A Potential-Based Variant of the Bellman-Ford Algorithm*†

Domenico Cantone[0000−0002−1306−1166], Pietro Maugeri, and Stefano Scafiti

Department of Mathematics and Computer Science, University of Catania
Viale A. Doria 6, 95125, Catania, Italy
{domenico.cantone,pietro.maugeri}@unict.it,
stefano.scafiti@studium.unict.it

Abstract. We present an improvement of the Bellman-Ford algorithm for the single-source shortest-path (SSSP) problem in directed graphs. Our algorithm exploits a potential-based heuristic for selecting the nodes to be scanned next. More specifically, for each node, we use as potential the difference between the current distance estimate and the distance estimate at the time the node was last scanned. A nice feature of our proposed heuristic is that it precisely yields Dijkstra's algorithm when applied to graphs with no negative edges. Extensive experimentations have been conducted for comparing our algorithm on various families of weighted graphs against the most efficient practical algorithms for the SSSP problem with negative weights, such as Pallottino's two-queue algorithm, D'Esopo-Pape deque algorithm, the threshold algorithm by Glover *et al.*, and Goldberg-Radzik's algorithm. Results are very promising and show that our algorithm is competitive with the above-mentioned algorithms, especially when graphs are not *nearly acyclic*.

Keywords: Single-source shortest path problem, Bellman-Ford algorithm, Potential-based heuristic.

1 Introduction

The *single-source shortest-path (SSSP, for short) problem* consists in finding the shortest paths from a given source to any other node in a weighted directed graph (or *digraph*).

When a digraph $G = (V, E)$ is subject to no particular restriction, the Bellman-Ford algorithm [Bel58,For56] solves the problem in $\mathcal{O}(VE)$ time. For graphs with no negative weight edges, an asymptotically better solution is provided by Dijkstra's algorithm [Dij59], which achieves a $\mathcal{O}(E + V \log V)$ time worst-case complexity when the service priority queue is implemented with Fibonacci min-heaps [FT87]. Asymptotically faster algorithms than the Bellman-Ford and Dijkstra's algorithms can be implemented depending on the topology

* We gratefully acknowledge support from "Università degli Studi di Catania, Piano della Ricerca 2016/2018 Linea di intervento 2".

of the input graph. For instance, if the graph is acyclic, by simply scanning the nodes according to any topological order of the underlying graph, it is possible to achieve a $\mathcal{O}(V + E)$-time complexity. Also, on *nearly acyclic* graphs, the Saunders-Takaoka algorithm [ST01] retains a complexity of $\mathcal{O}(|E| + r \log r)$, where r is the number of *trigger nodes*,[1] which is usually small in nearly acyclic graphs. When negative edges are not allowed to belong to cycles, the Two-Levels-Greedy algorithm [CF04] has the same time complexity as Dijkstra's algorithm, and shows a linear behaviour in practice. Finally, for graphs with 'few' destinations of negative weight edges, the Cantone-Faro hybrid algorithm [CF14] is asymptotically faster than the Bellman-Ford algorithm.

Although no algorithm with an $o(VE)$ worst-case time complexity for the general SSSP problem is currently known, several heuristic improvements have been proposed over the years, which outperform the Bellman-Ford algorithm in practice. Among them, we mention the Goldberg-Radzik's algorithm [GR93] that retains a worst-case complexity of $\mathcal{O}(VE)$, whereas the *two-queue* algorithm of Pallottino [Pal84] and the threshold algorithm by Glover *et al.* [GGK86] take $\Omega(V^2E)$ time in the worst case. In some practical cases, the D'Esopo-Pape algorithm [Pap74] can be much faster of the above-mentioned algorithms, though there are graphs on which it takes exponential time.

In this paper, we present a new heuristic, and also a variant to that heuristic, that yields an algorithm whose performance is competitive with the aforementioned algorithms, and which additionally yields Dijkstra's algorithm when negative-weight edges are not present in the input graph.

The paper is organized as follows. In Section 2, we recall the notation used in the paper, and also give a brief overview of the most common heuristic SSSP algorithms. Then, in Section 3, we present our heuristic and its related algorithm, and prove some of its properties. Next, in Section 4, we compare our proposed algorithm with the most efficient practical algorithms for the SSSP problem, also commenting on the results obtained. Finally we close the paper with some final remarks in Section 5.

