Fast Shortest Paths Algorithms in the Presence of Few Negative Arcs

Domenico Cantone and Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone | faro}@dmi.unict.it

Abstract. In this paper we present hybrid algorithms for the single-source shortest paths (SSSP) and the all-pairs shortest paths (APSP) problems which are asymptotically fast when run on graphs with few negative weight arcs. With a directed graph with \( n \) nodes and \( m \) arcs, our algorithm for the SSSP problem has a \( O(\ell(m + n \log n + \ell^2)) \)-time complexity, where \( \ell \) is the minimum between the number of sources and of destinations of negative weight arcs in the graph. In the case of the APSP problem, we propose a \( O(nm^* + n^2 \log n + kn^2) \) algorithm, where \( m^* \) is the number of arcs participating in shortest paths, and \( k \) is the number of endpoints of the arcs in the graph with a negative weight.

1 Introduction

The shortest paths problem on weighted directed graphs is one of the basic network optimization problems. Its importance is mainly due to its applications in various areas, such as communication and transportation. Given a source node \( s \) in a weighted directed graph \( G \), with \( n \) nodes and \( m \) arcs, the single-source shortest path problem from \( s \) (SSSP, for short) is the problem of finding the minimum weight paths from \( s \) to all other nodes of \( G \). The all-pairs shortest paths problem (APSP, for short) consists in finding the minimum weight paths for each pair of nodes in \( G \).

In this paper we propose two algorithms, one for the SSSP problem and one for the APSP problem, which are asymptotically fast when run on graphs with few negative weight arcs.

Negative arc weights arise in several applications, such as in arc- and vertex-diverse path computation, in network survivability, in job scheduling with deadlines, in arbitrage problems in foreign exchange markets, and so on.

If we denote by \( \ell_s \) and \( \ell_r \) the number of nodes which are, respectively, sources and destinations of negative weight arcs in a directed graph with \( n \) nodes and \( m \) arcs, then our proposed algorithm for the SSSP problem has a \( O((\ell(m + n \log n + \ell^2)) \)-time complexity, where \( \ell = \min\{\ell_s, \ell_r\} \). It easily turns out that when \( \ell = o(\sqrt{mn}) \), our algorithm is asymptotically faster than the Bellman-Ford-Moore algorithm [1, 11, 29]. Additionally, for \( \ell = o(h) \), where \( h \) is the minimum between \( n \) and the number of the negative weight arcs contained in the graph, our algorithm is asymptotically faster than Yap’s algorithm [39].
Concerning the APSP problem, if we denote by $k$ the number of the endpoints of the negative weight arcs in a directed graph with $n$ nodes and $m$ arcs (i.e., $k = \ell_o + \ell_r$), then our proposed algorithm has a $O(k^3 + kn^2 + nm^* + n^2 \log n)$-time complexity. It turns out immediately that when $k = o(n)$ and $m^* = o(n^2)$, our algorithm is asymptotically faster than Floyd's algorithm [10]. Moreover, when $k = O(m^*/n + \log n)$, our algorithm achieves the same $O(nm^* + n^2 \log n)$-time complexity of the Hidden-Paths algorithm [23], which however works only with nonnegative weighted graphs. In [23] it is argued that $m^*$ is likely to be small in practice, since $m^* = O(n \log n)$ with high probability for several probability distributions on arc weights.

The paper is organized as follows. After introducing the relevant notations and terminology used in the paper, in Section 2, we briefly review the shortest paths algorithms upon which our solutions are based. Then, in Sections 4 and 5 respectively, we present our algorithms for the SSSP and the APSP problems, deriving also their time complexity and proving their correctness. Finally, we draw our conclusions in Section 6.

2 Preliminaries

We begin by reviewing the relevant notations and terminology. A directed graph is represented as a pair $G = (V, E)$, where $V$ is a finite set of nodes (or vertices) and $E \subseteq V \times V$ is a set of arcs (or edges) such that $E$ does not contain any self-loop of the form $(v, v)$. In this context, we usually put $n = |V|$ and $m = |E|$. A path in $G = (V, E)$ from $u$ to $v$ is any finite sequence $(v_0, v_1, \ldots, v_k)$ of nodes such that $v_0 = u$, $v_k = v$, and $(v_i, v_{i+1})$ is an arc of $G$, for $i = 0, 1, \ldots, k - 1$. By $(u \rightsquigarrow v)$ we denote an unspecified path from $u$ to $v$ (possibly the edge $(u, v)$) and by $(v_0, v_1, \ldots, v_k \rightsquigarrow w)$ (resp., $(w \rightsquigarrow v_0, v_1, \ldots, v_k)$) we denote the concatenation of the paths $(v_0, v_1, \ldots, v_k)$ and $(v_k \rightsquigarrow w)$ (resp., $(w \rightsquigarrow v_0)$ and $(v_0, v_1, \ldots, v_k)$). Likewise, by $(v \rightsquigarrow w \rightsquigarrow u)$ we denote the concatenation of the paths $(v \rightsquigarrow w)$ and $(w \rightsquigarrow u)$. The length of a path is the number of arcs participating in it (with possible repetitions); thus, the length of the path $(v_0, v_1, \ldots, v_k)$ is $k$. An arc $(v_j, v_{j+1})$ in a path $(v_0, v_1, \ldots, v_k)$ is internal if $0 < j < k - 1$. Likewise, a node $v_j$ in a path $(v_0, v_1, \ldots, v_k)$ is internal if $0 < j < k$. A weight function $\omega$ on $G = (V, E)$ is any real function $\omega : E \to \mathbb{R}$. A weight function $\omega$ can be extended over paths by putting $\omega(v_0, v_1, \ldots, v_k) = \sum_{i=0}^{k-1} \omega(v_i, v_{i+1})$. A shortest path from $u$ to $v$ is a path in $G$ whose weight is minimum among all paths from $u$ to $v$. A path $(u \rightsquigarrow v)$ is optimal if it is a shortest path from $u$ to $v$. A path $(v_0, v_1, \ldots, v_k)$ is purely nonnegative if all its arcs have nonnegative weight. Given a weighted graph $(G, \omega)$, with $G = (V, E)$, the reverse of $(G, \omega)$ is the weighted graph $(G^R, \omega^R)$, where $G^R = (V, E^R)$, with $E^R = \{(v, u) \mid (u, v) \in E\}$, and $\omega^R : E^R \to \mathbb{R}$, with $\omega^R(v, u) = \omega(u, v)$, for every $(v, u) \in E^R$.

Provided that $v$ is reachable from $u$ and that no path from $u$ to $v$ goes through a negative weight cycle, a shortest path from $u$ to $v$ always exists; in such a case we denote by $\delta(u, v)$ the weight of a shortest path from $u$ to $v$. If $v$ is not reachable from $u$, we set $\delta(u, v) = +\infty$. If there is a path from $u$ to $v$ through a negative
weight cycle, we put \( \delta(u,v) = -\infty \). The function \( \delta : V \times V \rightarrow \mathbb{R} \cup \{+\infty, -\infty\} \) is called the distance function on \((G,\omega)\). Finally, an arc of a graph \( G \) is said to be optimal if it participates in some shortest path. We denote by \( m^*(G) \) (or simply \( m^* \), when the weighted graph \( G \) is understood from the context) the number of optimal arcs in \( G \). Observe that an arc \((u,v)\) is optimal if and only if \( \omega(u,v) = \delta(u,v) \).

**Definition 1 (Single-Source Shortest Paths Problem)** Given a source node \( s \) in a weighted graph \((G,\omega)\), the single-source shortest paths problem from \( s \) is the problem of computing the shortest path distance \( \delta(s,v) \) on \((G,\omega)\), for all \( v \in V \).

**Definition 2 (All-Pairs Shortest Paths Problem)** Given a weighted graph \((G,\omega)\), the all-pairs shortest paths problem is the problem of computing the shortest path distance \( \delta(u,v) \) on \((G,\omega)\), for all \( u, v \in V \).

