TR-2012-20. Published by the Egerváry Research Group, Pázmány P. sétány 1/C, H–1117, Budapest, Hungary. Web site: www.cs.elte.hu/egres . ISSN 1587–4451.

Shortest paths in mixed graphs

Zoltán Király

December 31, 2012

Shortest paths in mixed graphs

Zoltán Király[⋆]

Abstract

Finding shortest paths in mixed graphs is NP-hard even when the weighting is conservative. We give an FPT algorithm for this problem, where the parameter k is the number of “negative trees”. We may always assume that the undirected edges have negative weight, and in a conservative weighting they form some trees – these trees are called the negative trees. Our algorithm extends to mixed graphs with conservative weighting, where in each strongly connected component there are at most k negative trees. Necessarily we can also detect whether the weighting is conservative or not. If it is not conservative we give a strongly polynomial algorithm for the shortest exact (and bounded) walk problems, where we are looking the shortest walk from s to t consisting of exactly (at most) K edges.

1 Introduction

Arkin and Papadimitriou proved in [1] that the problems of detecting negative cycles, and of finding the shortest path in the absence of negative cycles are both NP-hard in mixed graphs. In this paper we give an FPT algorithm for these problems. The APSP (All Pairs Shortest Paths) problem we are going to solve has two parts, first we must decide, whether the weighting is conservative, next, if the answer for the previous question is YES, then for all pairs s, t of vertices the task is to determine the length of the shortest path from s to t.

We may assume that the undirected edges have negative weights (otherwise they can be safely replaced by two opposite arcs), and that they form some trees (otherwise we conclude that the weighting is not conservative). These trees are called the negative trees. If the graph has n vertices and each of its strongly connected components contains at most k negative trees then our algorithm runs in $O(k! \cdot n^4)$ steps.

[⋆]Department of Computer Science and EGRES (MTA-ELTE), Eötvös University, Pázmány Péter sétány 1/C, Budapest, Hungary. Research was supported by grants (no. CNK 77780 and no. CK 80124) from the National Development Agency of Hungary, based on a source from the Research and Technology Innovation Fund.

2 Definitions

Throughout the paper $G = (V, E, A)$ denotes a mixed graph on $n = |V|$ vertices, where E is the set of undirected edges, and A is the set of directed edges, also called *arcs*. We will also call elements of $E \cup A$ edges, and denote by m the number of them: $m = |E| + |A|$. We also fix a length function (a weighting) $c : E \cup A \to \mathbb{R}$. For an edge uv the value $c(uv)$ will be called the weight or the length of this edge. We will sometimes add some arcs, but we maintain the property $m = O(n^2)$.

A walk from v_0 to v_ℓ (or a v_0v_ℓ-walk) is a sequence $v_0, e_1, v_1, e_2, v_2, \dots, v_{\ell-1}, e_\ell, v_\ell$, where $v_i \in V$ for all i, and for each j either $e_j = v_{j-1}v_j$ is an undirected edge between v_{j-1} and v_j, or $e_j = \overrightarrow{v_{j-1}v_j}$ is a directed edge from v_{j-1} to v_j. A walk is closed if $v_0 = v_\ell$. A closed walk is also called here a v_0v_0-walk; it is empty, if it contains no edges, in this case we also call it an empty path. A number ℓ of edges used by a walk W is denoted by $|W|$. The length $c(W)$ of a walk W is defined as $\sum_{e \in E(W)} c(e)$, where $E(W)$ denotes the multi-set of edges in W, where an edge has multiplicity r, if it appears r times along the walk (the length of the empty path is 0, and in this paper unconventionally the empty path will also be considered as an empty cycle). If W_1 is a s_1v-walk and W_2 is a vt_2-walk, then their concatenation is denoted by $W_1 + W_2$. For a walk $W = v_0, e_1, v_1, e_2, v_2, \dots, v_{\ell-1}, e_\ell, v_\ell$ we use the notation $W[v_i, v_j]$ for the corresponding part $v_i, e_{i+1}, \dots, e_j, v_j$, if $i < j$.

A (closed) walk is **undirected-simple** (US for short), if no undirected edge appears more than once in it. An undirected-simple walk will be called shortly as USW, and an undirected-simple closed walk will be called shortly as USCW. A walk is a *path*, if all the vertices $v_0, \dots, v_\ell$ are distinct. A walk (or a path) is called a *diwalk* (a *dipath* resp.), if all of its edges are directed. A closed walk is a *cycle*, if all the vertices $v_0, \dots, v_\ell$ are distinct, with the exception of $v_0 = v_\ell$.