2 Preliminaries

The *single-source shortest-path problem* is the problem of finding the shortest paths from a designated source node $s \in V$ to all the nodes of a given weighted directed graph (G, w), where $G = (V, E)$, with V a finite set of *nodes* and $E \subseteq V \times V$ a set of *edges*, and where $w \colon E \to \mathbb{R}$ is a real-valued *weight function*. We shall assume that E contains no *self-loops*, i.e., edges of the form (v, v).

A *path* in $G = (V, E)$ (from node v_1 to v_k) is any finite sequence of nodes $(v_1, \ldots, v_k)$ such that $(v_i, v_{i+1}) \in E$, for $i = 1, \ldots, k-1$; when there is a path from v_1 to v_k, we say that the node v_k is *reachable* from v_1 in G. A path $(v_1, \ldots, v_k)$ such that $v_1 = v_k$ is a *cycle*. The weight function can naturally be extended over

[1] Trigger nodes are the roots of the trees that result when the graph is decomposed into trees.

paths by putting $w(v_1, \ldots, v_k) \coloneqq \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. A *shortest path* from u to v is any path from u to v whose weight is minimum among all paths from u to v in G. Given two nodes $u, v \in V$, provided that v is reachable from u and that there is no path from u to v containing a negative-weight cycle, the shortest path from u to v always exists. In such a case, we denote its weight by $\delta_{G,w}(u, v)$. If a node v is not reachable from u in G, we put $\delta_{G,w}(u, v) \coloneqq +\infty$. In addition, if there is a path from u to v containing a negative-weight cycle, we put $\delta_{G,w}(u, v) \coloneqq -\infty$. The function $\delta_{G,w} \colon V \times V \to \mathbb{R} \cup \{+\infty, -\infty\}$ is the *distance function* of (G, w).

When the weighted graph (G, w) is understood, we simply write δ in place of $\delta_{G,w}$. In addition, for any given source node $s \in V$, we shall write δ_s for the unary distance function defined by $\delta_s(u) \coloneqq \delta(s, u)$, for $u \in V$.

A common technique for solving the SSSP problem is the so-called *label-correcting* method, which makes use of the following functions:

- a *distance estimate* function $d \colon V \to \mathbb{R} \cup \{+\infty\}$,
- a *predecessor* function $\pi \colon V \to V \cup \{\text{NIL}\}$, and
- a *status* function $S \colon V \to \{\text{UNREACHED}, \text{LABELED}, \text{SCANNED}\}$.

On a graph (G, w) with source $s \in V$, a generic algorithm based on the label-correcting method works as follows (see the GENERIC-SSSP Algorithm 1). At start, the functions d, π, and S are so initialized, by the procedure INITIALIZE-SINGLE-SOURCE: $d(v) \coloneqq +\infty$, $\pi(v) \coloneqq \text{NIL}$, and $S(v) \coloneqq \text{UNREACHED}$, for every node $v \in V \setminus \{s\}$, and $d(s) \coloneqq 0$, $\pi(s) \coloneqq \text{NIL}$ and $S(s) \coloneqq \text{LABELED}$, for the source node s. Then the SCAN procedure is run on any LABELED node until none is left, namely until all nodes are either SCANNED or UNREACHED. If no negative-weight cycle is reachable from the source node s, the halting condition is eventually attained: in this case, it turns out that $d = \delta_s$ holds and, for each node $v \in V$,

- $\pi(v)$ is the node preceding v in a shortest path from s to v, if $v \neq s$ and v is reachable from s,
- $\pi(v) = \text{NIL}$, otherwise,

so that the predecessor function allows one to reconstruct a shortest path to each node reachable from the source.

Otherwise, if some negative-weight cycle is reachable from s, the halting condition is never attained and execution never stops, since a SCANNED node becomes LABELED when its distance estimates strictly decreases.

2.1 Label-correcting algorithms

Depending on the strategy used to select the node to be scanned next, different algorithms can be obtained from the GENERIC-SSSP algorithm.

The Bellman-Ford algorithm [Bel58,For56] maintains a FIFO queue of LABELED nodes. A node that becomes LABELED is added to the tail of the queue, while the node to be scanned next is extracted from the queue's head.