Throughout the paper we will assume that graphs satisfy the relation \( n = O(m) \). This is certainly the case, for instance, when all nodes of a graph are reachable from a given source node, which is very common. For ease of presentation, we will also assume that the graphs do not contain negative cycles. However, we observe that our algorithms detect correctly the presence of negative weight cycles (reachable from the source node, in the case of the SSSP problem). Finally, we are interested only in algorithms working on real-weighted graphs, where reals are manipulated only by comparison and addition operations.

Most of the algorithms working on the comparison-addition model are based on the general labeling method. Such method maintains a shortest-path estimate function \( d : V \times V \rightarrow \mathbb{R} \cup \{+\infty\} \), where \( d(u,v) \) represents the shortest path estimate from the source \( u \) to the target \( v \). When the source node \( s \) is fixed, for simplicity we write \( d(v) \) to denote the estimate function \( d(s,v) \).

In the case of the APSP problem, initially one sets \( d(u,v) := \omega(u,v) \) if \((u,v) \in E\) and \( d(u,v) := +\infty \) otherwise (see procedure \texttt{Initialize APSP}(\( G \)).

![Fig. 1. (On the left) procedures \texttt{Initialize APSP} and \texttt{Update} for APSP algorithms, and (on the right) procedures \texttt{Initialize SSSP} and \texttt{Scan} for SSSP algorithms.](image-url)
Fig. 1). Subsequently, the shortest-path estimate \( d \) is updated, within calls to the \( \text{Update}(u, z, v) \) procedure, by assignments of the form \( d(u, v) := d(u, z) + d(z, v) \), provided that \( d(u, v) > d(u, z) + d(z, v) \) holds (see \( \text{Update}(u, z, v) \) in Fig. 1). It turns out that \( d(u, v) \geq \delta(u, v) \) is maintained as an invariant, for each ordered pair \( u, v \in V \).

Concerning the SSSP problem, the shortest-path estimate \( d \) is updated only by assignments of the form \( d(v) := d(u) + \omega(u, v) \), provided that \( d(v) > d(u) + \omega(u, v) \) holds. Also in this case it turns out that \( d(v) \geq \delta(s, v) \) is maintained as an invariant, for \( v \in V \). Procedure \( \text{Scan}(u) \) performs such updates for all arcs \((u, v) \in E \) (see Fig. 1).

3 Previous results

In the case of the SSSP problem, the most general solution is given by the Bellman-Ford-Moore algorithm [1, 11, 29], which is characterized by a \( O(mn) \)-time complexity. However, several algorithms have been proposed over the years to handle efficiently restricted families of directed weighted graphs.

For instance, the case of nonnegative weight functions is solved efficiently by the celebrated Dijkstra’s algorithm [7], which admits a \( O(m + n \log n) \)-time implementation based on Fibonacci heaps [12].

A particular case occurs when the graph is nearly acyclic and negative weight arcs are allowed only outside of any cycle. In this case, the Two-Levels-Greedy algorithm proposed in [2], which is a natural combination of Dijkstra’s algorithm with the linear algorithm for acyclic graphs, solves the SSSP problem in \( O(m + n \log k) \)-time complexity, where \( k \) is the maximum cardinality of any strongly connected component in the graph.

For planar graphs, there has been a series of results yielding progressively better bounds. The first algorithm that exploited planarity was due to Lipton, Rose, and Tarjan [26], who gave a \( O(n^{3/2}) \) algorithm. Henzinger et al. [24] proposed a \( O(n^{3/2} \log^{3/4} D) \) algorithm, where \( D \) is the sum of the absolute values of the weights. Subsequently, Fakcharoenphol and Rao [9] gave an algorithm requiring \( O(n \log^2 n) \) time and \( O(n \log n) \) space. Finally, Klein et al. presented in [25] an algorithm for planar graphs with negative weights requiring linear space and \( O(n \log^3 n) \) time.

A restriction, particularly relevant to the present paper, concerns the number of negative weight arcs contained in the graph. In [39], C.K. Yap describes a hybrid algorithm for the shortest path problem between two nodes in a directed graph in the presence of few negative weight arcs, whose running time is \( O(h(m + n \log n + h^2)) \), where \( h \) is the minimum between \( n \) and the number of the negative weight arcs contained in the graph.

In the case of the APSP problem, the general case is classically solved by the dynamic programming algorithm proposed by Floyd [10] in 1962, which has a \( \Theta(n^3) \)-running time. The first subcubic algorithm was Fredman’s algorithm [13], characterized by a \( O(n^3 \log^{1/3} \log n / \log^{1/3} n) \)-time complexity. Several subcubic solutions have been presented over the years [8, 35, 16, 36, 40, 3, 37, 17, 4]. Among
the more recent results, the $O(n^3/\log n)$-time algorithm of Chan [3] deserves a special mention since it is based on a simple geometric approach and does not require explicit table lookups or word tricks. Moreover it must be noticed that the algorithm of Han [17] was the first to break the $O(n^3/\log n)$ barrier by exploiting sophisticated word-packing tricks. The best result known to date is due to Han and Takaoka [18] which achieves a $O(n^3 \log \log n/\log^2 n)$-time complexity. The latter result is very close to the running time $O(n^3/\log^2 n)$ which is currently believed to be the natural limit of combinatorial approaches to the APSP problem.

In the restricted case in which the graph is sparse, the APSP can be solved more efficiently by Johnson’s algorithm [22], which, after a preliminary reweight of the graph based on the shortest path weights computed by a run of the Bellman-Ford-Moore algorithm from an external source, applies Dijkstra’s algorithm from each node of the graph. If priority queues are implemented with Fibonacci heaps, it has a $O(mn + n^2 \log n)$-time complexity and turns out to be asymptotically faster than Floyd’s algorithm when $m = o(n^2)$. Recently the complexity of Johnson’s algorithm has been lowered to $O(mn + n^2 \log \log n)$ and $O(mn + n^2 \alpha(m, n))$ for directed and undirected graphs, respectively, by Pettie [33] and Pettie and Ramachandran [34], by applying Thorup’s hierarchy-based approach [38] to real-weighted graphs.

For the case of nonnegative weights, we mention the Hidden-Paths algorithm presented in [23], and developed independently by [28, 20], which solves the APSP problem in $O(nm^* + n^2 \log n)$-time, provided that one uses Fibonacci heaps to implement a priority queue, where $m^*$ is the number of optimal arcs (i.e., arcs participating in shortest paths). The Hidden-Path algorithm operates by running Dijkstra’s SSSP algorithm in parallel from all nodes in the graph, using information gained at each node to reduce the work done at other nodes. When the input graph is the complete graph with arc weights chosen independently from any of a large class of probability distributions, it turns out that $m^* = O(n \log n)$ with high probability (cf. [14, 30]). Thus, under this assumption, the Hidden-Path algorithm achieves a $O(n^2 \log n)$-time complexity on average (with high probability).

For the sake of completeness, we will briefly review below the shortest paths algorithms which are used as building blocks in the design of our algorithms, to be presented in Sections 4 and 5. These are the Bellman-Ford algorithm, Dijkstra’s algorithm, Floyd’s algorithm, and the Hidden-Paths algorithm.

### 3.1 The Bellman-Ford algorithm

In the general case, namely when the underlying graph is not required to satisfy any particular topological restriction and the weight function is not restricted in any way, the best known complexity bound for the SSSP problem is reached by the $O(mn)$-time algorithm due to Bellman [1], Ford [11], and Moore [29]. The algorithm maintains the set of nodes $Q$ into a FIFO queue. Thus, the next node to be scanned is removed from the head of $Q$, whereas a node whose shortest
Dijkstra(G, s)
Initialize_SSSP(G, s)
Q := Make_Queue(V, d)
while Q ≠ ∅ do
  v := Extract_Min(Q)
  Scan(v)

Generalized-Dijkstra(G, s)
Initialize_SSSP(G, s)
Q := Make_Queue({s}, d)
while Q ≠ ∅ do
  v := Extract_Min(Q)
  Scan(v)
  for all u ∈ nodes(G) : (v, u) ∈ arcs(G) do
    if S(u) = Labeled and u /∈ Q then
      Insert(u, Q)

Fig. 2. Dijkstra’s algorithm and Generalized Dijkstra’s algorithm

path estimate decreases is added to the tail of Q. It is to be noticed, though, that other variants [15, 5, 32, 31] of the Bellman-Ford-Moore algorithm show a better practical behavior than the original algorithm.