Weighting c is conservative on G, if there are no negative length cycles. Arkin and Papadimitriou proved in [1] that the problems of detecting negative cycles, and of finding the shortest path in the case of a conservative weighting are both NP-hard.

The relation: "there is a path from s to t and also from t to s", is obviously an equivalence relation, its classes are called the strongly connected components of G. (Notice that if xy is an undirected edge then x and y are in the same strongly connected component.)

An algorithm is fixed parameter tractable (FPT) for a problem with input size n and parameter k, if there is an absolute constant γ and a function f, such that the running time is $f(k) \cdot O(n^\gamma)$. In this paper we give FPT algorithms for the APSP problem. We have several restricted versions, where the description of the algorithm is simpler, and it runs faster. For the most general version we have running time $O(k! \cdot n^4)$.

We may (and will) assume that no cycle is formed by negative cost undirected edges, because otherwise the algorithm may simply stop by saying "not conservative". Consequently the negative undirected edges form a forest, its non-trivial components (i.e., components having at least one edge) are called the negative trees.

In the simplest version we assume that there is just one negative tree and it is spanning. Next we give an algorithm for the case, where we still have only one negative tree, but it is not spanning. These algorithms are polynomial.

In the moderate version we set the parameter k as the number of negative trees in the graph, in this case our algorithm runs in $O(k!{\cdot}n^4)$. Next we show that consequently we can easily deduce another, much more general algorithm with the same running time, if we set the parameter k as the maximum number of negative trees inside any strongly connected component.

In the next section we show some reductions and lemmata. In Section 4 we give some polynomial algorithms for the case $k = 1$. In Section 5 we give an algorithm and prove its correctness and running time, if k is the total number of negative trees in the graph. Next, in Section 6 we extend it to the case, where k only bounds the number of negative trees in any strongly connected component. In these two sections our goal is only giving the length of the shortest paths, in Section 7 we detail how the actual shortest paths themselves can be found. Finally in Section 8 we examine the case, when c is not necessarily conservative, and we give a strongly polynomial algorithm for finding the shortest exact or bounded walk, i.e., the shortest walk consisting of exactly (at most, resp.) K edges.

3 Reductions, lemmata

In this section we formulate some lemmata that can be considered well-known, but it seems that they were not formulated exactly this way before.

Lemma 3.1. *Weighting c is conservative on graph G if and only if there is no negative USCW.*

Proof. If C is a negative cycle then it is also a negative USCW. On the other hand, suppose that C is a negative USCW with a minimum number of edges, such that C is not a cycle, that is there are $0 < i < j \leq \ell$ such that $v_i = v_j$. Now C decomposes into two USCWs with less edges, clearly at least one of them has negative length, a contradiction. □

Lemma 3.2. *If c is conservative on G, and Q is an US st-walk, then we also have an st-path P with $c(P) \leq c(Q)$.*

Proof. Let Q be a shortest US st-walk (which exists by the previous lemma and by the conservativeness of the weighting) with minimum number of edges. By the previous lemma, if $s = t$ then the empty path serves well as P. So we may assume that $s \neq t$ and Q is not a path, i.e., there are $0 \leq i < j \leq \ell$ such that $v_i = v_j$. Now Q decomposes to a US sv_i-walk Q_1, a USCW C through v_i and a US v_jt-walk Q_2. As c is conservative, by the previous lemma C is non-negative, so $c(Q_1 + Q_2) \leq c(Q)$, consequently $Q_1 + Q_2$ is a not longer US st-walk with less number of edges, a contradiction. □

Now we turn into describing the basic reductions we use. First of all, if we have an undirected edge with non-negative length, we replace it by two oppositely directed arcs with the same length. Neither the conservativeness nor any shortest path is changed.

After this reduction, if the undirected edges form any cycle, the algorithm easily detects this fact, and returns the output "not conservative". Thus we assume that all undirected edges have negative length, and they form some trees $T_1, T_2, \ldots, T_k$, these trees are called the *negative trees*.

We also use the following notations. $V(T_i)$ denotes the vertex set and $E(T_i)$ denotes the edge set of the negative tree T_i. By our arrangement $E = \bigcup E(T_i)$, and we use the notation V_T for $\bigcup V(T_i)$ and V_D for $V - V_T$.