In the D'Esopo-Pape's algorithm [Pap74], labeled nodes are maintained in a *deque* (i.e., a queue that allows insertions at either ends). When a node becomes

Algorithm 1 GENERIC-SSSP

```
 1: procedure INITIALIZE-SINGLE-SOURCE(V, s);
 2:     for all v ∈ V do
 3:         d(v) := +∞;
 4:         π(v) := NIL;
 5:         S(v) := UNREACHED;
 6:     d(s) := 0;
 7:     S(s) := LABELED;

 8: procedure SCAN(u, E, w)
 9:     for all (u, v) ∈ E do
10:         if d(u) + w(u, v) < d(v) then
11:             d(v) := d(u) + w(u, v);
12:             S(v) := LABELED;
13:             π(v) := u;
14:     S(u) := SCANNED;

15:  GENERIC-SSSP ALGORITHM(G, w, s)
16:     - let V and E be the set of nodes and the set of edges of G, respectively;
17:     INITIALIZE-SINGLE-SOURCE(V, s);
18:     while there is some LABELED node in V do
19:         - let u ∈ V be any LABELED node;
20:         SCAN(u, E, w);
21:     return d, π;
```

LABELED, if it is LABELED for the first time, it is added at the deque's tail, otherwise it is added at the deque's head. As in the Bellman-Ford algorithm, at each iteration the node to be scanned next is extracted from the deque's head.

Pallottino's algorithm [Pal84] uses two FIFO queues to store the set of labeled nodes, a *high-priority* and a *low-priority* queue. When a node becomes LABELED, if it is LABELED for the first time, it is added to the low-priority queue, otherwise it is added to the high-priority queue. The node to be scanned next is extracted from the head of the high-priority queue, if non-empty, otherwise is extracted from the low-priority queue.

The threshold algorithm by Glover *et al.* [GGK86] arranges the set of labeled nodes into two FIFO queues, respectively NEXT and NOW. A threshold parameter t, which is equal to a weighted average of the minimum and the average distance estimates of the nodes in NEXT, is also maintained. At the beginning of each iteration, every node $v \in$ NEXT such that $d(v) \leq t$ is moved to the queue NOW, which is initially empty at each pass. Nodes belonging to the queue NOW are scanned, until the queue NEXT becomes empty at the end of an iteration.

The Goldberg-Radzik's algorithm [GR93] maintains the LABELED nodes of a given weighted digraph (G, w), with $G = (V, E)$, into two sets, A and B. At any instant, each node can only belong to one of such sets. When a node becomes LABELED, it is added to B. To compute the set A, a *reduced cost function* $w_d(u, v) = w(u, v) + d(u) - d(v)$ is defined on edges, together with the corresponding *admissible graph* $G_d = (V, \{e \in E : w_d(e) < 0\})$. At the beginning of each iteration, nodes with no outgoing edges and negative reduced cost are removed from B, and A is the set of nodes reachable from B in G_d. The node to be scanned next is extracted from A, following a topological order of G_d.

Finally, Dijkstra's algorithm [Dij59] maintains the LABELED nodes in a min-queue, and selects the node with lowest priority as the node to be scanned next.

In the next section, we present a new heuristic, based on a suitable potential function, for selecting the node to be scanned next by the GENERIC-SSSP algorithm. The resulting procedure, to be called *Potential algorithm*, turns out to be very fast in practice, and in most cases very competitive with the aforementioned heuristic algorithms for the SSSP problem. In particular, when run on digraphs with no negative edges, the Potential algorithm reduces just to Dijkstra's algorithm.

3 The Potential algorithm

The potential heuristic. Central to our heuristic is the following notion of *potential function* over the set of nodes of a weighted digraph. Consider any execution of the GENERIC-SSSP algorithm on a weighted digraph (G, w), with $G = (V, E)$, from a designated source $s \in V$. At any computation step (namely just prior to the execution of any SCAN operation), the potential $\mathsf{U}(u)$ of any node $u \in V$ that has already been scanned is the difference between its current distance estimate and its distance estimate when u was last scanned. For each of the remaining nodes $u \in V$ that have not yet been scanned, letting $d\colon V \to \mathbb{R}$ be the current distance estimate function, we found it more convenient to define the current potential by setting $\mathsf{U}(u) \coloneqq d(u)$, when the node u has already been discovered (namely if $d(u) \neq +\infty$), and $\mathsf{U}(u) \coloneqq 0$, otherwise.

The intuition behind our heuristic is that the *smaller* is the potential value of a node u, the *closer* is the distance estimate of u to the actual distance of u from the source node, and therefore the *higher* is the chance that the call SCAN(u) may result in conspicuous improvements to the distance estimates of the nodes adjacent to u. As a consequence, the distance estimates function is expected to converge *faster* to the distance function from the source node, especially in the case of graphs whose edge weights are uniformly randomly distributed. As will be discussed in Section 4, there is indeed experimental evidence that, for LABELED nodes $u, v \in V$ such that $\mathsf{U}(u) < \mathsf{U}(v)$, scanning u before v leads in general to a faster convergence of the distance estimate function to the distance function δ_s from a given source node s.