3.2 Dijkstra’s algorithm

Several algorithms have been proposed over the years to handle efficiently restricted families of directed weighted graphs. For instance, Dijkstra’s algorithm [7] solves the SSSP problem for directed graphs with nonnegative weight arcs only (see Fig. 2). It maintains all nodes in a min-priority queue, according to their shortest-path estimates. Then, at each iteration, it extracts the node u from the queue with the minimum shortest-path estimate d(u), following a greedy strategy, and it executes procedure Scan(u) in Fig. 1.

The complexity of Dijkstra’s algorithm depends on the implementation used for its service priority queue. A first naive implementation given in [7], in which the priority queue is maintained as a simple list, has a $O(n^2)$-time complexity. By implementing the priority queue by k-ary heaps [6] (for fixed k), one obtains a $O(m \log n)$-time algorithm. Finally, we mention that using Fibonacci heaps [12] one gets a $O(m + n \log n)$-time implementation.

It can be shown that in the case of nonnegative weighted graphs, every node u, when selected and removed from the queue by Dijkstra’s algorithm, satisfies $d(u) = \delta(s, u)$, so that it does not need to be scanned anymore. However, if negative weights are allowed, this is no longer true. Such a case can be handled by (re)inserting in the queue, if needed, any node which becomes Labeled, even if it has already been scanned. As a node may re-enter the queue several times, it turns out that the resulting Generalized-Dijkstra algorithm, reported in Fig. 2, has an exponential worst-case running time, as shown in [21].

3.3 Floyd’s algorithm

Let $G = (V, E)$ be a directed graph, with $V = \{v_1, v_2, \ldots, v_n\}$, and let $w : E \to \mathbb{R}$ be a real-valued weight function. For $k = 1, 2, \ldots, n$, let $V_k = \{v_1, v_2, \ldots, v_k\}$.
Also, put $V_0 = \emptyset$. Using a dynamic programming approach, Floyd’s algorithm [10] computes a sequence of real functions $d^0, d^1, \ldots, d^n$ over $V \times V$, where, at least in the case in which $G$ contains no negative weight cycle, $d^k(u, v)$ is the shortest path distance from vertex $u$ to vertex $v$ restricted to paths whose internal nodes are in $V_k$, if any such path exist, otherwise $d^k(u, v) = +\infty$, for $0 \leq k \leq n$ and $u, v \in V$. Observe that, for each pair of nodes $u, v$, we have
\[ d^0(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \in E \\ +\infty & \text{otherwise} \end{cases} \]
and $d^k(u, v) = \delta(u, v)$, for any $k \geq n$. In particular at iteration $i$ of the algorithm, the estimate function $d^i$ is computed by executing procedure \text{Update}$(u, v_{i+1}, v)$, for all $u, v \in V$, according to a dynamic programming approach.

### 3.4 The Hidden-Paths algorithm

The Hidden-Paths algorithm [23] solves the APSP problem in the restricted case of nonnegative weighted graphs by running Dijkstra’s algorithm in parallel from all nodes of the graph, in such a way that the partial results of any single-source thread can be used to reduce the work done by the other threads.

For a weighted graph $(G, \omega)$, with $G = (V, E)$, the Hidden-Paths algorithm maintains a min-heap of the best path from $u$ to $v$ found so far, for each ordered pair of distinct nodes $u, v \in V$. The queue is arranged according to shortest-path estimate values and, for paths with the same weight, according to path length. It is initialized with all pairs of distinct nodes $(u, v)$ (considered as paths of length 1) with weight $\omega(u, v)$, where we set $\omega(u, v) = +\infty$ for any $(u, v) \notin E$.

Let $Opt$ be initialized to the empty set. At each phase, the Hidden-Paths algorithm extracts the path at the top of the heap, say $(u \rightsquigarrow v)$, and add it to $Opt$. It turns out that the extracted path $(u \rightsquigarrow v)$ is optimal, so that $Opt$ is the collection of the optimal paths found so far. The extracted path $(u \rightsquigarrow v)$ can then be used to construct a set of new candidate optimal paths by calling procedure \text{Update}$(z, u, v)$, for each (optimal) arc $(z, u) \in Opt \cap E$, and, provided that $(u \rightsquigarrow v)$ has length 1 (i.e., it comprises of a single arc), by also calling procedure \text{Update}$(u, v, z)$, for each path $(v \rightsquigarrow z)$ in $Opt$.

The complexity of the Hidden-Paths algorithm depends on the implementation of the heap. Using Fibonacci heaps, it has a complexity of $O(m^* + n^2 \log n)$, where $m^*$ is the number of optimal arcs in the weighted graph $(G, \omega)$.

### 4 A fast single-source shortest-paths algorithm in the presence of few sources or destinations of negative arcs

In this section we present a new algorithm for the single-source shortest path problem in the presence of few sources or destinations of negative arcs, by generalizing Yap’s approach [39].
We begin with some notations. As before, let \( G = (V, E) \) be a directed graph with weight function \( \omega : E \to \mathbb{R} \) and source \( s \in V \), and let
\[
e_1 = (p_1, q_1), \; e_2 = (p_2, q_2), \; \ldots, \; e_\eta = (p_\eta, q_\eta)
\]
be the negative-weight arcs in \( G \). Let \( S_G = \{p_1, p_2, \ldots, p_\eta\} \) be the set of sources of the negative arcs in \( G \) and let \( T_G = \{q_1, q_2, \ldots, q_\eta\} \) be the set of destinations of the negative arcs in \( G \). Next, we define the hinge set \( H^S \) of \( (G, \omega) \), by putting
\[
H^S = \begin{cases} S_G & \text{if } |S_G| < |T_G| \\
T_G & \text{otherwise}
\end{cases}
\]
and the extended hinge set \( \overline{H}^S \) of \( (G, w) \), by putting \( \overline{H}^S = H^S \cup \{s\} \). Let \( \ell = |H^S| \) and \( \overline{\ell} = |\overline{H}^S| \), so that \( |H^S| = \ell \) and \( \ell \leq \overline{\ell} \leq \ell + 1 \). Finally, let \( s_1, s_2, \ldots, s_{\overline{\ell}} \) be the distinct elements of \( \overline{H}^S \), where \( s_1 \) is the source \( s \) and \( \overline{\ell} = |\overline{H}^S| \).

We present now our algorithm for the SSSP problem.

**ALGORITHM 1**

[Step 1]. For each \( s_i \in \overline{H}^S \), apply Dijkstra’s algorithm to \( (G, \omega) \) with source node \( s_i \). Let \( d_1(s_i, v) \) be the distance function computed by the call to Dijkstra’s algorithm from source \( s_i \), with \( v \in V \).

(Notice that since \( G \) contains negative weight arcs, in general \( d_1(s_i, v) \neq \delta(s_i, v) \).)

[Step 2]. Let \( \overline{G} = (\overline{H}^S, E) \) be the directed graph with
\[
E = \{(s_i, s_j) \mid i, j = 1, \ldots, \ell, \; i \neq j, \; d_1(s_i, s_j) \neq +\infty\}
\]
and let \( \overline{\omega} \) be the weight function on \( \overline{G} \) defined by \( \overline{\omega}(s_i, s_j) = d_1(s_i, s_j) \), for \( (s_i, s_j) \in E \).

Apply the Bellman-Ford-Moore algorithm to the weighted graph \( (\overline{G}, \overline{\omega}) \) from the source \( s_1 = s \) and let \( d_2(v) \) be the distance function so computed. If negative-weight cycles reachable from \( s_1 \) in \( (\overline{G}, \overline{\omega}) \) are detected, notify the presence in \( (G, \omega) \) of negative-weight cycles reachable from the source node and exit. Otherwise, go to Step 3.

