Practical Algorithms for the All-Pairs Shortest Path Problem

Andrej Brodnik and Marko Grgurovič

Abstract We study practical algorithms for solving the all-pairs shortest path problem. The Floyd-Warshall algorithm is frequently used to solve the aforementioned problem, and we show how it can be augmented to drastically reduce the number of path combinations examined. Very favorable results are shown via empirical tests that compare the new algorithm with known algorithms on random graphs. In addition to the all-pairs shortest path problem, we also investigate the highly related all-pairs bottleneck paths problem, and give an efficient average case algorithm. On top of that, we show how the bottleneck paths problem relates to the decremental transitive closure problem, and specifically how algorithms for the latter can be used to solve the former.

1 Introduction

Let $G = (V, E)$ denote a directed graph where E is the set of edges and $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices of the graph. The function $\ell(\cdot)$ maps edges to (possibly negative) lengths. For a path π, we define its length to be the sum of the lengths of its edges: $\ell(\pi) = \sum_{(u,v)\in\pi} \ell(u, v)$. Additionally, we define $\forall(u, v) \notin E : \ell(u, v) = \infty$. From hereon we make the standard assumption that there are no cycles whose total lengths are negative, and without loss of generality, we assume G is strongly connected. To simplify notation, we define $m = |E|$ and $n = |V|$.

The research presented in this chapter was partially financially supported by the Slovenian Research Agency under N2-0053 and P2-0359.

A. Brodnik (✉) · M. Grgurovič
University of Primorska, Glagoljaška 8, 6000 Koper, Slovenia
e-mail: andrej.brodnik@upr.si

M. Grgurovič
e-mail: marko.grgurovic@famnit.upr.si

A. Brodnik
University of Ljubljana, Večna pot 113, 1000 Ljubljana, Slovenia

A. Adamatzky (ed.), *Shortest Path Solvers. From Software to Wetware*,
Emergence, Complexity and Computation 32,
https://doi.org/10.1007/978-3-319-77510-4_6

Furthermore, we define $d(u, v)$ for two vertices $u, v \in V$ as the length of the shortest path from u to v. It is also useful to define m^* as the number of edges (u, v) such that $d(u, v) = \ell(u, v)$. These are the edges that form the shortest path graph, and are the only edges necessary for its computation.

Finding shortest paths in such graphs is a classic problem in algorithmic graph theory. Two of the most common variants of the problem are the single-source shortest path (SSSP) problem and the all-pairs shortest path problem (APSP). In the SSSP variant, we are searching for paths with the least total length from a fixed vertex $s \in V$ to every other vertex in the network. Similarly, the APSP problem asks for the shortest path between every pair of vertices $u, v \in V$. In this chapter we will focus exclusively on the APSP variant of the problem, and without loss of generality, assume that we are not interested in paths beginning in v and returning back to v.

The asymptotically fastest APSP algorithm for dense graphs to date runs in $O(n^3 \log\log^3 n/\log^2 n)$ time [1]. For non-negative edge length functions and for sparse graphs, there exist asymptotically fast algorithms for worst case inputs [2–4], and algorithms which are efficient average-case modifications of Dijkstra's algorithm [5–7].

The APSP problem can easily be solved by n calls to an SSSP algorithm. There exist strategies that are more effective than simply running independent SSSP computations, such as the Hidden Paths Algorithm [8], the Uniform Paths algorithm [5], and most recently the Propagation algorithm [9]. The Propagation algorithm is more general than the former two, which are modifications of Dijkstra, in the sense that it works for any SSSP algorithm. Besides providing a speed-up for arbitrary SSSP algorithms, it also performs well in practice, as shown in [9].

As a truly all-pairs algorithm, Floyd-Warshall [10, 11] is frequently used to solve APSP. There exist many optimizations for the Floyd-Warshall algorithm, ranging from better cache performance [12], optimized program-generated code [13], to parallel variants for the GPU [14, 15]. One can also approach APSP through funny matrix multiplication, and practical improvements have been devised to this end through the use of sorting [16].

In spite of intensive research on efficient implementations of the Floyd-Warshall algorithm, there has not been much focus devoted to improvement of the number of path combinations examined by the algorithm. In Sect. 2, we will propose a modification of the Floyd-Warshall algorithm that combines it with an hourglass-like tree structure, which reduces the number of paths that have to be examined. Only those path combinations that provably cannot change the values in the shortest path matrix are omitted. The resulting algorithm is simple to implement, uses no fancy data structures and in empirical tests is faster than the Floyd-Warshall algorithm for random complete graphs on 256–4096 nodes by factors ranging from 2.5 to 8.5. When we inspect the number of path combinations examined however, our modification reduces the number by a staggering factor of 12–90.

In Sect. 4 we consider the all-pairs bottleneck paths (APBP) problem, which is highly related to the all-pairs shortest path problem. We show that an efficient algorithm whose bound depends on m^* can be obtained, and show how the APBP problem can be reduced to that of decremental transitive closure.

2 The Hourglass Algorithm

The Floyd-Warshall algorithm [10, 11] is a simple dynamic programming approach to solve the all-pairs shortest path problem. Unlike Dijkstra's algorithm, Floyd-Warshall can find shortest paths in graphs which contain negatively-weighted edges. In this section we will outline improvements that build on the base algorithm, but first we outline the pseudocode of the Floyd-Warshall algorithm in Algorithm 1. Intuitively, one might expect that the minimum operation in line 5, also sometimes referred to as relaxation, would not succeed in lowering the value of $W[i][j]$ every time. This is precisely what we aim to exploit: instead of simply looping through every node in line 4, we utilize the structure of shortest paths that we have computed up until now. This allows us to avoid checking many path combinations that the Floyd-Warshall algorithm inspects, but which provably cannot reduce the current value stored inside $W[i][j]$.

