstra's algorithm appears to be still a good practical choice in many real world settings. For this reason, the quest for fast implementations has motivated researchers to study methods for speeding up Dijkstra's basic method based on priority queues and relaxation. For instance, it is now well understood that efficient data structures play a crucial role in the case of sparse graphs, while edge scanning is the typical bottleneck in dense graphs [1]. In the rest of this section, we focus on the algorithm of Figure 2, and we investigate heuristic methods for reducing the number of scanned edges.

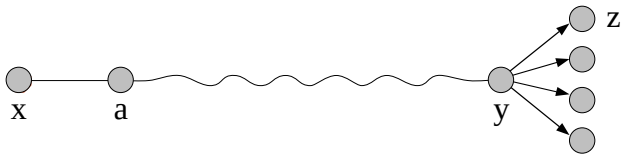


Fig. 3. Extending the shortest path from x to y by one edge.

Consider line 7 of Figure 2 (or, similarly, line 11): the loop scans all edges (y, z), seeking for tentative shortest paths of the form $x \rightsquigarrow y \rightarrow z$ with weight $d[x, y] + w(y, z)$. Do we really need to scan all edges (y, z) leaving y? More to the point, is there any way to avoid scanning an edge (y, z) if it cannot belong to a shortest path from x to z? Let π_{xy} be a shortest path from x to y and let $\pi_{xz} = \pi_{xy} \cdot (y, z)$ be the path obtained by going from x to y via π_{xy} and then from y to z via edge (y, z) (see Figure 3). Consider now the well-known optimal-substructure property of shortest paths (see e.g., [2]):

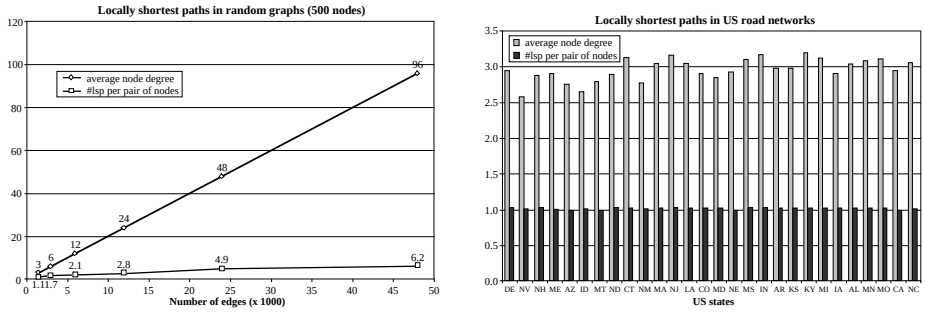
Lemma 1. *Every subpath of a shortest path is a shortest path.*

This property implies that, if π_{xz} is a shortest path and we remove either one of its endpoints, we still get a shortest path. Thus, edge (y, z) can be the last edge of a shortest path π_{xz} *only if* both $\pi_{xy} \subset \pi_{xz}$ and $\pi_{az} \subset \pi_{xz}$ are shortest paths, where (x, a) is the first edge in π_{xz}. Let us now exploit this property in our shortest path algorithm in order to avoid scanning "unnecessary" edges. In the following, we assume without loss of generality that shortest paths in the graph are unique. We can just maintain for each pair of vertices (x, y) a list of edges $R_{xy} = \{\ (y, z)$ s.t. $\pi_{xz} = \pi_{xy} \cdot (y, z)$ is a shortest path $\}$, and a list of edges $L_{xy} = \{\ (z, x)$ s.t. $\pi_{zy} = (z, x) \cdot \pi_{xy}$ is a shortest path $\}$, where π_{xy} is the shortest path from x to y. Such lists can be easily constructed incrementally as the algorithm runs at each pair extraction from the priority queue H. Suppose now to modify line 7 and line 11 of Figure 2 to scan just edges in R_{ay} and L_{xb}, respectively, where (x, a) is the first edge and (b, y) is the last edge in π_{xy}. It is not difficult to see that in this way we consider in the relaxation step only paths whose proper subpaths are shortest paths. We call such paths *locally shortest* [5]:

Definition 1. *A path π_{xy} is* locally shortest *in G if either:*

(i) π_{xy} consists of a single vertex, or
(ii) every proper subpath of π_{xy} is a shortest path in G.