The above remarks lead naturally to our *potential heuristic*, consisting in selecting at each iteration of the GENERIC-SSSP algorithm a LABELED node with the lowest potential as the node to be scanned next.

The Potential algorithm. The Potential algorithm (see Algorithm 2) makes use of the variants P-INITIALIZE-SINGLE-SOURCE and P-SCAN of INITIALIZE-SINGLE-SOURCE and SCAN, respectively, specifically adapted to handle the potential function.

The potential function U is initialized to the null function at line 4. After initialization, the execution of the Potential algorithm progresses through a sequence of iterations of the inner **while-loop** at lines 29–31 of Algorithm 2. Every iteration, but the first one, starts with a min-priority queue Q formed by the nodes with a negative potential and having as priority the potential function.

Instead, the first iteration starts with the queue Q initialized to the singleton of the source node.

For simplicity, in what follows we shall tacitly assume that no cycle with negative weight is reachable from the designated source node of the graphs under consideration. Thus the number of iterations of the inner **while-loop** at lines 29–31 will never exceed $|V| - 1$.

Each iteration consists of a sequence of SCAN operations on the nodes extracted from the queue Q, in accordance with their potential, until no node is left in Q, at which point the queue Q is restored for the next iteration with the nodes, if any, having a negative potential. The algorithm stops when, at the end of an iteration, no node has a negative potential.

During the execution of any iteration, as soon as a node v is marked LABELED by the P-SCAN procedure (at line 18), if the node v has not entered yet the queue Q in the current iteration, then it is readily inserted into Q, in accordance with its potential. The set Q_0 in procedure P-SCAN keeps track of the nodes that have entered the queue in the current iteration.

In addition, the procedure P-SCAN maintains the potential function. Letting d_A and d_B be the distance estimate functions just after and before a P-SCAN operation on a node u, respectively, we plainly have $d_A(v) - d_B(v) \leq 0$, for every node $v \in V$ such that $d_B(v) \neq +\infty$. Denoting by U_A and U_B the potential functions on V just after and before the execution of the given P-SCAN operation on node u, respectively, we have $\mathsf{U}_A(u) = 0$ and, for $v \in V \setminus \{u\}$,

$$\mathsf{U}_A(v) = \begin{cases} \mathsf{U}_B(v) + d_A(v) - d_B(v) & \text{if } d_B(v) \neq +\infty \text{ and } d_A(v) - d_B(v) < 0 \\ & \text{(line 14 in Algorithm 2)} \\ d_A(v) & \text{if } d_B(v) = +\infty \text{ and } d_A(v) \neq +\infty \\ & \text{(line 16 in Algorithm 2)} \\ \mathsf{U}_B(v) & \text{otherwise.} \end{cases}$$

The following lemmas, whose proofs are omitted for lack of space, state that the Potential algorithm is correct and that it behaves just as Dijkstra's algorithm on digraphs with non-negative weights.

Lemma 1. *The Potential algorithm is correct.*

Lemma 2. *On a digraph $G = (V, E)$ with a non-negative weight function $w\colon E \to \mathbb{R}_0^+$, the Potential algorithm stops at the end of the first iteration of its inner **while-loop** at lines 29–31, behaving just like Dijkstra's algorithm.*

Complexity issues. As remarked, the inner **while-loop** at lines 29–31 can be iterated at most $|V| - 1$ times (when no negative weight cycle is reachable from the source node). In addition, each such iteration involves at most $\mathcal{O}(V)$ INSERT and EXTRACT-MIN operations and $\mathcal{O}(E)$ DECREASE-KEY operations (as a consequence of the updates made by the instruction at line 17). Hence, if the service min-priority queue is implemented as a Fibonacci heap [FT87], each iteration takes $\mathcal{O}(E + V \log V)$ time, for an overall $\mathcal{O}(VE + V^2 \log V)$-time complexity in the worst-case. Summing up, we have:

Lemma 3. *The Potential algorithm runs in $\mathcal{O}(VE + V^2 \log V)$ worst-case time.*