Algorithm 1 Floyd-Warshall Algorithm

```
1: procedure FLOYD-WARSHALL($W$)
2:     for $k := 1$ to $n$ do
3:         for $i := 1$ to $n$ do
4:             for $j := 1$ to $n$ do
5:                 $W[i][j] := \min(W[i][j], W[i][k] + W[k][j])$
6:             end for
7:         end for
8:     end for
9: end procedure
```

We will say a path $u \overset{k}{\rightsquigarrow} v$ is a k-shortest path if it is the shortest path between u and v that is only permitted to go through nodes $\{v_1, ..., v_k\}$. This means that $u \overset{n}{\rightsquigarrow} v$ would be the shortest path from u to v in the traditional sense. We denote the length of a path $u \overset{k}{\rightsquigarrow} v$ by writing $\ell(u \overset{k}{\rightsquigarrow} v)$, where the length is simply the sum of the lengths of all edges that are on the path.

The resulting algorithm is still a dynamic programming algorithm, but it now has a smaller pool of candidates to perform relaxation on, which makes it run faster. In Sect. 2.1, we show how to lower the number of candidates looped through in line 4 of Algorithm 1 by exploiting the tree structure of $k \overset{k-1}{\rightsquigarrow} j$ paths. In Sect. 2.2, we show how to exploit the structure of $i \overset{k-1}{\rightsquigarrow} k$ paths and further reduce the number of candidates in line 4. Both reductions are achieved by traversing a tree structure rather than looping through all nodes. These modifications yield two tree data structures, and joining them in the root yields an hourglass shaped data structure that combines the power of both.

2.1 The Single-Tree Algorithm

The simplest improvement involves the use of a tree, denoted as OUT_k, which is the shortest path tree containing paths that begin in node v_k and end in some node $w \in V \backslash \{v_k\}$, but only go through nodes in the set $\{v_1, ..., v_{k-1}\}$. In other words, these are paths of the form $v_k \overset{k-1}{\rightsquigarrow} w \quad \forall w \in V \backslash \{v_k\}$. Traversal of this tree is used to replace the FOR loop on variable j in line 4 of Algorithm 1. In order to reconstruct the shortest paths, the Floyd-Warshall algorithm needs to maintain an additional matrix, which specifies the path structure, but this additional matrix is otherwise not required for the functioning of the algorithm. In our algorithm, however, this information is essential, since the path structure is used during the algorithm's execution. We augment the Floyd-Warshall algorithm with a matrix $L[i][j]$ which specifies the penultimate node on the shortest path from i to j (i.e. the last node that is not j). This suffices for reconstructing the shortest path tree for all paths going out of k as follows: create n trees $\{T_1, ..., T_n\}$, now go through $j = 1$ to n and place T_j as the child of $T_{L[k][j]}$. This takes $O(n)$ time.

Assume that we have the $(k-1)$-shortest paths $i \overset{k-1}{\rightsquigarrow} j \quad \forall i, j \in V$ and we are trying to extend the paths to go through v_k, i.e. we want to compute $i \overset{k}{\rightsquigarrow} j \quad \forall i, j \in V$. First we construct OUT_k in $O(n)$ time. Now we can use the following lemma when extending the paths to go through v_k:

Lemma 1 *Let $v_x \in V \backslash \{v_k\}$ be some non-leaf node in OUT_k and let $v_y \neq v_x$ be an arbitrary node in the subtree rooted at v_x. Now let $v_i \in V \backslash \{v_k\}$ and consider a path $v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_x$. If $\ell(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_x) \geq \ell(v_i \overset{k-1}{\rightsquigarrow} v_x)$, then we claim $\ell(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_y) \geq \ell(v_i \overset{k-1}{\rightsquigarrow} v_y)$.*

Proof By choice of v_y and v_x, we have $v_k \overset{k-1}{\rightsquigarrow} v_y = v_k \overset{k-1}{\rightsquigarrow} v_x \overset{k-1}{\rightsquigarrow} v_y$. Thus we want to show:

$$\ell(v_i \overset{k-1}{\rightsquigarrow} v_y) \leq \ell(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_x) + \ell(v_x \overset{k-1}{\rightsquigarrow} v_y).$$

Observe that $x < k$, since v_x is neither a leaf nor the root of OUT_k. Because $v_i \overset{k-1}{\rightsquigarrow} v_y$ is the $(k-1)$-shortest path and $x < k$ we have:

$$\ell(v_i \overset{k-1}{\rightsquigarrow} v_y) \leq \ell(v_i \overset{k-1}{\rightsquigarrow} v_x) + \ell(v_x \overset{k-1}{\rightsquigarrow} v_y).$$

Putting these together we get:

$$\ell(v_i \overset{k-1}{\rightsquigarrow} v_y) \leq \ell(v_i \overset{k-1}{\rightsquigarrow} v_x) + \ell(v_x \overset{k-1}{\rightsquigarrow} v_y) \leq \ell(v_i \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_x) + \ell(v_x \overset{k-1}{\rightsquigarrow} v_y).$$

Which is what we wanted to prove.

The algorithm then extends the $(k-1)$-shortest paths for each incoming node v_i by depth-first traversal[1] of OUT_k, starting with the root and avoiding the inspection of subtrees whose roots v_x did not yield a shorter path than $v_i \overset{k-1}{\rightsquigarrow} v_x$. Intuitively, one would expect this to exclude large subtrees from ever being considered. The pseudocode is given in Algorithm 2.

Algorithm 2 Single-tree Algorithm

```
 1: procedure SINGLE-TREE($W$)
 2:     Initialize $L$, a $n \times n$ matrix, as $L[i][j] := i$.
 3:     for $k := 1$ to $n$ do
 4:         Construct $OUT_k$.
 5:         for $i := 1$ to $n$ do
 6:             Stack := empty
 7:             Stack.push($v_k$)
 8:             while Stack $\neq$ empty do
 9:                 $v_x :=$ Stack.pop()
10:                 for all children $v_j$ of $v_x$ in $OUT_k$ do
11:                     if $W[i][k] + W[k][j] < W[i][j]$ then
12:                         $W[i][j] := W[i][k] + W[k][j]$
13:                         $L[i][j] := L[k][j]$
14:                         Stack.push($v_j$)
15:                     end if
16:                 end for
17:             end while
18:         end for
19:     end for
20: end procedure
```

Observe that the extra space required by the trees is merely $O(n)$, since we can reuse the same space. Constructing the tree takes $O(n)$ time which yields in total $O(n^2)$ time over the course of the entire algorithm.