We add some new arcs resulting the graph $\bar{G}$. For each undirected edge uv with negative length $c(uv) = -\ell$ we add arcs $\overrightarrow{uv}$ and $\overrightarrow{vu}$ with length ℓ. These arcs are called the *tree-arcs*.

Lemma 3.3. *If P a shortest st-path (a shortest ss-cycle with minimum number of edges) in $\bar{G}$ then it uses no tree-arcs. Consequently c is conservative on G if and only if it is conservative on $\bar{G}$. Moreover, if c is conservative on G then for any pair s, t of vertices, the length of the shortest st-path (ss-cycle, if $t = s$) is the same in G and in $\bar{G}$.*

Proof. The second and third statements follow easily from the first one. Let uv be an arbitrary undirected edge, it is enough to prove that either P does not contain any of $uv, \overrightarrow{uv}, \overrightarrow{vu}$, or it contains only the undirected edge uv. Path P contains vertex u at most once, so the statement follows by observing that undirected edge uv is the shortest among the three. If P is a shortest ss-cycle then the argument is almost the same, but we must note, that if P has two edges then the empty cycle is not longer and contains less number of edges. $\square$

From this point we denote by G the extended graph $\bar{G}$. Clearly the checks and reductions described here can be done in $O(n^2)$.

Summarized, from now on, $G = (V, E, A)$ *denotes a mixed graph, where the undirected edges have negative length, and they form a forest with nontrivial components $T_1, T_2, \ldots, T_k$, and moreover, if uv is an undirected edge then we also have $\overrightarrow{uv}, \overrightarrow{vu} \in A$ with length $-c(uv)$.*

Let G_A denote the directed subgraph $(V, A) = G - E$, which is formed by the arcs. By the Floyd-Warshall algorithm (see in any lecture notes, e.g., in [2]) it is easy to check whether c is conservative on G_A in $O(n^3)$ time, if it is not conservative then we also return with output "not conservative", and if it is conservative then this algorithm also calculates the length of all shortest paths (if vertex t is not reachable from vertex s then it gives $d(s, t) = +\infty$).

Lemma 3.4. *If $s \neq t$ are vertices, and P is a st-dipath, and G' denotes the graph we get by adding arc st with length $c(P)$, then conservativeness and shortest paths are preserved.*

Proof. If C is a negative cycle or Q is shortest path using arc st then this arc can be replaced by P giving a USCW (or USW, resp.) in the original graph with the same length. By Lemmas 3.1 and 3.2 we are done. $\square$

Lemma 3.5. *Suppose T is a negative tree, and that c is conservative on graph $G' = G - E(T)$ with distance function d'. Weighting c is conservative on G if and only if* *for any pair of vertices $u, v \in V(T)$ we have $d'(u, v) \geq -d^T(u, v)$, where $d^T(u, v)$ is the length of the unique path between u and v in T. Moreover, if c is conservative on G and $s, t \in V(T)$, then the length of the unique shortest st-path in G is $d^T(s, t)$.*

Proof. It is obvious that if we have a uv-path P in G' with $c(P) < -d^T(u, v)$ then c is not conservative. To prove the other direction suppose that C is a negative cycle in G, and for any maximal part of C that resides in G' we replace this part by the corresponding dipath consisting of solely tree-arcs. In this way we shortened C to a USCW C', whose all arcs are tree-arcs and whose all undirected edges are edges of T. However such a USCW clearly cannot have negative length. The second statement can be seen similarly, we may shorten any st-path to use only tree edges and tree arcs, and then we use Lemma 3.3, which also gives the uniqueness. □

The next lemma plays a central role in our algorithms.