Algorithm 2 Potential algorithm

```
 1: procedure P-Initialize-Single-Source(V, s);
 2:     for all v ∈ V do
 3:         d(v) := +∞;
 4:         U(v) := 0;
 5:         π(v) := nil;
 6:         S(v) := unreached;
 7:     d(s) := 0;
 8:     S(s) := labeled;
 9: procedure P-Scan(u, E, w, Q, Q_0);
10:     for all (u, v) ∈ E do
11:         if d(u) + w(u, v) < d(v) then
12:             d'(v) := d(u) + w(u, v);
13:             if d(v) ≠ +∞ then
14:                 U(v) := U(v) + d'(v) − d(v);
15:             else
16:                 U(v) := d'(v);
17:             d(v) := d'(v);
18:             S(v) := labeled;
19:             π(v) := u;
20:             if v ∉ Q_0 then
21:                 Insert(Q, v);
22:                 Q_0 := Q_0 ∪ {v};
23:     U(u) := 0;
24:     S(u) := scanned;
25: Potential Algorithm(G, w, s)
26:     P-Initialize-Single-Source(V, s);
27:     Q_0 := Q := {s};
28:     while Q ≠ ∅ do
29:         while Q ≠ ∅ do
30:             u := Extract-Min(Q, U);
31:             P-Scan(u, E, w, Q, Q_0);
32:         Q_0 := Q := {u ∈ V | U(u) < 0};
33:     return d, π;
```

3.1 A variant of the Potential algorithm

We have seen that, when the weight function is non-negative, the best node to be scanned next is the one with the minimum potential, as this corresponds to the node with the minimum distance estimate (as in Dijkstra's algorithm). On the other hand, when negative edge costs are allowed, this approach leads to a worst-case time bound worse than the Bellman-Ford algorithm's one, since the Potential algorithm maintains a priority queue. By relaxing our heuristic as explained next, we can obtain a $\mathcal{O}(VE)$ algorithm in the worst-case. However, on graphs with no negative weight edges, the resulting algorithm will no longer behave as Dijkstra's algorithm.

At each step, the node to be scanned next is not necessarily the one with the minimum value of the cumulative potential function, rather it is extracted from a *deque* Q according to the following strategy. When a node u must be added to Q, its potential is preliminarily compared with that of the first node of the queue, $\textsc{head}(Q)$: if $\mathsf{U}(u) < \mathsf{U}(\textsc{head}(Q))$, the node u is added to the front of the queue, otherwise it is added to the back. The same strategy is used during the initialization of the set Q, at the end of each iteration. The pseudocode of such a variant of the Potential algorithm is reported in Algorithm 3. As we shall

see in the next section, this variant achieves the best performances with random graphs, when negative edge costs are allowed.

Algorithm 3 Relaxed Potential algorithm (deque variant)

```
 1: procedure P-Scan-Deque(u)
 2:     for all (u, v) ∈ E do
 3:         if d(u) + w(u, v) < d(v) then
 4:             d'(v) := d(u) + w(u, v);
 5:             if d(v) ≠ +∞ then
 6:                 U(v) := U(v) + d'(v) − d(v);
 7:             else
 8:                 U(v) := d'(v);
 9:             d(v) := d'(v);
10:             S(v) := LABELED;
11:             π(v) := u;
12:             if v ∉ Q and entered-Q(v) = false then
13:                 if Q = ∅ or U(v) < U(HEAD(Q)) then
14:                     PUSH-FRONT(Q, v)
15:                 else
16:                     PUSH-BACK(Q, v)
17:                 entered-Q(v) := true;
18:     U(u) := 0;
19:     S(u) := SCANNED;
20: procedure Initialize-Deque(Q)
21:     Q := ∅;
22:     for v ∈ V do
23:         if U(v) < 0 then
24:             if Q = ∅ or U(v) < U(HEAD(Q)) then
25:                 PUSH-FRONT(Q, v);
26:             else
27:                 PUSH-BACK(Q, v);
28:             entered-Q(v) := true
29:         else
30:             entered-Q(v) := false
31:  Potential1 Algorithm(G, w, s)
32:     P-Initialize-Single-Source(V, s)
33:     Q ← {s};
34:     while Q ≠ ∅ do
35:         while Q ≠ ∅ do
36:             v ← Pop-Front(Q);
37:             P-Scan-Deque(v);
38:         Initialize-Deque(Q);
39:     return d, π;
```

4 Experimental Results

In this section we compare the experimental behaviour of our proposed algorithms with some of the well-known SSSP algorithms. We shall refer to each algorithm with the following acronyms:

- "BFP" is the parent-checking variant of the Bellman-Ford algorithm,
- "PAPE" is the Pape-Levi algorithm,
- "TWO_Q" is the Pallottino's two queues algorithm,
- "THRESH" is the threshold algorithm by Glover *et al.*,

- "GOR" and "GOR1" are the Goldberg-Radzik's algorithm and its variant with distance updates, respectively,
- "DIKH" is the Dijkstra algorithm implemented using a k-ary heap,
- "POT" and "POT1" are the Potential algorithm and its variant described in Section 3.1, respectively.