2.1.1 Optimized Implementation

Instead of maintaining a stack and visiting nodes in the tree as in Algorithm 2, a much faster implementation is possible in practice. After building the tree OUT_k, we keep track of two permutation arrays: *dfs[]* and *skip[]*. The *dfs* array is simply the depth-first traversal of the tree, i.e. *dfs[x]* contains the x-th vertex encountered on a DFS traversal of OUT_k. For a vertex v_z, *skip[z]* contains the index in *dfs* of the first vertex after v_z in the DFS order that is not a descendant of v_z in OUT_k. Then, all we need to do is simply traverse *dfs* and whenever an improvement is not made, we jump to the next index via the *skip* array. It should be pointed out that the asymptotic time remains the same, as this is solely a practical optimization.

[1]Breadth-first traversal is also possible, of course.

2.2 The Hourglass Algorithm

We can augment Algorithm 2 with another tree. The second tree is similar to OUT_k, except that it is the shortest path "tree" for paths $w \overset{k-1}{\rightsquigarrow} v_k \quad \forall w \in V \setminus \{v_k\}$. Strictly speaking, this is not a tree,[2] but we can reverse the directions of the edges, which turns it into a tree with v_k as the root. We denote this tree as IN_k. Observe that if $v_a \neq v_k$ is a node in IN_k and v_b is a child of v_a in IN_k, then the $(k-1)$-shortest path from v_b to v_k goes through v_a, since the edges are reversed in the tree. Traversal of IN_k will be used as a replacement of the FOR loop on variable i in line 3 of Algorithm 1. In order to construct IN_k efficiently, we need to maintain an additional matrix $F[i][j]$ which stores the second node on the path from i to j (i.e. the first node that is not i). The construction of IN_k is now similar to what we had before: create n trees $\{T_1, ..., T_n\}$, then go through $i = 1$ to n and place T_i as the child of $T_{F[i][k]}$. This takes $O(n)$ time. Consequently, we have the following lemma:

Lemma 2 *Let $v_a \in V \setminus \{v_k\}$ be some non-leaf node in IN_k and let $v_b \neq v_a$ be an arbitrary node in the subtree rooted at v_a. Now let $v_j \in V \setminus \{v_k\}$ and consider a path $v_a \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$. If $\ell(v_a \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq \ell(v_a \overset{k-1}{\rightsquigarrow} v_j)$, then we claim $\ell(v_b \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j) \geq \ell(v_b \overset{k-1}{\rightsquigarrow} v_j)$.*

Proof Due to the choice of v_a and v_b we have: $v_b \overset{k-1}{\rightsquigarrow} v_k = v_b \overset{k-1}{\rightsquigarrow} v_a \overset{k-1}{\rightsquigarrow} v_k$. We want to show, that:

$$\ell(v_b \overset{k-1}{\rightsquigarrow} v_j) \leq \ell(v_b \overset{k-1}{\rightsquigarrow} v_a) + \ell(v_a \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j).$$

Observe that $a < k$, since v_a is neither a leaf nor the root of IN_k. Thus we have:

$$\ell(v_b \overset{k-1}{\rightsquigarrow} v_j) \leq \ell(v_b \overset{k-1}{\rightsquigarrow} v_a) + \ell(v_a \overset{k-1}{\rightsquigarrow} v_j).$$

Putting these together we get the desired inequality:

$$\ell(v_b \overset{k-1}{\rightsquigarrow} v_j) \leq \ell(v_b \overset{k-1}{\rightsquigarrow} v_a) + \ell(v_a \overset{k-1}{\rightsquigarrow} v_j) \leq \ell(v_b \overset{k-1}{\rightsquigarrow} v_a) + \ell(v_a \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j).$$

Observe that if we perform depth-first traversal on IN_k, we can temporarily prune OUT_k as follows: if v_a is the parent of v_b in IN_k and $v_a \overset{k-1}{\rightsquigarrow} v_j \leq v_a \overset{k-1}{\rightsquigarrow} v_k \overset{k-1}{\rightsquigarrow} v_j$, then the subtree of v_j can be removed from OUT_k while we are inspecting the subtree of v_a in IN_k, and later re-inserted. This is easy to do by using a stack to keep track of deletions. The pseudocode for the Hourglass algorithm is given in Algorithm 3. In practice, recursion can be replaced with another stack, and each node in the IN_k tree is then visited twice—the second visit would restore the subtrees that were removed from OUT_k by that node.

[2]The hourglass name comes from placing this structure atop the OUT_k tree, which gives it an hourglass-like shape, with v_k being the neck.

Algorithm 3 Hourglass Algorithm

```
1: procedure HOURGLASS($W$)
2:     Initialize $L$, a $n \times n$ matrix, as $L[i][j] := i$.
3:     Initialize $F$, a $n \times n$ matrix, as $F[i][j] := j$.
4:     for $k := 1$ to $n$ do
5:         Construct $OUT_k$.
6:         Construct $IN_k$.
7:         for all children $v_i$ of $v_k$ in $IN_k$ do
8:             RECURSEIN($W, L, F, IN_k, OUT_k, v_i$)
9:         end for
10:    end for
11: end procedure
12: procedure RECURSEIN($W, L, F, IN_k, OUT_k, v_i$)
13:    Stack := empty
14:    Stack.push($v_k$)
15:    while Stack $\neq$ empty do
16:        $v_x$ := Stack.pop()
17:        for all children $v_j$ of $v_x$ in $OUT_k$ do
18:            if $W[i][k] + W[k][j] < W[i][j]$ then
19:                $W[i][j] := W[i][k] + W[k][j]$
20:                $L[i][j] := L[k][j]$
21:                $F[i][j] := F[i][k]$
22:                Stack.push($v_j$)
23:            else
24:                Remove the subtree of $v_j$ from $OUT_k$.
25:            end if
26:        end for
27:    end while
28:    for all children $v_{i'}$ of $v_i$ in $IN_k$ do
29:        RECURSEIN($W, L, F, IN_k, OUT_k, v_{i'}$)
30:    end for
31:    Restore any subtrees we may have removed in line 24.
32: end procedure
```

The only extra space requirement of the Hourglass algorithm that bears any significance is the matrix F, which contains n^2 entries. It is important to note that the worst-case time complexity of the Hourglass (and Single-tree) algorithm remains $O(n^3)$. The simplest example of this is when all shortest paths are the edges themselves, at which point the tree structure is essentially flat and never changes.