(As will be shown later, if no negative-weight cycle is detected, then \( d_2(s_i) = \delta(s_i, s_1) \), for \( i = 1, \ldots, \ell \).)

[Step 3]. Initialize the shortest-path estimate \( d(v) \), for each \( v \in V \), by setting
\[
d(v) = \begin{cases} d_2(s_i) & \text{if } v = s_i, \; \text{for some } i = 1, \ldots, \ell \\
+\infty & \text{otherwise}
\end{cases}
\]

Then call procedure \( \text{SCAN}(s_i) \) for each \( s_i \in \overline{H}^S \).

Let \( d_3(v) \), for \( v \in V \), be the distance function computed by Step 3.
[Step 4]. Apply Dijkstra’s algorithm to the weighted graph \((G, \omega)\) with source node \(s\), where for each \(v \in V\) the shortest-path estimate \(d(v)\) is initialized with the value \(d_3(v)\) computed in Step 3, and return the distance function \(d_4(v)\), for \(v \in V\), so computed.

In Section 4.2 we will show that Algorithm 1 is correct, namely that \(d_4(v) = \delta(s, v)\) holds, for every \(v \in V\).

Remark 1. We observe that in the particular case in which the hinge set of \(G\) is equal to \(T \setminus G\), Steps 3 and 4 in Algorithm 1 can be replaced by the following simplified

[Step 3’]. Initialize the shortest-path estimate \(d(v)\), for each \(v \in G\), as follows

\[ d(v) = \begin{cases} 
  d_2(s_i) & \text{if } v = s_i, \text{ for some } i = 1, \ldots, \bar{\ell} \\
  +\infty & \text{otherwise}, 
\end{cases} \]

apply Dijkstra’s algorithm to the weighted graph \((G, \omega)\) with source node \(s\), and return the distance function so computed.

However, it turns out that such a simplification does not affect the asymptotic behavior of Algorithm 1.

4.1 Complexity issues

Since, as noted before, we have \(\ell \leq \bar{\ell} \leq \ell + 1\), then the \(\bar{\ell}\) applications of Dijkstra’s algorithm in Step 1 take a total time complexity of \(O(\ell(m + n \log n))\), provided that we use Fibonacci heaps to implement the service priority queue. In addition, Step 2 takes \(O(\ell^3)\) time, whereas Step 3 takes \(O(\ell + m)\) time. Finally, the last application of Dijkstra’s algorithm in Step 4 takes an additional \(O(m + n \log n)\) time. Summing up, Algorithm 1 has an overall \(O(\ell(m + n \log n + \ell^2))\)-time complexity.

If \(\ell = o(\sqrt[3]{mn})\) (and \(n = O(m)\)—we have assumed such relationship throughout the paper), then Algorithm 1 is asymptotically faster than the Bellman-Ford-Moore algorithm. This can be shown easily by observing that if \(\ell = o(\sqrt[3]{mn})\) then we have \(\bar{\ell} = o(mn)\) and \(\ell m = o(mn)\), since \(m = O(n^2)\) so that \(\ell = o(\sqrt[3]{n^2m}) = o(n)\). We also have \(\bar{\ell} n \log n = o(mn)\); indeed, the assumption \(n = O(m)\) yields \(n \sqrt[m]{m} \log n = O(n \sqrt[m]{m^2 \log m})\); in addition, as \(\sqrt[m]{m^2 \log m} = O(m)\) we have \(n \sqrt[m]{m^2 \log m} = O(mn)\). Thus, since \(\bar{\ell} n \log n = o(n \sqrt[m]{mn} \log n)\), we have \(\bar{\ell} n \log n = o(mn)\). The above considerations yield immediately that \(\ell(m + n \log n + \ell^2) = o(mn)\), provided that \(\ell = o(\sqrt[3]{mn})\).

Next we compare Algorithm 1 with Yap’s algorithm [39]. Notice that \(\ell = O(h)\), where we recall that \(h\) denotes the minimum between \(n\) and the number of the negative weight arcs in the graph. Therefore we have \(\ell(m + n \log n + \ell^2) = O(h(m + n \log n + h^2))\), i.e., Algorithm 1 is always asymptotically at least as
good as Yap’s one. In addition, if \( \ell = o(h) \), then \( \ell(m + n \log n + \ell^2) = o(h(m + n \log n + h^2)) \), i.e., in this case Algorithm 1 is asymptotically faster than Yap’s algorithm.

### 4.2 Correctness proof

For the sake of simplicity, we will limit our considerations only to the case in which no negative weight cycle is reachable from the given source node \( s \).

Algorithm 1 exploits the fact that even when negative weight arcs are allowed, the standard Dijkstra’s algorithm calculates correctly the distances from the source node to all safe nodes. These are defined as follows.

**Definition 3 (Safe nodes and paths)** Let \( G = (V, E) \) be a directed graph with a weight function \( \omega : E \to \mathbb{R} \), and let \( s, v \in V \). We say that node \( v \) is safe relative to \( s \) in \((G, \omega)\) if there exists a shortest path \( \pi \) from \( s \) to \( v \), called safe path for \( v \) relative to \( s \), whose internal arcs have a nonnegative weight (thus, only the first and the last arc of a safe path \( \pi \) are allowed to have a negative weight).

In order to prove that safe nodes are handled correctly by Dijkstra’s algorithm, we state and prove the following preliminary lemma.

**Lemma 4** Let \( G = (V, E) \) be a directed graph with a weight function \( \omega : E \to \mathbb{R} \) containing no negative-weight cycle reachable from a given source \( s \in V \). Let \( v \in V \) and let \( \pi \) be a shortest path from \( s \) to \( v \) whose arcs have a nonnegative weight, with the only possible exception of the first arc. Then, when node \( v \) is extracted from the queue during the execution of Dijkstra’s algorithm on the graph \( G \) from source \( s \), \( d(v) = \delta(s,v) \) holds.

**Proof.** Let \( v \in V \) and let \( \pi \) be a path from \( s \) to \( v \) satisfying the hypotheses of the lemma, but assume by contradiction that \( d(v) \neq \delta(s,v) \) when the node \( v \) is extracted from the queue, during the execution of Dijkstra’s algorithm on \((G, \omega)\) from the source \( s \). Then \( v \neq s \), as we are assuming that no negative-weight cycle is reachable from the source \( s \), so that \( d(s) = 0 = \delta(s,s) \) holds when \( s \) is extracted from the queue.

Plainly, \( d(v) > \delta(s,v) \) holds when \( v \) is extracted from the queue, since the invariant \( d(u) \geq \delta(s,u) \) is maintained for every node \( u \) during the execution of any shortest path algorithm based on the labeling method (cf. [6]). Without loss of generality, we can assume that \( v \) is the closest node to \( s \) on the path \( \pi \) from \( s \) to \( v \) which satisfies \( d(u) \neq \delta(s,u) \), when extracted from the queue.

Let \( S_v \) be the collection of nodes on the path \( \pi \) which have not yet been extracted from the queue when the node \( v \) is extracted (maybe because they did not even enter the queue at that time). The set \( S_v \) is nonempty, as the predecessor \( w \) of \( v \) in \( \pi \) must belong to it. Indeed, if this were not the case, then after the execution of \( \text{SCAN}(w) \) following the extraction of \( w \) from the queue, we would have \( d(v) = \delta(s,v) \). Let then \( w' \) be the closest node to \( s \) on the path
Let \( G = (V, E) \) be a directed graph with a weight function \( \omega : E \rightarrow \mathbb{R} \), and let \( s, v \in V \). Let us assume that \( v \) is safe relative to \( s \). Then at the end of the execution of Dijkstra’s algorithm we have \( d(v) = \delta(s, v) \).

**Proof.** Let \( \pi \) be a safe path for \( v \) relative to \( s \). If \( v = s \), then we simply have \( d(v) = \delta(s, v) = 0 \). Otherwise, let \( u \) be the predecessor of \( v \) on \( \pi \). By Lemma 4, \( d(u) = \delta(s, u) \) holds when the node \( u \) is extracted from the queue. Thus, after the subsequent execution of procedure \( \text{SCAN}(u) \) we have also \( d(v) = \delta(s, v) \) (regardless of the fact that the node \( v \) might already been extracted from the queue).