With the modifications above, the algorithm requires $O(|LSP| + n^2 \log n)$ worst-case time, where $|LSP|$ is the number of locally shortest paths in the graph. A natural question seems to be: how many locally shortest paths can we have in a graph? The following lemma is from [5]:



(a) (b)

Fig. 4. Average number of locally shortest paths connecting a pair of vertices in: (a) a family of random graphs with increasing density; (b) a suite of US road networks obtained from `ftp://edcftp.cr.usgs.gov`.

Lemma 2. *If shortest paths are unique in G, then there can be at most mn locally shortest paths in G. This bound is tight.*

This implies that our modification of Dijkstra's algorithm does not produce an asymptotically faster method. But what about typical instances? In [3], we have performed some counting experiments on both random and real-world graphs (including road networks and Internet Autonomous Systems subgraphs), and we have discovered that in these graphs $|LSP|$ tends to be surprisingly very close to n^2. In Figure 4, we show the average number of locally shortest paths connecting a pair of vertices in a family of random graphs with 500 vertices and increasing number of edges, and in a suite of US road networks. According to these experiments, the computational savings that we might expect using locally shortest paths in Dijkstra's algorithm increase as the edge density grows. In Figure 5, we compare the actual running time of a `C` implementation the algorithm given in Figure 2 (`S-DIJ`), and the same algorithm with the locally shortest path heuristic described above (`S-LSP`). Our implementations are described in [3] and are available at: `http://www.dis.uniroma1.it/~demetres/experim/dsp/`. Notice that in a random graph with density 20% `S-LSP` can be 16 times faster than `S-DIJ`. This confirms our expectations based on the counting results given in Figure 4. On very sparse graphs, however, `S-LSP` appears to be slightly slower than `S-DIJ` due to the data structure overhead of maintaining lists L_{xy} and R_{xy} for each pair of vertices x and y.

3 Back from engineering to theory: a new dynamic algorithm

Let us now consider a scenario in which the input graph changes over time, subject to a sequence of edge weight updates. The goal of a dynamic shortest paths algorithm is to update the distances in the graph more efficiently than

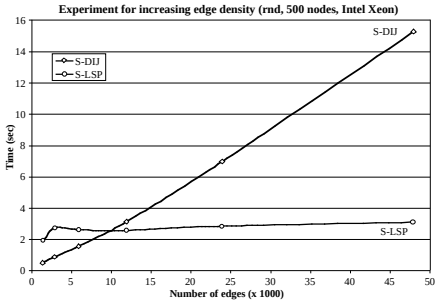


Fig. 5. Comparison of the actual running time of a `C` implementation of Dijkstra's algorithm with (`S-LSP`) and without (`S-DIJ`) the locally shortest paths heuristic in a family of random graphs with 500 vertices, increasing edge density, and integer edge weights in $[1, 1000]$. Experiments were performed on an Intel Xeon 500MHz, 512KB L2 cache, 512MB RAM.

recomputing the whole solution from scratch after each change of the weight of an edge. In this section, we show that the idea of using locally shortest paths, which appears to be a useful heuristic for computing all pairs shortest paths as we have seen in Section 2, can play a crucial role in designing asymptotically fast update algorithms for the dynamic version of the problem. Let us consider the case where edge weights can only be increased (the case where edge weights can only be decreased in analogous). Notice that, after increasing the weight of an edge, some of the shortest paths containing it may stop being shortest, while other paths may become shortest, replacing the old ones. The goal of a dynamic update algorithm is find efficiently such replacement paths. Intuitively, a locally shortest path is either shortest itself, or it just falls short of being shortest. Locally shortest paths are therefore natural candidates for being replacement paths after an edge weight increase. A possible approach could be to keep in a data structure all the locally shortest paths of a graph, so that a replacement path can be found quickly after an edge update. On the other hand, keeping such a data structure up to date should not be too expensive. To understand if this is possible at all, we first need to answer the following question: how many paths can start being locally shortest and how many paths can stop being locally shortest after an edge weight increase? The following theorem from [5] answers this question:

Theorem 1. *Let G be a graph subject to a sequence of edge weight increases. If shortest paths are unique in G, then during each update:*

(1) $O(n^2)$ paths can stop *being locally shortest;*
(2) $O(n^2)$ paths can start *being locally shortest, amortized over $\Omega(n)$ operations.*

According to Theorem 1, we might hope to maintain explicitly the set of locally shortest paths in a graph in quadratic amortized time per operation. Since shortest paths in a graph are locally shortest, then maintaining such a

set would allow us to keep information also about shortest paths. As a matter of fact, there exists a dynamic variant of the algorithm given in Figure 2 that is able to update the locally shortest paths (and thus the shortest paths) of a graph in $O(n^2 \log n)$ amortized time per edge weight increase; details can be found in [5]. For graphs with $m = \Omega(n \log n)$ edges, this is asymptotically faster than recomputing the solution from scratch using Dijkstra's algorithm. Furthermore, if the distance matrix has to be maintained explicitly, this is only a polylogarithmic factor away from the best possible bound. Surprisingly, no previous result was known for this problem until recently despite three decades of research in this topic.

To solve the dynamic all pairs shortest paths problem in its generality, where the sequence of updates can contain both increases and decreases, locally shortest paths can no longer be used directly: indeed, if increases and decreases can be intermixed, we may have worst-case situations with $\Omega(mn)$ changes in the set of locally shortest paths during each update. However, using a generalization of locally shortest paths, which encompasses the history of the update sequence to cope with pathological instances, we devised a method for updating shortest paths in $O(n^2 \log^3 n)$ amortized time per update [5]. This bound has been recently improved to $O(n^2 \cdot (\log n + \log^2(m/n)))$ by Thorup [11].

4 Conclusions

In this paper, we have discussed our own experience in engineering different all pairs shortest paths algorithms based on Dijkstra's method. The interplay between theory and practice yielded significant results. We have shown that the novel notion of locally shortest paths, which allowed us to design a useful heuristic for improving the practical performances of Dijkstra's algorithm on dense graphs, led in turn to the first general efficient dynamic algorithms for maintaining the all pairs shortest paths in a graph.

Despite decades of research, many aspects of the shortest paths problem are still far from being fully understood. For instance, can we compute all pairs shortest paths in $o(mn)$ in dense graphs? As another interesting open problem, can we update a shortest paths tree asymptotically faster than recomputing it from scratch after an edge weight change?

References

1. B.V. Cherkassky, A.V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73:129–174, 1996.
2. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill, 2001.
3. C. Demetrescu, S. Emiliozzi, and G.F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*, 2004.
4. C. Demetrescu, I. Finocchi, and G.F. Italiano. Algorithm engineering. *The Algorithmics Column (J. Diaz), Bulletin of the EATCS*, 79:48–63, 2003.

5. C. Demetrescu and G.F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03), San Diego, CA*, pages 159–166, 2003.
6. E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
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. S. Pettie. A faster all-pairs shortest path algorithm for real-weighted sparse graphs. In *Proceedings of 29th International Colloquium on Automata, Languages, and Programming (ICALP'02), LNCS Vol. 2380*, pages 85–97, 2002.
9. S. Pettie and V. Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, pages 267–276. SIAM, January 6–8 2002.
10. S. Pettie, V. Ramachandran, and S. Sridhar. Experimental evaluation of a new shortest path algorithm. In *4th Workshop on Algorithm Engineering and Experiments (ALENEX'02), LNCS Vol. 2409*, pages 126–142, 2002.
11. M. Thorup. Tighter fully-dynamic all pairs shortest paths, 2003. Unpublished manuscript.