A very extensive investigation of the practical performances of the various SSSP algorithms has been carried out over the years, pointing out how graph properties may (strongly) affect performances. We therefore performed our tests on the families that turned out to be the most complex ones in each of the standard graph classes used in such analyses (see [CGR96] for more details). More specifically, for our experiments, we considered the following graph families:

- *SPGRID*:
  Grid-NHard (hard problems with mixed edge length),
- *SPRAND*:
  Rand-1:4 (random Hamiltonian graphs with density 4),
  Rand-P (random Hamiltonian graphs with potential transformation),
- *SPACYC*:
  Acyc-Neq (acyclic graphs with negative edge lengths),
  Acyc-P2N (acyclic graphs with variable fraction of negative edges).

All tests have been performed on a PC with a 3.40 GHz Intel Quad Core i5-4670 processor, with 6144 KB cache memory, and running Linux Ubuntu 19.04. Running times have been measured with a hardware cycle counter, available on modern CPUs and have been reported in milliseconds. Each table entry has been obtained as the average over five runs, on problems produced with the same generator parameters except for the pseudorandom generator seed.

4.1 Results for SPGRID class

In SPGRID graphs, nodes correspond to points on the plane with integer coordinates $[x, y]$, where $1 \leq x \leq X$ and $0 \leq y \leq Y$, for given $X, Y \geq 1$. Each node is connected

- *forward*, by edges of the form $([x, y], [x + 1, y]), 1 \leq x < X,\ 0 \leq y \leq Y$,
- *up*, by edges of the form $([x, y], [x, (y + 1 \bmod Y)]),\ 1 \leq x \leq X,\ 0 \leq y < Y$,
- *down*, by edges of the form $([x, y], [x, (y-1 \bmod Y)]),\ 1 \leq x \leq X,\ 0 \leq y < Y$.

For any fixed value $x \in \{1, \ldots, X\}$, the nodes $[x, y]$ form a doubly connected cycle called a *layer*. Finally, the source node is connected to all the nodes in the first layer.

In particular, in our experiments we considered the family *Grid-NHard* of SPGRID graphs. In *Grid-NHard* graphs, the weight of the edges inside a layer is small and non-negative. In addition to the edges that connect one layer with the next one, there are also edges that connect lower to higher numbered layers, all the inter-layer edges having a non-positive weight. Weights were generated uniformly at random. A significant property of *Grid-NHard* graphs is that a path between two nodes with many edges is more likely to have smaller weight than

nodes/edges	POT	POT1	BFP	TWO_Q	THRESH	PAPE	GOR	GOR1
8193/63808	22.39	43.36	108.36	305.02	248.41	2995.80	4.74	**3.03**
16385/129344	62.69	164.48	446.17	645.69	1064.58	5208.70	10.42	**6.56**
32769/260416	181.10	666.98	1870.08	1348.39	4739.12	9889.34	23.26	**14.68**
65537/522560	480.93	3031.38	9567.19	2710.32	24015.21	18302.97	53.90	**35.90**
131073/1046848	1306.93	14495.88	48213.83	5456.46	121938.01	36521.89	120.77	**78.19**

Table 1. Experimental execution time in milliseconds for some *Grid-NHard* families

a path with fewer edges. This makes it difficult to formulate an optimal choice on a local information basis only.

Table 1 shows the results of an experimental session for the *Grid-NHard* family (Dijkstra's algorithm was not tested for this family since negative weight edges were present). The Goldberg-Radzik's algorithms are the most efficient ones for this family. Interestingly, they show a linear behaviour even when the *Grid-NHard* graphs are not acyclic, while all the remaining algorithms show a quadratic behaviour. However, among the remaining algorithms, our Potential algorithm turns out to be the fastest.

4.2 Results for SPRAND class

SPRAND graphs are constructed by creating a Hamiltonian cycle and then adding edges with distinct random end points. In our experiments, we set the weight of the edges in the cycle to 1. Two families of graphs were chosen from this class for our experiments, *Rand-Len* and *Rand-P*.