2.3 Empirical Comparison

We now empirically examine how many path combinations are skipped by the Hourglass and Single-tree algorithms compared to the Floyd-Warshall algorithm. We performed two experiments, one on random complete graphs, and one on random sparse graphs. We measured the number of path combinations examined. Since the results are numbers that range from very small to very large in both cases, we display

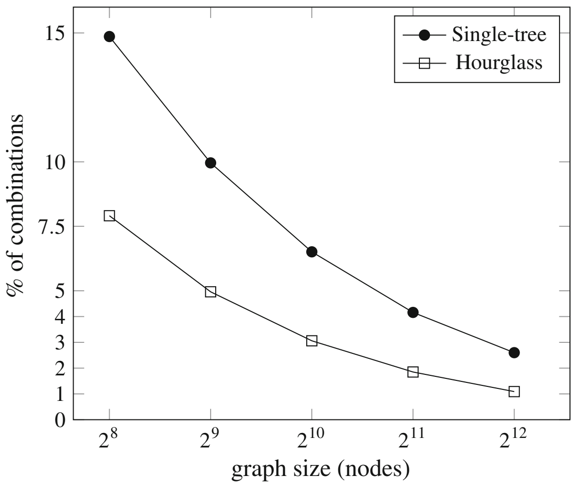

Fig. 1 The percentage of path combinations examined by the two modifications of Floyd-Warshall, when compared to the original algorithm (which is always at 100%, not shown), for the input of complete graphs of various sizes

the results as a percentage of the Floyd-Warshall algorithm, which is always 100% in the plots, but is not drawn explicitly.

The input graphs were pseudorandomly generated. For complete graphs, this meant assigning each edge an independently uniformly distributed random length in the range (0, 1). Sparse graphs were generated by starting with an empty graph on 1024 nodes and adding a desired number of edges, which were chosen independently according to the uniform random distribution, and assigned an independently uniformly distributed random length in the range (0, 1). Edge lengths were represented using floating-point numbers in both cases.

The first experiment was for the input of random complete graphs of varying sizes. The results are shown in Fig. 1. The second experiment was for the input of a random graph of 1024 nodes whose number of edges varied from 10 to 80% where $100\% = n^2$. To make the comparison between Floyd-Warshall and the modified versions fairer in the second experiment, we augmented the Floyd-Warshall algorithm with a simple modification, that allowed it to skip combinations i, k where $W[i][k] = \infty$, which reduced the number of path combinations examined. The results of the second experiment are shown in Fig. 2.

In Fig. 1 we can see a significant reduction in terms of path combinations examined. This quantity dominates the algorithm's asymptotic running time and, as observed, decreases compared to the cubic algorithm when inputs grow larger. It might be possible to obtain sub-cubic asymptotic bounds in the average-case model, which is an open question. The experiments on sparse graphs in Fig. 2 show a reduction in path combinations examined as the graph becomes sparser, but the effect on the running time seems to be very minor.

Overall, the Single-tree algorithm is the simplest to implement and offers good performance. The Hourglass algorithm has the potential to be even faster, but would

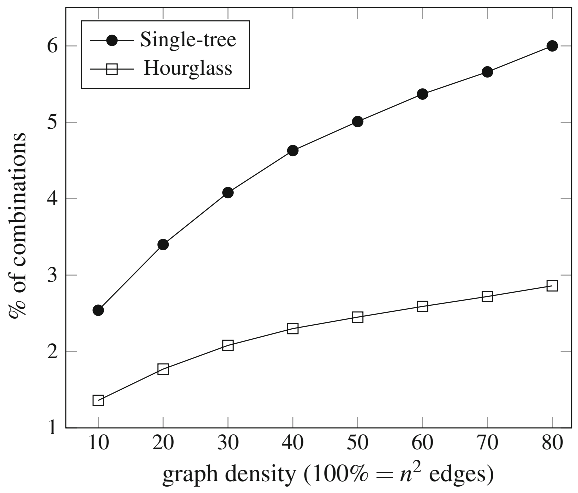

Fig. 2 The percentage of path combinations examined by the two modifications of Floyd-Warshall, when compared to the original algorithm (which is always at 100%, not shown), for the input of a graph with 1024 nodes and various edge densities

likely require a better implementation. It is also worthwhile to note that the additional space requirements for the Single-tree algorithm are very modest, as most applications would typically require storing the path reconstruction matrix regardless.

3 Empirical Comparison of APSP Algorithms

In this section, we analyze the results of empirical tests consisting of running five shortest path algorithms on random graph instances and measuring their running times.

3.1 Graphs and Algorithms

The experiments were conducted on the following two types of directed random graphs.

Uniform random graphs: the edge length is uniformly randomly distributed inside the interval [0, 1). As these graphs grow denser, they start to favor the average-case algorithms, since $m^* = \mathcal{O}(n \lg n)$ with high probability in complete graphs with uniformly distributed random edge lengths [17].

Unweighted random graphs: edge lengths are set to 1. These graphs can be viewed as a type of worst-case for the average-case algorithms, since $m^* = m$ always holds, i.e. a direct edge is the shortest path between two nodes. It should be pointed out, that breadth-first search (BFS) is extremely efficient in solving these instances given its simplicity and $O(mn)$ running time (when solving APSP). However, since we consider these instances only as a worst-case of a more general shortest path problem, we did not include BFS in the comparisons.

In both cases, the graphs were constructed by first setting a desired vertex count and density. Then, a random Hamiltonian cycle is constructed, ensuring that the graph is strongly connected. Edges are added into the graph at random until the desired density is reached. Finally, algorithms are executed on the instance, and their running times recorded. We have explored densities ranging from $m = n^{1.1}$ to $m = n^2$, and vertex counts ranging from $n = 512$ to $n = 4096$. For each density and vertex count combination, we have generated 10 different random instances and averaged the running times of each algorithm.