Let \( G = (V,E), \omega : E \rightarrow \mathbb{R}, s \in V, \) and \( \overline{P}^5 = \{s_1, s_2, \ldots, s_t\} \) (with \( s_1 = s \)) be as in the opening of Section 4. Let \( s_i \in \overline{P}^5, v \in V, \) and let \( (s_i \leadsto v) \) be a shortest path in \( G \) from \( s_i \) to \( v \).

After the execution of Step 1 of Algorithm 1, Theorem 5 implies that \( d_1(s_i, v) = \delta(s_i, v) \) holds, provided that no internal node of the path \( (s_i \leadsto v) \) belongs to \( \overline{P}^5 \) (where we recall that \( d_1(s_i, v) \) denotes the distance function computed by the call to Dijkstra’s algorithm from source \( s_i \), within Step 1).

If \( (s_i \leadsto v) \) has the form \( (s_i \leadsto s_j \leadsto v) \), where both of its subpaths \( (s_i \leadsto s_j) \) and \( (s_j \leadsto v) \) contain no node in \( \overline{P}^5 \) among their internal nodes, then \( s_j \) is safe relative to \( s_i \) and \( v \) is safe relative to \( s_j \), so that by two applications of Theorem 5 we have

\[
\delta(s_i, v) = \delta(s_i, s_j) + \delta(s_j, v) = d_1(s_i, s_j) + d_1(s_j, v).
\]

By repeating the above reasoning, it is easy to prove the following more general result.

**Lemma 6** Let \( s_i \in \overline{P}^5 \) and \( v \in V \), and let \( (s_i \leadsto s_{i_1} \leadsto \ldots \leadsto s_{i_t} \leadsto v) \) be a shortest path in \( (G, \omega) \) from \( s_{i_0} \) to \( v \) such that all of its subpaths \( (s_{i_j} \leadsto s_{i_{j+1}}), \) for \( j = 0, 1, \ldots, t-1 \), and \( (s_{i_t} \leadsto v) \) contain no node in \( \overline{P}^5 \) among their internal nodes. Then \( \delta(s_{i_0}, v) = \sum_{j=0}^{t-1} d_1(s_{i_j}, s_{i_{j+1}}) + d_1(s_{i_t}, v) \).
In the particular case in which in the previous lemma we have \( s_i = s \) and \( v = s_{i+1} \in H_5 \), then we obtain \( \delta(s, s_{i+1}) = \sum_{j=0}^{i} d_1(s_j, s_{j+1}) \).

But, by Step 2, \( \sum_{j=0}^{i} d_1(s_j, s_{j+1}) \geq d_2(s_{i+1}) \geq \delta(s, s_{i+1}) \), so that \( d_2(s_{i+1}) = \delta(s, s_{i+1}) \), proving the following result.

**Corollary 7** After the execution of Step 2 of Algorithm 1, we have \( d_2(s_i) = \delta(s, s_i) \), for every \( s_i \in \mathcal{H}_5 \).

Thus, after the execution of Step 2 all distances from the source \( s \) to each node in the hinge set \( H_5 \) have been correctly computed. Next we show that Step 3 and 4 propagate the correct distance estimation also to the remaining nodes of the graph.

**Theorem 8 (Correctness)** Let \( d_2(v) \) be the distance function computed by Algorithm 1 (by the last call to Dijkstra’s algorithm in Step 4). Then \( d_2(v) = \delta(s, v) \) holds, for each \( v \in V \).

**Proof.** Let \( \text{Adj}(H_5) = \{v \in V \mid (s_i, v) \in E\} \) for some \( s_i \in H_5 \) be the set of all the nodes in \( G \) adjacent to some node in the hinge set \( H_5 \) of \((G, \omega)\).

It is convenient to define the following auxiliary graph \( \tilde{G} = (V, \tilde{E}) \) with weight function \( \tilde{\omega} : \tilde{E} \rightarrow \mathbb{R} \), where \( \tilde{E} = E \cup \{(s, v) \mid v \in (H_5 \cup \text{Adj}(H_5)) \setminus \{s\}\} \) and

\[
\tilde{\omega}(u, v) = \begin{cases} 
   d_3(v) & \text{if } u = s \text{ and } v \in (H_5 \cup \text{Adj}(H_5)) \setminus \{s\} \\
   \omega(u, v) & \text{otherwise.}
\end{cases}
\]

Let \( \tilde{d} \) be the distance function computed by an execution of Dijkstra’s algorithm on the weighted graph \((\tilde{G}, \tilde{\omega})\) from the source \( s \).

It is easy to check that the execution in Step 4 of Dijkstra’s algorithm on \((G, \omega)\) from \( s \) when the shortest-path estimate \( d(v) \) has been initialized to \( d_3(v) \), for \( v \in V \), is equivalent to an execution of the standard Dijkstra’s algorithm on \((\tilde{G}, \tilde{\omega})\) from \( s \). Hence we have \( \tilde{d}(v) = d_2(v) \), for \( v \in V \). It then follows that it is enough to show that \( \tilde{d}(v) = \delta(s, v) \), for \( v \in V \), in order to prove the theorem.

Thus, let \( v \in V \) and let \((s \rightsquigarrow v)\) be a shortest path in \((\tilde{G}, \tilde{\omega})\). If \( v \in (H_5 \cup \text{Adj}(H_5)) \setminus \{s\} \), then \((s, v)\) is an optimal arc of \((\tilde{G}, \tilde{\omega})\), which is obviously safe for \( v \) relative to \( s \). Otherwise, \((s \rightsquigarrow v)\) has the form \((s \rightsquigarrow w \rightsquigarrow v)\), for a suitable node \( w \in (H_5 \cup \text{Adj}(H_5)) \setminus \{s\} \) such that the subpath \((w \rightsquigarrow v)\) contains no node in \((H_5 \cup \text{Adj}(H_5)) \setminus \{s\}\) among its internal nodes. Then \((s, w \rightsquigarrow v)\) is a safe path for \( v \) relative to \( s \) in \((\tilde{G}, \tilde{\omega})\). In both cases, \( v \) is safe relative to \( s \) in \((\tilde{G}, \tilde{\omega})\), so that by Theorem 5 it follows that \( \tilde{d}(v) = \delta(s, v) \), completing the proof of the theorem.

\footnote{We recall that \( d_2 \) is the distance function computed by the call to the Bellman-Ford-Moore algorithm within Step 2.}
5 A fast all-pairs shortest-paths algorithm in the presence of few negative arcs

In this section we present a new algorithm for the APSP problem in the presence of few negative-weight arcs. Our algorithm is a hybridization of Dijkstra’s, Floyd’s, and the Hidden-Paths algorithms, and is obtained by generalizing the approach for the single-source case presented in the previous section.

As before, let \( G = (V, E) \) be a directed graph, with \( n = |V| \) and \( m = |E| \), let \( \omega: E \to \mathbb{R} \) be a weight function, and let \( e_1 = (p_1, q_1), e_2 = (p_2, q_2), \ldots, e_\eta = (p_\eta, q_\eta) \) be the negative weight arcs in \( G \). Also, let \( S_G^- = \{p_1, p_2, \ldots, p_\eta\} \) and \( T_G^- = \{q_1, q_2, \ldots, q_\eta\} \) be respectively the sets of sources and of destinations of the negative arcs in \( G \).

In the present case of the APSP problem, we define the hinge set \( H^A \) of \((G, \omega)\) as \( H^A = S_G^- \cup T_G^- \). Let then \( k = |H^A| \) and \( H^A = \{s_1, \ldots, s_k\} \). Plainly, \( k \leq 2\eta \), where \( \eta \) is the number of negative arcs in \( G \).

We are now ready to describe our algorithm for the APSP problem.

**ALGORITHM 2**

**Step 1.** Apply Dijkstra’s algorithm to \((G, \omega)\) from each source node \( s_i \in H^A \). Let \( d_1(s_i, v) \) be the distance functions so computed, for \( s_i \in H^A \) and \( v \in V \).