In *Rand-Len* graphs, the weight of the edges outside the cycles is chosen at random in an interval $[0, U]$. During the first test, all the edges had a weight of 1. Then, at each test, the value of U was incremented.

In the *Rand-P* family, a *potential* value p in the interval $[0, P]$, is assigned to each node, and then the weight function w is computed in the same way as for *Rand-Len* graphs. Finally, a reduced weight function w_p is computed by setting $w_p(u, v) \coloneqq w(u, v) + p(u) - p(v)$, and w_p is used as weight function in the test. Notice that while w is non-negative, w_p may take negative values.

The *Rand-Len* family is the only family of graphs with non-negative weights. Hence, it was possible to test on it also Dijkstra's algorithm. Since *Rand-Len* graphs are not acyclic, the runtimes of the Goldberg-Radzik's algorithms are comparable with those of the remaining algorithms. With this family, our Potential algorithm and its variant are very fast and, as shown in Table 2, the Potential algorithm's timings are very close to those of Dijkstra's ones, giving evidence in support of the property proved in Lemma 2.

The family *Rand-P* is very similar to the family *Rand-Len*, with the main difference that *Rand-P* graphs admit negative weight edges. The Potential algorithm and its variant are faster than the remaining algorithms, as reported in Table 3.

$[L, U]$	POT	POT1	BFP	GOR	GOR1	TWO_Q	THRESH	PAPE	DIKH
$[1, 1]$	23.96	11.71	10.90	17.39	58.95	10.66	18.95	**10.64**	21.01
$[0, 10]$	28.47	25.17	34.91	40.97	73.50	27.63	**20.08**	28.04	25.14
$[0, 100]$	29.69	36.15	76.21	73.38	88.65	64.24	35.47	63.96	**25.47**
$[0, 10000]$	31.64	55.71	171.49	123.08	119.46	169.27	120.49	160.25	**26.94**
$[0, 1000000]$	30.31	72.80	220.93	159.42	69.69	250.48	177.29	244.91	**26.85**

Table 2. Experimental execution time in milliseconds for some *Rand-Len* family; all graphs have 131072 nodes and 524288 edges.

P	POT	POT1	BFP	GOR	GOR1	TWO_Q	THRESH	PAPE
0	**30.67**	59.12	159.77	129.74	125.35	165.17	118.81	160.6
1000	**51.20**	69.47	155.21	121.16	121.34	165.12	114.97	160.21
5000	90.27	**77.82**	154.25	123.56	122.00	163.33	118.83	161.49
10000	117.24	**83.02**	156.26	125.98	123.14	162.39	128.64	163.15
100000	183.29	**97.82**	161.40	130.55	119.72	164.70	326.36	163.79
1000000	218.53	**100.03**	164.02	135.03	121.63	162.92	412.45	164.66

Table 3. Experimental execution time in milliseconds for some *Rand-P* families; all graphs have 131072 nodes and 524288 edges.

4.3 Results for SPACYC class

The graphs in the SPACYC class are generated by first numbering all the nodes from 1 to n and then building a path by adding the *path edges* $(i, i + 1)$, for $1 \leq i < n$. Additional edges are added subsequently by picking pairs of nodes at random and connecting the lower node to the higher numbered one and assigning to them weights chosen uniformly at random in an interval $[L, U]$. Specifically, for our tests we used the *Acyc-Neg* and *Acyc-P2N* families of graphs.

For *Acyc-Neg* graphs, the weight of path edges are all set to -1, while L is set to -10000 and U to 0. Note that all weights are non-positive and chosen uniformly at random in the interval $[L, U]$. Such configuration may show up in practical situations such as in the problem of finding the longest path in acyclic graphs, where a common solution is to change the sign of all the weights and then find the shortest path in the resulting graph. For the *Acyc-P2N* family, the number of edges and nodes was fixed, while the values L and U changed at each test. The fraction f of negative edges is then computed from L and U.

The graphs in the SPACYC class are all acyclic. The Goldberg-Radzik's algorithms turned out to be the fastest, since, thanks to their use of topological sorting, their running time is in practice linear on acyclic graphs. We mention that our Potential algorithm was significantly faster than the remaining algorithms, as shown in Tables 4 and 5.