Priority queues are integral to many shortest path algorithms. Pairing heaps were used in all experiments, since they are known to perform especially well in this capacity in practice. Unlike Fibonacci heaps, which have an $\mathcal{O}(1)$ amortized decrease key operation, the amortized complexity of decrease-key for pairing heaps is $\mathcal{O}(2^{2\sqrt{\lg \lg n}})$ [18]. The following algorithms have been compared:

Dijkstra [19]: solves all-pairs by solving multiple independent single-source problems. Using pairing heaps this algorithm runs in $\mathcal{O}(n^2 \lg n + mn2^{2\sqrt{\lg \lg n}})$.

Floyd-Warshall [10, 11]: classic dynamic programming formulation. Does not use priority queues and runs in $\mathcal{O}(n^3)$.

Propagation: the algorithm described in [9]. In essence, it is a more efficient way of using an SSSP algorithm to solve APSP. The underlying SSSP algorithm is Dijkstra's algorithm. Using pairing heaps this algorithm runs in $\mathcal{O}(n^2 \lg n + m^* n2^{2\sqrt{\lg \lg n}})$.

Single-tree: the algorithm from Sect. 2.1, with the optimizations outlined in Sect. 2.1.1.

Hourglass: the algorithm from Sect. 2.2.

The code has been written in `C++` and compiled using `g++ -march=native -O3`. We have used the implementation of pairing heaps from the Boost Library, version 1.55. All tests were run on an Intel(R) Core(TM) i7-2600@3.40GHz with 8GB RAM running Windows 7 64-bit.

Results are shown as plots where the y axis represents time in milliseconds in logarithmic scale, and the x axis represents the graph edge density as $m = n^x$.

3.2 *Uniform Random Graphs*

The results for uniform random graphs are shown in Figs. 3, 4, 5 and 6.

The tests show that Propagation and Single-tree together outperform the other algorithms on all densities. As the size increases, Hourglass starts catching up to Single-tree, but the constant factors still prove to be too much for it to benefit from its more clever exploration strategy. The running time of Propagation depends on m^* instead of m, and $\frac{m}{m^*}$ in the uniform random graphs increases as the graphs grow

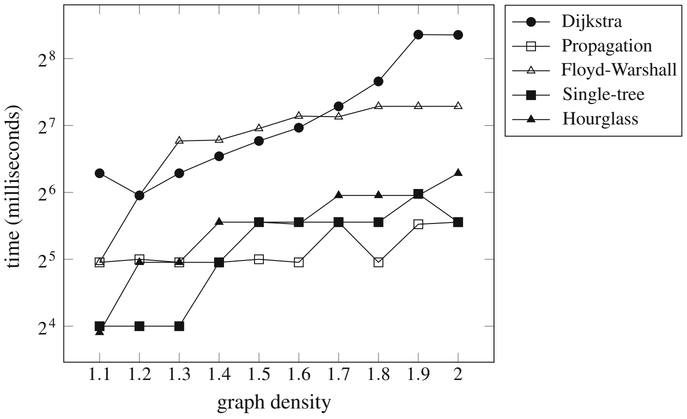

Fig. 3 512 vertices, uniform weights. The plot is quite erratic due to the extremely short running times

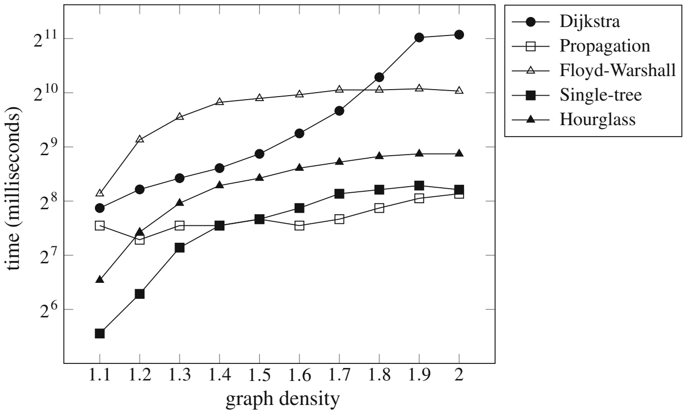

Fig. 4 1024 vertices, uniform weights. The general trend starts to form. Differences between the fastest three algorithms on the sparse instances are quite significant

denser, so it is expected that Dijkstra would be relatively slower the denser the graph is. It is quite surprising that the Single-tree and Hourglass algorithms are so efficient on sparse graphs, outperforming even Dijkstra, something that would seem incredibly difficult given its $O(n^3)$ worst-case time. This would suggest that its average-case time is significantly lower than its worst-case, but no theoretical bounds are known so far.

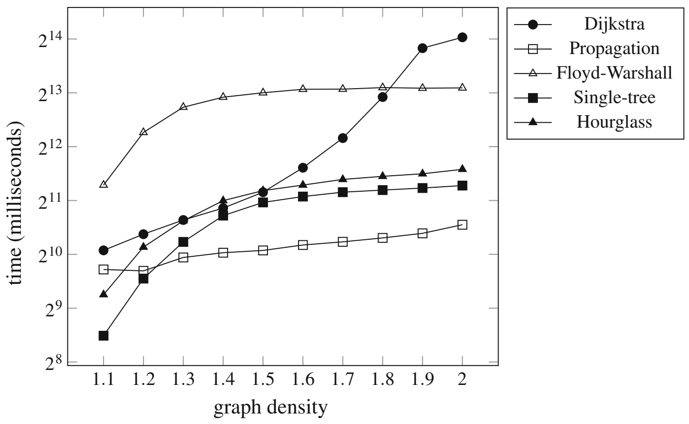

Fig. 5 2048 vertices, uniform weights. Floyd-Warshall's running time begins to increase drastically, as expected due to its cubic complexity. Differences between the fastest three algorithms on sparse instances start to decrease