(Notice that due to the presence of negative-weight arcs, the distances computed by Step 1 are not necessarily correct.)

**Step 2.** Let \((\tilde{G}, \tilde{\omega})\) be the weighted graph where \( \tilde{G} = (H^A, \tilde{E}) \) is the directed graph (with no self-loops) over \( H^A \), with \( \tilde{E} = \{(s_i, s_j) \mid i, j = 1, \ldots, k, i \neq j \} \), and \( \tilde{d}_1(s_i, s_j) \neq +\infty \), and \( \tilde{\omega}(s_i, s_j) = d_1(s_i, s_j) \), for all \((s_i, s_j) \in \tilde{E}\). Apply Floyd’s algorithm to the weighted graph \((\tilde{G}, \tilde{\omega})\), and let \( d_2(s_i, s_j) \) be the distances so computed, for \( i, j = 1, \ldots, k \) with \( i \neq j \).

If negative-weight cycles are detected in \((\tilde{G}, \tilde{\omega})\), notify the presence of negative-weight cycles in \((G, \omega)\) and stop. Otherwise, go to Step 3.

(We plainly have \(|\tilde{E}| \leq k(k-1) < k^2\). As will be shown later, when no negative-weight cycle is present in \((\tilde{G}, \tilde{\omega})\), then \( d_2(s_i, s_j) = \delta(s_i, s_j) \), for all \( i, j = 1, \ldots, k \) with \( i \neq j \).)

**Step 3.** Let \((\tilde{G}, \tilde{\omega})\) be the weighted graph obtained by superimposing \((\tilde{G}, \tilde{\omega})\) to \((G, \omega)\) in the following way. Let \( \tilde{E} = E \cup \tilde{E} \). Then put \( \tilde{G} = (V, \tilde{E}) \) and

\[
\tilde{\omega}(u, v) = \begin{cases} 
\tilde{d}_2(u, v) & \text{if } (u, v) \in \tilde{E} \\
\omega(u, v) & \text{otherwise},
\end{cases}
\]

for \((u, v) \in \tilde{E}\).

Apply Dijkstra’s algorithm to the reverse \((\tilde{G}^R, \tilde{\omega}^R)\) of \((\tilde{G}, \tilde{\omega})\), from each source
node \(s_i \in H^A\).
Let \(d_3'(s_i, v)\) be the distance function so computed on \((\tilde{G}^R, \tilde{\omega}^R)\), for \(s_i \in H^A\)
and \(v \in V\), and let \(d_3\) be the reverse of \(d_3'\), i.e., \(d_3(v, u) = d_3'(u, v)\) for each
\((u, v) \in \tilde{E}\) (so that \(d_3\) is a distance estimate on \((\tilde{G}, \tilde{\omega})\)).

(Notice that \(|E| \leq |\tilde{E}| < m + k^2\). In addition, as will be shown later, the distance
function \(d_3\) computed by Step 3 is correct, in the sense that \(d_3(v, s_i) = \delta(v, s_i)\),
for all \(v \in V\) and \(s_i \in H^A\).)

[Step 4]. Let \(\tilde{G} = (V, \tilde{E})\), where \(\tilde{E} = E \cup \{(u, v) : u \in V \text{ and } v \in H^A\}\) and let \(\tilde{\omega}\) be the weight function on \(\tilde{E}\) defined by

\[
\tilde{\omega}(u, v) = \begin{cases} 
  d_3(u, v) & \text{if } v \in H^A \\
  \omega(u, v) & \text{otherwise}
\end{cases}
\]

Apply the Hidden-Paths algorithm to the weighted graph \((\tilde{G}, \tilde{\omega})\) and return the
distance function \(d_4\) so computed.

As will be shown in Section 5.2, at the end of Step 4 we have \(d_4(u, v) = \delta(u, v)\),
for every \(u, v \in V\), i.e., Algorithm 2 is correct.

### 5.1 Complexity Analysis

We will evaluate the asymptotic behavior of Algorithm 2 under the assumption that all service priority queues are implemented with Fibonacci heaps.

To begin with, the \(k\) applications of Dijkstra’s algorithm in Step 1 take a
total time complexity of \(O(k(m + n \log n))\), where we recall that \(k\) is the
size of the hinge set \(H^A\). In addition, the execution of Floyd’s algorithm in
Step 2, with an input graph with \(k\) nodes, takes \(O(k^3)\)-time, whereas the \(k\) applications of Dijkstra’s algorithm in Step 3 take a total time complexity of
\(O(k(m + k^2 + n \log n))\), since \(\tilde{G}\) contains \(O(m + k^2)\) arcs. Finally, concerning
Step 4, we observe that its time complexity is dominated by the execution of the
Hidden-Paths algorithm on the weighted graph \((\tilde{G}, \tilde{\omega})\), which takes \(O(n\tilde{m}^* +
kn^2 \log n)\)-time, where \(\tilde{m}^*\) is the number of optimal arcs in \((\tilde{G}, \tilde{\omega})\). Since \(|\tilde{E}| \leq
|E| + kn\), the weighted graph \((\tilde{G}, \tilde{\omega})\) can contain at most \(kn\) optimal arcs more
than \((G, \omega)\), i.e., \(\tilde{m}^* \leq m^* + kn\). Thus the total time complexity for Step 4 is
\(O(n\tilde{m}^* + kn^2 \log n + kn^2)\).

In conclusion, Algorithm 2 has an overall \(O(nm^* + n^2 \log n + kn^2)\)-time complexity,
as \(k = O(kn^2)\), \(km = O(kn^2)\), and \(kn \log n = O(n^2 \log n)\).

Based on the results in [14, 19, 27], in [23] it is argued that \(m^*\) is likely to
be small in practice, since \(m^* = O(n \log n)\) with high probability for many
probability distributions on arc weights. Thus the assumptions that we will make
below on \(m^*\) are quite likely to be met in practice. Notice that when
\(m^* = O(n \log n)\), Algorithm 2 achieves a \(O(n^2 \log n + kn^2)\)-time complexity.
If, furthermore, \(k = O(\log n)\), then the complexity of our algorithm reduces to
\(O(n^2 \log n)\).

Next we compare the asymptotic behavior of Algorithm 2 with that of other
algorithms for the APSP problem present in the literature.
Algorithm 2 vs. Hidden-Path algorithm. To begin with, we observe at once that when
\[ k = \mathcal{O}\left(\frac{m^*}{n} + \log n\right), \tag{1} \]
Algorithm 2 achieves the same time complexity \( \mathcal{O}(nm^* + n^2 \log n) \) of the Hidden-Paths algorithm, which however solves the APSP problem only in the case of nonnegative weighted graphs. Indeed, if (1) holds, then we have \( \mathcal{O}(kn^2) = \mathcal{O}(nm^* + n^2 \log n) \), so that \( nm^* + n^2 \log n + kn^2 = \mathcal{O}(nm^* + n^2 \log n) \).

Algorithm 2 vs. Johnson’s algorithm. We recall that Johnson’s algorithm has a \( \mathcal{O}(mn + n^2 \log n) \)-time complexity. When \( k = \mathcal{O}(\frac{m^*}{n} + \log n) \), Algorithm 2 achieves the same asymptotic of Johnson’s algorithm. However, if \( m^* = o(m) \), \( m = \omega(n \log n) \), \( k = o\left(\frac{m}{n}\right) \), then Algorithm 2 is asymptotically faster than Johnson’s one, since in this case \( m^*n = \mathcal{O}(n^2 \log n) \), \( n^2 \log n = o(mn) \), and \( kn^2 = o(mn) \).

Algorithm 2 vs. Floyd’s algorithm. When \( k = o(n) \) and \( m^* = o(n^2) \), Algorithm 2 is asymptotically faster than Floyd’s algorithm.