Lemma 3.6. *Suppose that T is a negative tree, and also that c is conservative on G. If P is a shortest st-path using some vertex of V_T, then let u be the first vertex of P in V_T and let v be the last vertex of P in V_T. Then $P[u, v]$ uses only edges from $E(T)$.*

Proof. As c is conservative, any maximal part of $P[u, v]$ that uses no edges from $E(T)$ can be replaced by tree-arcs without increasing the length resulting finally the uv-path Q. Obviously Q is a shortest uv-path inside V_T, so by applying Lemma 3.5 for the subgraph induced by $V(T)$ we are done, because Q contains none tree-arc, hence we did not make any replacement. □

4 Polynomial algorithms for the case $k = 1$

First we give an $O(n^2)$ algorithm for the very restricted case, where we have only one negative tree T, and moreover it is a spanning tree of G. By Lemma 3.5, c is conservative, if and only if for each arc $\overrightarrow{uv}$ we have $c(\overrightarrow{uv}) \geq -d^T(u, v)$, and in this case for any pair $s, t \in V$ the length of the shortest path is $d^T(s, t)$. Consequently it is enough to give an $O(n^2)$ algorithm for calculating distances $d^T(s, t)$. We suppose that $V = \{1, \ldots, n\}$ and initialize a length-n all-zero array D_u for each vertex u. Then we fill up these arrays in a top-down fashion starting from the root vertex 1. Let $\mathcal{P}$ denote the subset of vertices already processed, initially it is $\{1\}$. If a parent u of an unprocessed vertex v is already processed, we process v: for each processed vertex x we set $D_v(x) = D_x(v) = D_u(x) + c(uv)$ and put v into $\mathcal{P}$.

Next we give an $O(n^4)$ algorithm for the case, where still we have only one negative tree T, but we do not assume it to be a spanning tree. Let $G_A = (V, A) = G - E$ denote the directed graph after deleting the undirected edges. With the Floyd-Warshall algorithm we check conservativeness and calculate pairwise distances $d_A(s, t)$ in graph G_A. (If c is not conservative on G_A then we also answer "not conservative", so further we suppose that c is conservative on G_A.) Then we calculate also distances $d^T(u, v)$ in time $O(n^2)$ as in the previous section. By Lemma 3.5, c is conservative on G, if and only if for all pairs $u, v \in V_T$ we have $d_A(u, v) \geq -d^T(u, v)$, this can be checked in time $O(n^2)$. We remained to calculate the pairwise distances. If P is a shortest

st-path, then it is either a dipath (having length $d_A(s,t)$), or it has a first undirected edge uu' and a last undirected edge $v'v$. The part $P[u,v]$ must reside inside $E(T)$ by Lemma 3.6.

Lemma 4.1. *If c is conservative on G, and T is the only negative tree, then the distance $d(s,t)$ is $d^T(s,t)$, if $s,t \in V(T)$, and generally it is*

$$d(s,t) = \min\Big(d_A(s,t),\ \min_{u,v\in V_T} [d_A(s,u) + d^T(u,v) + d_A(v,t)]\Big).$$

Proof. This is a consequence of Lemmas 3.5 and 3.6. □

These values are easily calculated for all pairs s,t in total time $O(n^4)$, so we are done.

5 FPT algorithm for the restricted case

Now we turn to the FPT algorithms. In this section we suppose that there are at most k negative trees in G. First we compute distances d^{T_i} for all i in total time $O(n^2)$. Next we compute distances d_A in graph $G_A = G - E$ in time $O(n^3)$ (and, again, if c is not conservative on G_A then we stop).

We define a recursive algorithm. If $k = 0$ (i.e., G is directed) then we are done. For each $1 \le i \le k$ we solve recursively the APSP problem on the graph $G_i = G - E(T_i)$, giving distance function d_i; if any of the recursive calls returns "not conservative" then we also stop and return this message.

By Lemma 3.5, we can check whether c is conservative on G using only distance functions d^{T_1} and d_1.

We remained to determine the pairwise distances $d(s,t)$. If s and t are in the same negative tree T_i then $d(s,t) = d^{T_i}(s,t)$. Next we determine $d(s,t)$ in the case $s \in V(T_i),\ t \in V - V(T_i)$ for some i. By Lemma 3.6 this path first goes inside T_i, then it leaves T_i at some vertex $u \in V(T_i)$, and then it resides in G_i. Consequently $d(s,t) = \min_{u\in V(T_i)}[d^{T_i}(s,u) + d_i(u,t)]$. Similarly, if $t \in V(T_i),\ s \in V - V(T_i)$ for some i then we have $d(s,t) = \min_{u\in V(T_i)}[d_i(s,u) + d^{T_i}(u,t)]$.