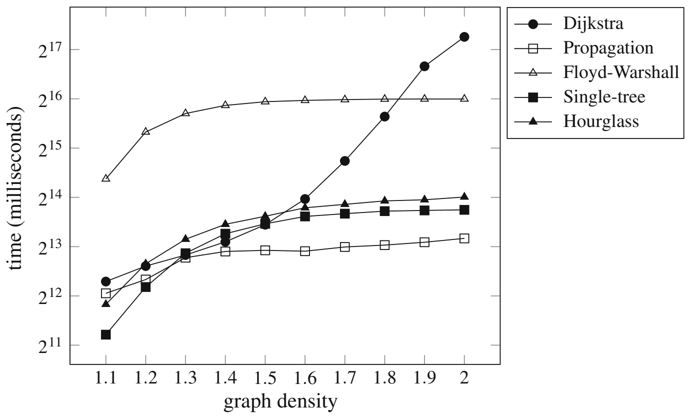

Fig. 6 4096 vertices, uniform weights. Propagation and Single-tree prove to be the fastest, with Single-tree outperforming Propagation on the sparser instances

3.3 Unweighted Random Graphs

The results for unweighted random graphs are shown in Figs. 7, 8, 9 and 10.

In these tests, Propagation performs quite poorly, but that is to be expected since $m = m^*$ in these graphs, resulting in no benefit from Propagation's more clever search strategy compared to Dijkstra. What is interesting is that the Single-tree and Hourglass algorithms are able to remain competitive with Dijkstra in spite of this, and

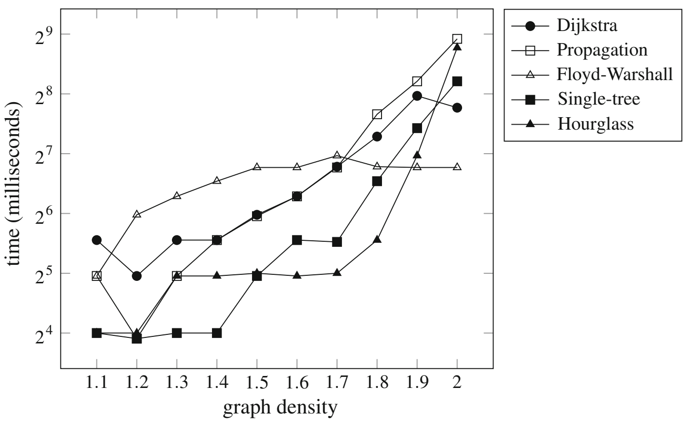

Fig. 7 512 vertices, unweighted. The plot is quite erratic due to the extremely short running times

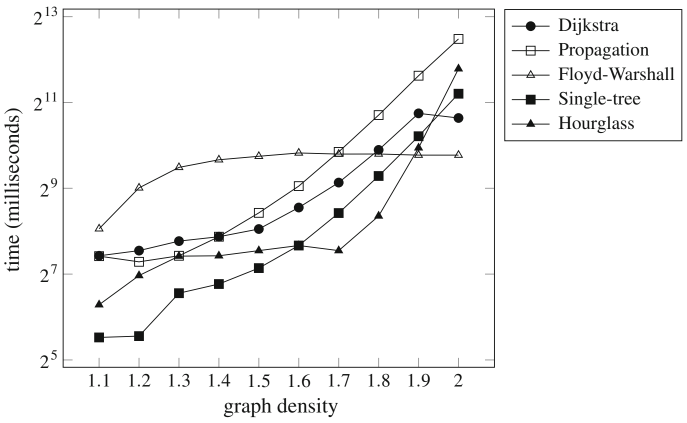

Fig. 8 1024 vertices, unweighted. A clearer picture begins to form, with Single-tree performing surprisingly well and being overtaken by Hourglass briefly as the graph grows dense

even outperforming it on the smaller graphs in some instances. It is worth mentioning that the n^2 case for unit graphs is somewhat pathological, as the instance is already solved since every vertex has a unit-length edge to every other vertex, which can be seen to cause a consistent dip in the running time in the case of Dijkstra.

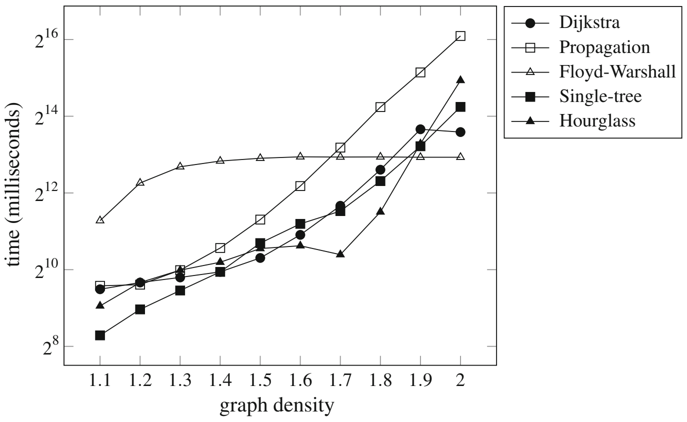

Fig. 9 2048 vertices, unweighted. Hourglass continues to perform best in the 1.7–1.8 range. Differences between the algorithms on the sparse instances begin to decrease, but Single-tree maintains good performance and is matched closely by Dijkstra

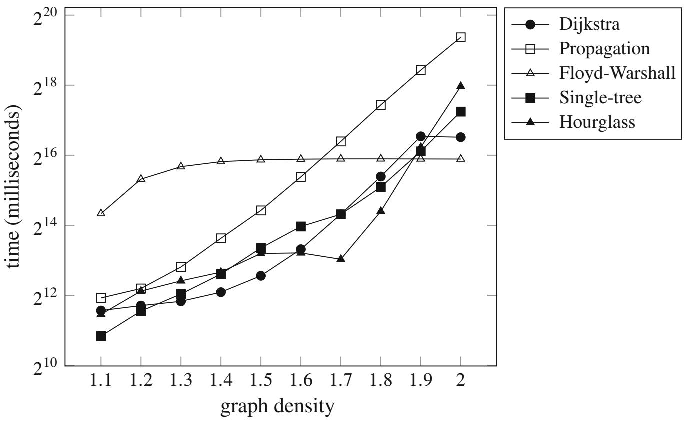

Fig. 10 4096 vertices, unweighted. Single-tree and Dijkstra remain closely matched, and Hourglass continues to dominate the 1.7–1.8 density range