Algorithm 2 vs. \( \mathcal{O}\left(\frac{n^3}{f(n)}\right) \)-time algorithms. The quest for truly subcubic algorithms for the APSP problem has produced so far several algorithms characterized by a \( \mathcal{O}\left(\frac{n^3}{f(n)}\right) \)-time complexity, with \( f(n) = o(\log^2 n) \); see [13, 35, 8, 16, 36, 37, 40, 3, 17, 4, 18]. Up to the time of the writing of this paper, the fastest algorithm for the APSP problem is due to Han and Takano [18], where \( f(n) = \log^2 n/\log \log n \), and it is believed that current techniques are not able to break the \((\log^2 n)\)-barrier for \( f(n) \). It is an easy matter to check that when \( m^* = o\left(\frac{n^2}{\log \log n}\right) \) and \( k = o\left(\frac{n^3}{\log^2 n}\right) \), our Algorithm 2 achieves a \( o\left(\frac{n^3}{f(n)}\right) \)-time complexity.

5.2 Correctness proof

Let, as above, \( G = (V, E) \) be a directed graph and let \( \omega : E \to \mathbb{R} \) be a weight function on \( G \). For the sake of simplicity, we will limit again our considerations to the case in which no negative weight cycle is present in \((G, \omega)\).

Our algorithm exploits the facts that when negative-weight arcs are allowed, the standard Dijkstra’s algorithm from a source node \( s \) calculate correctly the distances from \( s \) to all safe nodes relative to \( s \) (cf. Definition 3 and Theorem 5) and, likewise, the Hidden-Paths algorithm calculate correctly the distances from any source node \( s \) to all strongly safe nodes relative to \( s \), where the notion of strongly safe nodes (and paths) is defined as follows.
Definition 9 (Strongly safe nodes and paths) Let $G = (V, E)$ be a directed graph with a weight function $\omega : E \to \mathbb{R}$, and let $u, v \in V$. We say that a node $v$ is strongly safe relative to $u$ in $(G, \omega)$ if there exists a shortest path $\pi$ from $u$ to $v$, called a strongly safe path for $v$ relative to $u$, where only the first arc of $\pi$ is allowed to have a negative weight.

We will show below that strongly safe paths are handled correctly by the Hidden-Paths algorithm. But before doing that, it is convenient to strengthen the correctness proof of the Hidden-Paths algorithm (cf. [23, Theorem 2.2]) when run in presence of negative-weight arcs.

Lemma 10 Let $G = (V, E)$ be a directed graph with a weight function $\omega : E \to \mathbb{R}$, containing no negative-weight cycle. Then, during the execution of the Hidden-Paths algorithm on $(G, \omega)$, when a path $(u \leadsto v)$ whose endpoints $u$ and $v$ can be connected by a purely nonnegative shortest path is at the top of the heap, it is optimal.\footnote{We recall that a path is said to be purely nonnegative path if all its arcs have nonnegative weight.}

Proof. For the sake of brevity, in the following we say that a pair $(u, v)$ of distinct nodes of $G$ is good if there exists a purely nonnegative shortest path from $u$ to $v$ in $(G, \omega)$.

Let $Opt$ be the set of the paths extracted so far from the heap (these do not need to be necessarily optimal, because of the possible presence of negative-weight arcs in $(G, \omega)$; cf. Section 3.4). The lemma can be proved by induction on the size of $Opt$, by using as inductive hypothesis a modification of the one used in the proof of Theorem 2.2 in [23]. More precisely, our inductive hypothesis is that at the beginning of each iteration during the execution of the Hidden-Paths algorithm, when the path $(u \leadsto v)$ is at the top of the heap, the following properties hold:

1. if the pair $(u, v)$ is good, then $(u \leadsto v)$ is an optimal path;
2. $Opt$ contains an optimal path between each good pair of nodes of distance less than $\omega(u \leadsto v)$;
3. for each good pair of connected nodes $u$ and $v$ such that $(u \leadsto v) \notin Opt$, there exists a path of minimal weight from $u$ to $v$ of the form
   $$(v_0, v_1, \ldots, v_k \leadsto v_{k+1}),$$
   with $v_0 = u$ and $v_{k+1} = v$, such that
   • all the pairs $(v_i, v_{i+1})$, for $i = 0, 1, \ldots, k$, are good,
   • the path $(v_k \leadsto v_{k+1})$ is on the heap,
   • each of the arcs $(v_i, v_{i+1})$, for $i = 0, \ldots, k - 1$, is either on the heap or in $Opt$.

Plainly, the lemma follows from point 1 of the inductive hypothesis. \hfill \blacksquare
We are now ready to show that strongly safe paths are handled correctly by the Hidden-Paths algorithm.

**Lemma 11** Let \( G = (V, E) \) be a directed graph with a weight function \( \omega : E \to \mathbb{R} \). Then \( d(u, v) = \delta(u, v) \) holds at the end of the execution of the Hidden-Paths algorithm on \( (G, \omega) \), for every pair of distinct nodes \( u, v \in V \), such that \( v \) is strongly safe relative to \( u \) in \( (G, \omega) \).

**Proof.** Let \( u \leadsto v \) be a strongly safe shortest path for \( v \) relative to \( u \) in \( (G, \omega) \).

If \( u \leadsto v \) were a purely nonnegative path, then the thesis would follow by the preceding lemma.

On the other hand, if the path \( u \leadsto v \) contains some negative-weight arc, we distinguish two cases according to whether \( u \leadsto v \) has length 1 or more. If \( u \leadsto v \) has length 1, then it consists only of the negative-weight arc \( (u, v) \). In this case \( d(u, v) \) is set to \( \delta(u, v) \) already in the initialization phase, and its value can never change during the subsequent execution of the Hidden-Paths algorithm since \( (u, v) \) is an optimal edge. Otherwise, if the path \( u \leadsto v \) contains at least two arcs, then it has the form \( (u, w \leadsto v) \), where the arc \( (u, w) \) has negative weight and the subpath \( (w \leadsto v) \) is a purely nonnegative shortest path from \( w \) to \( v \). Thus, again by the preceding lemma, when a path from \( w \) to \( v \) is at the top of the heap, it must be optimal, i.e., \( d(w, v) = \delta(w, v) \) must hold. At that time the arc \( (u, w) \) must already have left the heap, as \( d(u, w) < 0 \leq d(w, v) \), and therefore procedure \( \text{UPDATE}(u, w, v) \) will be called. After such a call we have

\[
\delta(u, v) \leq d(u, v) \leq d(u, w) + d(w, v) = \delta(u, v),
\]

which implies that \( d(u, v) = \delta(u, v) \) holds, and since the value \( d(u, v) \) can not decrease subsequently, \( d(u, v) = \delta(u, v) \) will hold also at the end of the execution of the Hidden-Paths algorithm.

We are now ready to prove the correctness of Algorithm 2, by analyzing in turn its four steps. Thus, let \( G = (V, E) \) be a directed graph with a weight function \( \omega : E \to \mathbb{R} \) and let \( s_1, s_2, \ldots, s_k \) be the distinct elements of the hinge set \( H^A \) of \( (G, \omega) \).

By Theorem 5, after the execution of Step 1 \( d_1(s_i, v) = \delta(s_i, v) \) holds, for every \( s_i \in H^A \) and \( v \in V \) such that \( v \) is safe relative to \( s_i \) in \( (G, \omega) \). Thus, if \( s_{i_0} \leadsto s_{i_1} \leadsto \ldots \leadsto s_{i_t} \) is a shortest path from \( s_{i_0} \) to \( s_{i_t} \), with \( s_j \in H^A \) for \( j = 0, \ldots, t \), and such that each of the subpaths \( (s_{i_j} \leadsto s_{i_{j+1}}) \), for \( j = 0, \ldots, t - 1 \), does not contain any nodes in the hinge set \( H^A \) but its endpoints, then

\[
d_2(s_{i_0}, s_{i_t}) \geq \delta(s_{i_0}, s_{i_t}) = \sum_{j=0}^{t-1} \delta(s_{i_j}, s_{i_{j+1}}) = \sum_{j=0}^{t-1} d_1(s_{i_j}, s_{i_{j+1}}) \geq d_2(s_{i_0}, s_{i_t}),
\]

where \( d_2 \) is the distance function computed by the call to the Floyd’s algorithm on the graph \( (G, \omega) \) within Step 2, so that \( d_2(s_{i_0}, s_{i_t}) = \delta(s_{i_0}, s_{i_t}) \) holds.