Finally we remained to determine $d(s,t)$ in the case $s,t \in V_D$. The shortest st-path P is either disjoint from V_T (in this case it has length $d_A(s,t)$), or it has a first vertex u in V_T and a last vertex v in V_T. Replacing the part $P[u,v]$ by a shortest uv-path we get a USW, so by Lemma 3.2 we may assume that $P[u,v]$ has length $d(u,v)$. On the other hand, for any $u,v \in V_T$ the concatenation of any su-dipath and any uv-path and any vt-dipath is a USW. Consequently $d(s,t) = \min\big(d_A(s,t),\ \min_{u,v\in V_T}[d_A(s,u) + d(u,v) + d_A(v,t)]\big)$. Observe, that we have already determined $d(u,v)$ for all possible pairs $u,v \in V_T$.

For the running time $R(n,k)$ we have the following recursion: $R(n,k) = k \cdot R(n,k-1) + O(n^4)$. As $R(n,1) = O(n^4)$, this can be upper-bounded by $R(n,k) = O((2\cdot k! - 1)\cdot n^4) = O(k!\cdot n^4)$. Summarized, we proved

Theorem 5.1. *If G has at most k negative trees then the algorithm given in this section correctly solves the APSP problem in time $O(k!\cdot n^4)$.*

6 General FPT algorithm

In this section we assume for every strongly connected component K of G, that c is conservative on K and we have at most k negative trees in K. Thus, by the previous section we can calculate APSP inside each strongly connected component in total time $O(k! \cdot n^4)$. The distance function inside K is denoted by d_K. In this situation clearly c is conservative on G if and only if it is conservative on every strongly connected component. It remains to calculate APSP in G for pairs s, t, that are in different strongly connected components.

We construct a new acyclic digraph D by first substituting every strongly connected component K by acyclic digraph D_K as follows. Suppose $V(K) = \{x_1^K, x_2^K, \ldots, x_r^K\}$, the vertex set of D_K will consists of $2r$ vertices, $\{a_1^K, a_2^K, \ldots, a_r^K,\ b_1^K, b_2^K, \ldots, b_r^K\}$. For each $1 \le i, j \le r$ the digraph D_K contains arc $a_i^K b_j^K$ with length $d_K(x_i^K, x_j^K)$.

In order to finish the construction of D, for every arc $x_i^K x_j^L$ of G connecting two different strongly connected components $K \ne L$, digraph D contains the arc $b_i^K a_j^L$ with length $c(x_i^K x_j^L)$. It is easy to see that D is truly acyclic and has $2n$ vertices. As D is a simple digraph, paths can be given by only listing the sequence of its vertices. We can calculate APSP in D in time $O(n^3)$ by the method of Morávek [4] (see also in [2]), if we run this famous algorithm from all possible sources s. It gives distance function d_D (where, if t is not reachable from s then we write $d_D(s,t) = +\infty$). The total running time is still $O(k! \cdot n^4)$. We remark, that if every strongly connected component is either directed or it has a spanning negative tree, then the running time is $O(n^3)$.

Theorem 6.1. *Suppose $s = x_{i_0}^{K_0} \in V(K_0)$ and $t = x_{j_r}^{K_r} \in V(K_r)$, where $K_0 \ne K_r$ are different strongly connected components of G. Then the shortest st-path in G has length exactly $d_D(a_{i_0}^{K_0}, b_{j_r}^{K_r})$.*

Proof. Vertex t is not reachable from s in G, if and only if $b_{j_r}^{K_r}$ is not reachable from $a_{i_0}^{K_0}$ in D. Otherwise suppose first that $a_{i_0}^{K_0}, b_{j_0}^{K_0}, a_{i_1}^{K_1}, b_{j_1}^{K_1}, \ldots, a_{i_r}^{K_r}, b_{j_r}^{K_r}$ is a shortest path P in D. For $0 \le \ell \le r$ let path P_ℓ be a shortest path in G from $x_{i_\ell}^{K_\ell}$ to $x_{j_\ell}^{K_\ell}$, this path obviously goes inside K_ℓ. We construct st-path Q in G with the same length as P has in D: $Q = P_0 + x_{j_0}^{K_0} x_{i_1}^{K_1} + P_1 + x_{j_1}^{K_1} x_{i_2}^{K_2} + P_2 + \ldots + P_{r-1} + x_{j_{r-1}}^{K_{r-1}} x_{i_r}^{K_r} + P_r$.