4 All-Pairs Bottleneck Paths

The all-pairs bottleneck paths problem (APBP) is closely related to the all-pairs shortest path problem. The difference is only in the definition of the length of a path π, which is defined to be: $\ell(\pi) = \max_{(u,v)\in\pi} \ell(u, v)$. The length of edges is constrained to be non-negative. A solution to this problem is readily available by simply modifying the relaxation equation of shortest path algorithms to use maximum instead of addition. For example, modifying Dijkstra's algorithm in this way leads to a solution that runs in $O(mn + n^2 \lg n)$ using Fibonacci heaps [20]. A more efficient folklore modification of Dijkstra's algorithm is known to reduce this time down to

$O(mn)$. This folklore modification uses a bucket instead of a heap and sorts the edges by their lengths beforehand in $O(m \lg n)$ time. Then, shortest path lengths can be expressed as monotonically increasing integers from 1...m (referencing the edge lengths), and by using this bucket each SSSP computation takes $O(m)$ time in total. It should be pointed out, that in the case of undirected edges, we can solve APBP on the minimal spanning tree of G instead of on G itself, and still obtain the correct result. This can be a significant speed-up, since $m = n$ for any minimal spanning tree.

In this section, we will describe an algorithm that is more efficient, with an asymptotic running time of $O(m^*n + m \lg n)$. This algorithm will also allow us to state an interesting relationship between APBP and the dynamic transitive closure problem.

Given a graph $G = (V, E)$, the algorithm works by incrementally building the graph $G^* = (V, E^*)$ where $E^* \subset E$ are the edges (u, v) such that $d(u, v) = \ell(u, v)$. It accomplishes this by inserting edges into an initially disconnected set of vertices. The first step is to sort the set of edges E using their lengths. This can be done with any off-the-shelf sorting algorithm in $O(m \lg n)$ time.

Now we consider each edge in this sorted list from smallest to largest. Given an edge (u, v), check if v is reachable from u in G^*. If it is, ignore it and move to the next edge, and if it isn't, add (u, v) to G^*, and for every pair of vertices (w, q) that become reachable, set $d(w, q) = \ell(u, v)$. The algorithm finishes when we have considered every edge.

We summarize the algorithm in pseudocode as Algorithm 4.

Algorithm 4 APBP Algorithm

```
1: procedure APBP($V$, $E$)
2:     Initialize $D$, a $n \times n$ matrix, as $D[i][j] := \infty$.
3:     $E^* := \emptyset$
4:     for all $(u, v) \in E$ from shortest to longest length do
5:         if $D[u][v] = \infty$ then
6:             $E^* := E^* \cup (u, v)$
7:             $D[u][v] := \ell(u, v)$
8:             for all $(x, y)$ where $D[x][y] = \infty$ and $x \to y$ is reachable in $G^* = (V, E^*)$ do
9:                 $D[x][y] := \ell(u, v)$
10:            end for
11:        end if
12:    end for
13: end procedure
```

Lemma 3 *For a graph $G = (V, E)$ the algorithm correctly computes $d(u, v)$: $\forall u, v \in V$.*

Proof By induction on the stage of the algorithm. Let $e_1, e_2, ..., e_n$ be the edges in sorted order, i.e. $\ell(e_1) \leq \ell(e_2) \leq \cdots \leq \ell(e_n)$. Assume the algorithm is at a stage k, i.e. having examined the first $k - 1$ edges. For the case of $k = 1$, the shortest edge in the graph clearly forms the shortest path between the two vertices it connects.

For some case $n \geq k > 1$, let $e_k = (u, v)$ and consider first the case that u and v are already reachable in the current version of the graph G^*. That would imply that $d(u, v) \leq \ell(e_{k-1})$, due to the definition of the length of bottleneck paths, which means a shorter (or equal) path as e_k already exists, thus the edge can be safely omitted as it is redundant.

In the case u cannot yet reach v, then this is the shortest edge to connect the two vertices, and thus clearly $d(u, v) = \ell(e_k)$. For any additional vertex pairs (w, q) that become reachable after e_k is added into the graph, they clearly contain e_k on the path that connects them. Since all the other edges in the graph are shorter, by the definition of the length of bottleneck paths it holds that $d(w, q) = \ell(e_k)$, which completes the proof.

The running time of the algorithm depends heavily on how we check which previously unreachable vertex pairs have become reachable. The following simple approach works when adding some edge (u, v):

1. Gather all vertices that can reach u. This takes $O(n)$ time.
2. For each vertex that can reach u, start a breadth-first exploration of G^* from u, visiting only vertices that were previously not reachable.

Over the entire course of the algorithm, m^* edges are added to G^*, so the time for the first step is $O(m^*n)$. The second step is not more expensive than the cost of each vertex performing a full breadth-first exploration of G^* when it is fully built, thus at most $O(m^* + n)$ per vertex, amounting to $O(m^*n)$ in total. Overall, the cost is $O(m^*n)$.

Combining both times for the edge sorting and reachability checking, we arrive at the bound of $O(m^*n + m \lg n)$. It is worth pointing out that in the case of undirected graphs, G^* corresponds to the minimum spanning tree of G. This is interesting, because it means $m^* = O(n)$, so the running time of the algorithm becomes simply $O(n^2 + m \lg n)$ for undirected graphs. This remains true even if the *representation* is directed, i.e. each edge is simply repeated in both directions with the same length. In some limited sense, the algorithm is adaptive to the input graph.