Thus, we have proved the following result.
Lemma 12 After the execution of Step 2 of Algorithm 2, we have \( d_2(s_i, s_j) = \delta(s_i, s_j) \), for all \( i, j = 1, \ldots, k \). □

In Step 3 we have constructed the weighted graph \((\hat{G}, \hat{\omega})\), where \( \hat{G} \) is obtained by adding to our input graph \( G \) all arcs in \( \delta \), namely all arcs \((u, v)\) between distinct nodes \( u \) and \( v \) of the hinge set \( H^A \) such that \( v \) is reachable from \( u \) in \( G \).

By Lemma 12, all arcs in \( \delta \) are optimal with respect to the weight function \( \hat{\omega} \) or, more formally, \( \hat{\omega}(u, v) = \delta(u, v) \), for all distinct \( u, v \in H^A \).

Next, in the following lemma we prove that all shortest path distances in \((G, \omega)\) to nodes in the set \( H^A \) are correctly computed by Step 3.

Lemma 13 After the execution of Step 3 of our algorithm, we have \( d_3(v, s_i) = \delta(v, s_i) \), for every \( v \in V \) and \( s_i \in H^A \), where we recall that \( d_3 \) is the distance function computed by Step 3.

Proof. To begin with, we observe that, by Lemma 12, all arcs \((s_i, s_j)\) between distinct nodes \( s_i \) and \( s_j \) of the hinge set \( H^A \), such that \( s_j \) is reachable from \( s_i \) in \( G \), are optimal in the weighted graph \((\hat{G}, \hat{\omega})\) constructed in Step 3. Therefore, their reverse are also optimal in the reverse weighted graph \((\hat{G}^R, \hat{\omega}^R)\).

Let \((v \xrightarrow{} s_i)\) be a shortest path in \((G, \omega)\), with \( v \in V \) and \( s_i \in H^A \) and such that \( v \neq s_i \). Then, its reverse path \((s_i \xrightarrow{} v)\) is a shortest path in \((G^R, \omega^R)\). Let \( s_j \) be the last node in \((s_i \xrightarrow{} v)\) which belongs to \( H^A \). If \( s_j = v \), then \((s_i, v)\) is an optimal arc of \((\hat{G}^R, \hat{\omega}^R)\). Otherwise, if \( s_j \neq v \), there is an optimal path in \((\hat{G}^R, \hat{\omega}^R)\) of the form \((s_i, s_j \xrightarrow{} v)\), where the subpath \((s_j \xrightarrow{} v)\) contains no arc in \( H^A \) but its source. Thus, in any case it follows that the node \( v \) is (strongly) safe relative to \( s_i \) in \((\hat{G}^R, \hat{\omega}^R)\), so that, by Theorem 5, after the call to Dijkstra’s algorithm from \( s_i \) within Step 3, we have \( d'_3(s_i, v) = \delta^R(s_i, v) = \delta(v, s_i) \), where \( d'_3 \) is the shortest-path estimate function computed by the calls to Dijkstra’s algorithm within Step 3 and \( \delta^R \) is the distance function on \((G^R, \omega^R)\). But then \( d_3(v, s_i) = \delta(v, s_i) \), completing the proof of the lemma. □

Finally, we show that Step 4 propagates the correct distance estimate also to the remaining nodes of the graph.

Theorem 14 (Correctness) Let \( d_4 \) be the distance function computed by Algorithm 2 (namely, after the execution of the Hidden-Paths algorithm within Step 4). Then \( d_4(u, v) = \delta(u, v) \), for all \( u, v \in V \).

Proof. Let \((\hat{G}, \hat{\omega})\) be the weighted graph constructed in Step 4. Observe that, for each distinct nodes \( u \in V \) and \( v \in H^A \) such that \( v \) is reachable from \( u \) in \( G \), the graph \( \hat{G} \) contains the arc \((u, v)\), and these are optimal in \((\hat{G}, \hat{\omega})\) since \( \hat{\omega}(u, v) = d_3(u, v) \) (by initialization) and \( d_3(u, v) = \delta(u, v) \) (by Lemma 13).

Let \( u, v \in V \) be two connected nodes in our input graph \( G \) such that \( u \neq v \), and let \((u \xrightarrow{} v)\) be a shortest path from \( u \) to \( v \) in \((G, \omega)\).

If the path \((u \xrightarrow{} v)\) contains no negative-weight arc, then, by Lemma 10, after the execution of the Hidden-Paths algorithm within Step 4 we have \( d_4(u, v) = \delta(u, v) \).
On the other hand, if the path \((u \leadsto v)\) contains some negative-weight arc, we distinguish two cases according to whether \(v \in H^A\) or not. If \(v \in H^A\), then the graph \((\hat{G}, \hat{\omega})\) contains the optimal arc \((u, v)\), so that \(v\) is strongly safe relative to \(u\) in \((\hat{G}, \hat{\omega})\). Otherwise, the path \((u \leadsto v)\) must have the form \((u \leadsto s_i \leadsto v)\), for some \(s_i \in H^A\), where the subpath \((s_i \leadsto v)\) contains no node in \(H^A\) but its source. But then, again, \(v\) would be strongly safe relative to \(u\) in \((\hat{G}, \hat{\omega})\), because of the strongly safe path \((u, s_i \leadsto v)\) in \((\hat{G}, \hat{\omega})\). Thus, by Lemma 11, even in the case in which the path \((u \leadsto v)\) contains some negative-weight arc we can establish that \(d_4(u, v) = \delta(u, v)\), concluding the proof of the theorem.

6 Conclusions

We have presented two asymptotically fast algorithms for the single-source shortest paths and the all-pairs shortest paths problems, respectively, in the presence of few negative weight arcs.

Our algorithm for the SSSP problem is an improvement of an algorithm due to C.K. Yap for the shortest-path problem between two nodes in a directed graph with \(n\) nodes and \(m\) arcs in the presence of few negative weight arcs, whose running time is \(O(h(m + n \log n + h^2))\), where \(h\) is the minimum between \(n\) and the number of the negative weight arcs in the graph. If we denote by \(\ell_\sigma\) and \(\ell_\tau\) the number of nodes which are, respectively, sources and destinations of negative weight arcs, then our algorithm has a \(O(\ell(m + n \log n + \ell^2))\)-time complexity, where \(\ell = \min(\ell_\sigma, \ell_\tau)\). Thus, when \(\ell = o(h)\), our algorithm is asymptotically faster than Yap’s one. In addition, when \(\ell = o(\sqrt{mn})\), our algorithm turns out to be asymptotically faster than Yap’s one. In addition, when \(\ell = o(\sqrt{mn})\), our algorithm turns out to be asymptotically faster than the \(O(mn)\) Bellman-Ford-Moore algorithm.

In the case of the APSP problem, we have presented a \(O(nm^* + n^2 \log n + kn^2)\)-time algorithm, where \(m^*\) is the number of arcs participating in shortest paths and \(k\) is the number of nodes which are sources or destinations of negative weight arcs. When \(k = o(n)\) and \(m^* = o(n^2)\), our algorithm turns out to be asymptotically faster than Floyd’s algorithm. In addition, if \(k = O(\frac{n^2}{\log^2 n})\), our algorithm achieves the same \(O(nm^* + n^2 \log n))\)-time complexity of the Hidden-Paths algorithm which, however, works only with nonnegative-weighted graphs. We also compared our algorithm with Johnson’s algorithm and pointed out the cases in which they reach the same time complexity and when our algorithm outperforms that of Johnson. Finally, we showed that when \(m^* = o\left(\frac{n^2}{\log^2 n}\right)\) and \(k = o\left(\frac{n}{\log^2 n}\right)\), our algorithm achieves a \(o\left(\frac{n^3}{\log^2 n}\right)\)-time complexity, outperforming the best known subcubic algorithms for the APSP problem.

The above restrictions on \(m^*\) are very acceptable, as \(m^*\) is likely to be \(O(n \log n)\) in practice (cf. [23]).

References