For the other direction, suppose that there are strongly connected components $K_0, K_1, \ldots, K_r$, such that the shortest st-path Q in G meets these components in this order, and that for all ℓ the path Q arrives into K_ℓ at vertex $x_{i_\ell}^{K_\ell}$ and leaves K_ℓ at vertex $x_{j_\ell}^{K_\ell}$. As Q is a shortest path it clearly contains a path of length $d_{K_\ell}({x_{i\ell}}^{K_\ell}, x_{j_\ell}^{K_\ell})$ inside K_ℓ for each ℓ, consequently the following path has the same length in D: $P = a_{i_0}^{K_0}, b_{j_0}^{K_0}, a_{i_1}^{K_1}, b_{j_1}^{K_1}, \ldots, a_{i_r}^{K_r}, b_{j_r}^{K_r}$. $\square$

Corollary 1. *If there is an absolute constant γ, such that in any strongly connected component of G there are at most γ negative trees, then there is a polynomial time algorithm for the APSP problem, that runs in $O_\gamma(n^4)$.*

7 Finding the paths

In this section we assume that c is conservative on G.

We usually are not only interested in the lengths of the shortest paths, but also some (implicit) representation of the paths themselves. The requirement for this representation is that for any given s and t, one shortest st-path P must be recovered from it in time $O(\ell)$, if ℓ is the number of edges in P.

It is well known (see e.g., in [2]), that both the algorithm of Floyd and Warshall and the algorithm of Morávek can compute predecessor matrices Π (by increasing the running time by a constant factor only), with the property that for each $s \neq t$ the entry $\Pi(s,t)$ points to the last-but-one vertex of a shortest st-path. This representation clearly satisfies the requirement described in the previous paragraph.

For a graph H let Π_H denote the predecessor matrix of this type, and first suppose, that for each strongly connected component K we computed Π_K and that we also computed Π_D. Then Π_G is also easily computable as follows. Suppose that $s = x_{i_0}^{K_0}$ and $t = x_{j_r}^{K_r}$, and $\Pi_D(a_{i_0}^{K_0}, b_{j_r}^{K_r}) = a_{i_r}^{K_r}$. If $i_r \neq j_r$, then define $\Pi_G(s,t) = \Pi_{K_r}(x_{i_r}^{K_r}, x_{j_r}^{K_r})$, otherwise let $b_{j_{r-1}}^{K_{r-1}} = \Pi_D(a_{i_0}^{K_0}, a_{i_r}^{K_r})$ and define $\Pi_G(s,t) = x_{j_{r-1}}^{K_{r-1}}$.

It remained to compute the predecessor matrices Π_K in the case, where K is a strongly connected component of G. In accord with Section 5 from now on we call K as G (and forget the other vertices of the graph), and the matrix we are going to determine is simply Π.

If s and t are vertices of the same negative tree T_i, then the method given in the first paragraph of Section 4 easily calculates $\Pi(s,t) = \Pi_{T_i}(s,t)$ (it is correct by Lemma 3.5). Next we call Floyd-Warshall on G_A (the undirected edges of G are deleted) and it can give Π_{G_A}, then by the recursive calls we determine matrices Π_{G_i} for all i.

First suppose that $t \in V(T_i)$ and $s \in V - V(T_i)$, and vertex $u^* \in V(T_i)$ minimizes $\min_{u \in V(T_i)}[d_i(s,u) + d^{T_i}(u,t)]$. If $u^* \neq t$ then $\Pi(s,t) = \Pi_{T_i}(u^*,t)$, otherwise, if $u^* = t$ then $\Pi(s,t) = \Pi_{G_i}(s,t)$.

Next suppose that $s \in V(T_i)$ and $t \in V - V(T_i)$, and vertex $u^* \in V(T_i)$ minimizes $\min_{u \in V(T_i)}[d^{T_i}(s,u) + d_i(u,t)]$. In this case $\Pi(s,t) = \Pi_{G_i}(u^*,t)$.

Finally suppose that $s,t \in V_D$. If $d(s,t) = d_A(s,t)$ then $\Pi(s,t) = \Pi_{G_A}(s,t)$. Otherwise suppose that vertices $u^*, v^* \in V_T$ minimizes $\min_{u,v \in V_T}[d_A(s,u) + d(u,v) + d_A(v,t)]$. In this case (as $t \notin V_T$) we have $\Pi(s,t) = \Pi_{G_A}(v^*,t)$.