4.1 Reduction to Decremental Transitive Closure

If instead of adding edges into the graph we consider the opposite scenario, that of removing edges (from largest to smallest) and checking when vertices are no longer reachable, we can reduce the problem to that of decremental transitive closure. In the latter problem, we are given a graph and a series of edge deletions, and the task is to maintain the ability to efficiently answer reachability queries. Relatively recent advancements in decremental transitive closure have led to an algorithm that has a total running time of $O(mn)$ under m edge deletions [21]. This immedately leads to an $O(mn)$ algorithm for all-pairs bottleneck paths. However, since transitive closure can be computed in $O(\frac{mn}{\lg n})$ time [22], a decremental algorithm that matches that running time could also lead to an $o(mn)$ *combinatorial* algorithm for APBP. While subcubic algebraic algorithms for APBP based on matrix multiplication exist [23], no $o(mn)$ combinatorial algorithm is known.

5 Discussion

In this chapter we have looked at practical algorithms for solving the all-pairs shortest path problem. It is typical of the more practically-minded APSP algorithms to rely on average-case properties of graphs, and most of them are modifications of Dijkstra's algorithm. However, the Floyd-Warshall algorithm is known to perform well in practice when the graphs are dense. To this end, we have suggested the Single-tree and Hourglass algorithms: modifications of the Floyd-Warshall algorithm that combine it with a tree structure, that allows it to avoid checking unnecessary path combinations. However, these two algorithms have no known average-case bounds, which would be an interesting topic for further research.

To compare practical performance, we have devised empirical tests using actual implementations. Since, as mentioned, the algorithms studied typically rely on average-case properties of graphs, we looked at both uniform random graphs and unweighted random graphs of varying density. The latter present a hard case for many of the algorithms and can highlight their worst-case performance, whereas the former are much more agreeable to the algorithms' assumptions. For the choice of algorithms we have included those known from past work, as well as the novel Hourglass and Single-tree algorithms. As it turns out, the new algorithms have proven to be quite efficient in the empirical tests that we have performed. The simpler Single-tree algorithm has ranked especially well alongside the Propagation algorithm, while at the same time it was more resilient when it came to worst-case inputs.

In addition, we have also briefly considered the case of all-pairs bottleneck paths, where we proposed a simple algorithm, the asymptotic running time of which can be parametrized with m^*. Additionally, we have shown ties to the decremental transitive closure problem, which might lead to faster algorithms for all-pairs bottleneck paths if faster algorithms for decremental transitive closure are found.

References

1. T.M. Chan, More algorithms for all-pairs shortest paths in weighted graphs. SIAM J. Comput. **39**(5), 2075–2089 (2010)
2. S. Pettie, A new approach to all-pairs shortest paths on real-weighted graphs. Theor. Comput. Sci. **312**(1), 47–74 (2004)
3. S. Pettie, V. Ramachandran, A shortest path algorithm for real-weighted undirected graphs. SIAM J. Comput. **34**(6), 1398–1431 (2005)
4. M. Thorup, Undirected single-source shortest paths with positive integer weights in linear time. J. ACM **46**(3), 362–394 (1999)

5. C. Demetrescu, G.F. Italiano, Experimental analysis of dynamic all pairs shortest path algorithms. ACM Trans. Algorithms **2**(4), 578–601 (2006)
6. D.R. Karger, Random sampling in cut, flow, and network design problems. Math. Oper. Res. **24**(2), 383–413 (1999)
7. Y. Peres, D. Sotnikov, B. Sudakov, U. Zwick, All-pairs shortest paths in $O(n^2)$ time with high probability, in *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science, FOCS '10* (IEEE Computer Societ, Washington, DC, USA, 2010), pp. 663–672
8. D.R. Karger, D. Koller, S.J. Phillips, Finding the hidden path: time bounds for all-pairs shortest paths. SIAM J. Comput. **22**(6), 1199–1217 (1993)
9. A. Brodnik, M. Grgurovic, Solving all-pairs shortest path by single-source computations: theory and practice. Discret. Appl. Math. **231**(Supplement C), 119–130 (2017) (Algorithmic Graph Theory on the Adriatic Coast)
10. R.W. Floyd, Algorithm 97: shortest path. Commun. ACM **5**(6), 345 (1962)
11. S. Warshall, A theorem on boolean matrices. J. ACM **9**(1), 11–12 (1962)
12. G. Venkataraman, S. Sahni, S. Mukhopadhyaya. A blocked all-pairs shortest-paths algorithm. J. Exp. Algorithmics, (8 Dec 2003)
13. S.C. Han, F. Franchetti, M. Püschel, Program generation for the all-pairs shortest path problem, in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, PACT '06* (ACM, New York, NY, USA, 2006), pp. 222–232
14. P. Harish, P.J. Narayanan, Accelerating large graph algorithms on the GPU using CUDA, in *Proceedings of the 14th International Conference on High Performance Computing, HiPC '07* (Springer, Berlin, Heidelberg, 2007), pp. 197–208
15. G.J. Katz, J.T. Kider, All-pairs shortest-paths for large graphs on the GPU, in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, GH '08* (Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2008), pp. 47–55
16. J.J. McAuley, T.S. Caetano, An expected-case sub-cubic solution to the all-pairs shortest path problem in R (2009), arXiv:0912.0975
17. R. Davis, A. Prieditis, The expected length of a shortest path. Inf. Process. Lett. **46**(3), 135–141 (1993)
18. S. Pettie. Towards a final analysis of pairing heaps, in *46th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2005* (IEEE Computer Society, Washington, DC, USA, Oct 2005), pp.174–183
19. E.W. Dijkstra, A note on two problems in connexion with graphs. Numer. Math. **1**, 269–271 (1959)
20. M.L. Fredman, R.E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3), 596–615 (1987)
21. J. Lacki, Improved deterministic algorithms for decremental transitive closure and strongly connected components, in *Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '11* (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2011), pp. 1438–1445
22. T.M. Chan, All-pairs shortest paths with real weights in $O(n^3/\lg n)$ time, in *Proceedings of the 9th International Conference on Algorithms and Data Structures, WADS '05* (Springer, Berlin, Heidelberg, 2005), pp. 318–324
23. V. Vassilevska, R. Williams, R. Yuster, All-pairs bottleneck paths for general graphs in truly sub-cubic time, in *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing, STOC '07* (ACM, New York, NY, USA, 2007), pp. 585–589