Interestingly enough, in the case of *Acyc-P2N* graphs with a low percentage of negative weight edges all algorithms showed a very similar behavior.

nodes/edges	POT	POT1	BFP	TWO_Q	THRESH	PAPE	GOR	GOR1
8193/63808	22.90	25.46	127.86	845.57	404.42	1017.32	**0.69**	0.83
16385/129344	60.530	76.75	522.25	3936.48	1737.12	4610.27	**1.79**	2.04
32769/260416	155.92	260.46	2589.59	17815.66	7967.76	28055.02	**4.43**	4.92
65537/522560	118.81	924.27	13295.41	93352.41	39584.39	109143.61	**10.19**	11.23
131073/1046848	1177.61	3293.57	66871.14	375492.12	218589.70	/	**26.07**	28.11

Table 4. Experimental execution time in milliseconds for some *Acyc-Neg* families

$f(\%)$	POT	POT1	BFP	TWO_Q	THRESH	PAPE	GOR	GOR1
0	3.20	2.04	2.03	**2.07**	2.13	2.05	3.30	2.28
10	3.49	2.45	2.57	2.59	**2.31**	2.54	3.32	2.37
20	6.60	4.81	6.06	6.90	6.88	7.00	5.71	**2.55**
30	12.68	13.94	21.08	30.79	30.74	30.90	9.84	**2.53**
40	27.47	42.04	114.18	282.77	225.11	337.11	16.36	**2.23**
50	60.79	90.47	662.63	3777.12	1779.03	4124.36	23.82	**2.01**
60	110.27	154.10	1250.98	7504.86	3680.62	7302.66	25.49	**1.97**
100	131.11	187.48	1403.05	3809.14	4502.53	7655.65	**1.79**	1.98

Table 5. Experimental execution time in milliseconds for some *Acyc-P2N* families; all graphs have 16384 nodes and 262144 edges.

5 Conclusions

We have presented a new heuristic for the Bellman-Ford algorithm for the SSSP path problem, based on a potential heuristic for selecting the nodes to be scanned next. More specifically, for each node, we use as potential the difference between the current distance estimate and the distance estimate at the time the node was last scanned. It turns out that, on graphs with no negative weight edges, our potential heuristic yields Dijkstra's algorithm. Though in the worst case the resulting Potential algorithm has an $\mathcal{O}(VE + V^2 \log V)$-time complexity, in practice it is very competitive with the most efficient algorithms for the SSSP problem.

We also proposed an $\mathcal{O}(VE)$-time variant of the Potential algorithm that achieves the best performances on random graphs where negative weights are allowed.

In view of the very promising practical performances exhibited by our Potential algorithm and its variant, we shall attempt to adapt the potential heuristic also to the all-pairs shortest-path problem. We also intend to investigate the topological properties of input graphs that suit well to our potential heuristic. Finally, in preparation for a more comprehensive analysis, we plan to analyze probabilistically the behaviour of our potential heuristic in the case of graphs with very few negative arcs.

Acknowledgements

The authors wish to thank the anonymous reviewers for some helpful comments.

References

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[CF04] Domenico Cantone and Simone Faro. Two-levels-greedy: a generalization of Dijkstra's shortest path algorithm. *Electronic Notes in Discrete Mathematics*, 17:81 – 86, 2004. Workshop on Graphs and Combinatorial Optimization.

[CF14] Domenico Cantone and Simone Faro. Fast shortest-paths algorithms in the presence of few destinations of negative-weight arcs. *Journal of Discrete Algorithms*, 24:12 – 25, 2014.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical Programming*, 73(2):129–174, 1996.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[For56] Lester R. Ford Jr. Network flow theory. Technical report, RAND Corporation, Santa Monica, California, 1956. Paper P-923.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[GGK86] Fred Glover, Randy Glover, and Darwin Klingman. Threshold assignment algorithm. In Giorgio Gallo and Claudio Sandi, editors, *Netflow at Pisa, Mathematical Programming Studies*, volume 26, pages 12–37. Springer Berlin Heidelberg, 1986.

[GR93] Andrew Goldberg and Tomasz Radzik. A heuristic improvement of the Bellman-Ford algorithm. *Applied Mathematics Letters*, 6(3):3–6, 1993.

[Pal84] Stefano Pallottino. Shortest-path methods: Complexity, interrelations and new propositions. *Networks*, 14(2):257–267, 1984.

[Pap74] U. Pape. Implementation and efficiency of Moore-algorithms for the shortest route problem. *Mathematical Programming*, 7(1):212–222, 1974.

[ST01] Shane Saunders and Tadao Takaoka. Improved shortest path algorithms for nearly acyclic graphs. *Electronic Notes in Theoretical Computer Science*, 42:232 – 248, 2001. Computing: The Australasian Theory Symposium (CATS 2001).