8 Strongly polynomial algorithms for shortest exact and bounded walk problems

In this section we give strongly polynomial algorithms for the shortest exact and bounded walk problems. Given a mixed graph G with length function c (we do not assume any conservativeness), vertices s,t and a number K, the shortest K-exact walk problem asks for the shortest st-walk having exactly K edges. The similar shortest K-bounded walk problem asks for the shortest st-walk having at most K edges. Of course, there is a possibility that no K-exact (or K-bounded) st-walk exists at all, in this case the algorithm should return with symbol $+\infty$).

The input size is considered as $n+m+\log(K)$, where $m=|E|+|A|$. We are going to give algorithms, which have running time polynomial in the input size, regardless of the weights. To be able to do this, we use the word RAM model, where n as well as any length c_{uv} is considered a small number that fit into one word, and two words can be added in constant time. K is considered as a large number that fits into $\log K$ words, so, e.g., the integer division $K:n$ needs $O(\log K)$ time.

Remark: The results in this section could be described by speaking only about digraphs, as all undirected edges can be replaced by two oppositely directed edges with the same length. We use the language of the more general mixed graphs only for consonance with the first part of the paper. It would be very interesting to give a polynomial algorithm for the K-exact US st-walk problem, however we conjecture that this problem is NP-hard. In this section we are not trying to optimize the running time, our goal is only proving that it is polynomial.

The algorithm of Bellman and Ford can be also used to calculate shortest K-exact or K-bounded walks (see e.g., in [2]), but this method uses K iterations, so it needs Km steps. If $K \le 2n^2$ then we use this method for our problems. Karp observed [3], that with this method the minimum mean cycle can also be computed in $O(nm)$, where for any walk W we define the mean of W as $\mu(W)=c(W)/|W|$. We use his method as follows, starting with graph G_0 on vertex set V_0, which arises from G by deleting vertices that are not reachable from s, and also those vertices, from where t is not reachable (G_0 can be calculated by 2 BFSs in time $O(m)$).

First we calculate the minimum mean cycle C_0 in G_0, and U_0 denotes its vertex set $U_0=V(C_0)$, $V_1=V_0-U_0$ and $G_1=G-U_0$. Then we repeat the same procedure for $G_1,G_2,\ldots,G_r$, the minimum mean cycle in G_i is C_i, and U_i denotes its vertex set $U_i=V(C_i)$, $V_{i+1}=V_i-U_i$, $n_i=|V_i|$ and $G_{i+1}=G-U_0-\ldots-U_i$. We stop when G_{r+1} is acyclic, or if t is not reachable from s in G_{r+1}, or if either s or t is in U_r. For each $0\le i\le r$ let u_i denote an arbitrary vertex of C_i, $c_i=|C_i|$ and $\mu_i=\mu(C_i.)$ Evidently every closed walk in G_i has a mean at least μ_i.

A K-exact st-walk is called *j-nice* (for an integer $0\le j\le r$), if it is inside G_j, and if it has the form $P+q\cdot C_j[u_j,u_j]+Q$, where P is an su_j-walk, Q is a u_jt-walk, and $q\cdot C_j[u_j,u_j]$ denotes the closed walk, which starting from u_j traverses cycle C_j q times arriving also at u_j. A j-nice walk is *very j-nice*, if among the shortest j-nice walks it minimizes $|P|+|Q|$.

Theorem 8.1. *If $K>2n^2$ and a K-exact st-walk exists, then there is also a very j-nice one with $0\le j\le r$, $|P|\le n_jc_j$ and $|Q|\le n_jc_j$.*

Proof. Suppose $W=v_0,e_1,v_1,\ldots,e_K,v_K$ is a shortest K-exact st-walk, and that j is the smallest index, such that W goes through U_j. First we show that there is another shortest K-exact st-walk W', which is j-nice. There is vertex $u'_j\in U_j$ contained in W, by symmetry we may assume, that there is an index ℓ, such that $v_\ell=u'_j$ and $\ell>n_jc_j$. Now there is a vertex v that appears more than c_j times before the ℓth position, and, by the pigeonhole principle, there are indices $0\le i_1<i_2<\ell$ such that $v_{i_1}=v_{i_2}=v$ and that i_2-i_1 is divisible by c_j. We get W' as follows. Cut off the part between v_{i_1} and v_{i_2} from W and replace it (after $v_\ell=u'_j$) by $q\cdot C_j[u'_j,u'_j]$, where $q=(i_2-i_1)/c_j$.

Using our notation $W' = W[v_0, v_{i_1}] + W[v_{i_2}, v_\ell] + q \cdot C_j[u'_j, u'_j] + W[v_\ell, v_K]$. Obviously W' is not longer than W. Observe that W' goes through u_j, so it has decomposition $P + q' \cdot C_j[u_j, u_j] + Q$, with possibly $q' = 0$.

Let $W'' = P + q \cdot C_j[u_j, u_j] + Q$ be a very j-nice walk, we claim that $|P|, |Q| \leq n_j c_j$. If this is not the case suppose e.g., that $|Q| > n_j c_j$, that is, by pigeonhole principle, there are indices $|P| + qc_i \leq i_1 < i_2 \leq K$ such that $v_{i_1} = v_{i_2} = v$ and that $i_2 - i_1$ is divisible by c_j. Again, we can replace the closed walk part of Q between v_{i_1} and v_{i_2} by $((i_2 - i_1)/c_j) \cdot C_j$ (starting and ending with u_j), contradicting to the fact, that W'' was very nice. □

We are ready to describe our algorithm for finding the shortest K-exact st-walk. If $K \leq 2n^2$ then we use the Bellman-Ford method that runs in $O(n^2 m)$. Otherwise, by the algorithm of Karp we can calculate for all i the sets U_i, the graphs G_i and the cycles C_i in total time $O(n^2 m)$.

First we "process" graph G_{r+1}. As K is big, either there is no st-walk or every st-walk is shorter than K, so we set $min = +\infty$. Then for each $j = 0, 1, \ldots r$ we find the nicest walk in G_j passing through u_j as follows. We try all possible values $0 \leq a \leq n_j c_j$ for $|P|$, calculate $b' = K - a \mod c_j$, and try all possible $0 \leq b \leq n_j c_j$, $b \equiv b' \mod c_j$ for $|Q|$. For each pair (a, b) constructed, we calculate the shortest a-exact su_j walk P, and the shortest b-exact $u_j t$-walk Q in graph G_j. Whenever we find such walks, we calculate the length of the corresponding nicest walk: $length = c(P) + c(Q) + (K - a - b)\mu_j$. If $length < min$ then replace min by $length$. For any given j, calculating all possible shortest a-exact and b-exact walks can be done in $O(n_j c_j m)$ time by Bellman-Ford. We also calculate the reminder of $K : c_j$ in $O(\log K)$ time. Then, in the right order, we can calculate all the possible values in question (thus also the minimum) by doing $n^2 c_j$ additions/subtractions. By induction one can easily show, that $\sum_{j=0}^{r} n_j c_j \leq n^2$ and $\sum_{j=0}^{r} n_j^2 c_j \leq n^3$, consequently the total time needed is $O(n^3 + n^2 m + \log K) = O(n^2 m + \log K)$.

Theorem 8.2. *The algorithm given above correctly calculates the length of the shortest K-exact st-walk in time $O(n^2 m + \log K)$.*

Proof. Clearly if there is any K-exact st-walk, then our algorithm find one. If j is the smallest index, such that the optimal walk touches U_j then by Theorem 8.1 it is a very j-nice walk and it is found by the algorithm. □

For the problem of finding a shortest K-bounded st-walk we use essentially the same algorithm, so we only detail the differences. First of all, if we have $\mu_0 \geq 0$ we need to calculate the shortest st-path (assuming also here $K > 2n^2$). Otherwise we calculate the graph G_i until we reach $\mu_{r+1} \geq 0$. In G_{r+1} we calculate shortest st-path, and for each $0 \leq j \leq r$ we calculate the shortest K'-exact st-walk for each $K - c_j < K' \leq K$. Here the number of possible pairs (a, b) can be as high as $n_j^2 c_j^2$, so the total time needed is $O(n^4 + \log K)$.

Acknowledgment

The author is thankful to András Frank, who, as always, asked the right questions.

References

[1] E.M. Arkin, C.H. Papadimitriou On negative cycles in mixedgraphs, *Operations Research Letters* **4** (3) (1985), pp. 113–116.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein Introduction to Algorithms, *MIT Press, Cambridge* third edition, (2009)

[3] R.M. Karp A characterization of the minimum cycle mean in a digraph, *Discrete Mathematics* **23** (1978), pp. 309–311.

[4] J. Morávek A note upon minimal path problem, *Journal of Mathematical Analysis and Applications* **30** (1970), pp. 702–717.