Compromis espace-temps pour le problème de k plus courts chemins simples

Ali Al Zoobi, David Coudert, Nicolas Nisse

Compromis espace-temps pour le problème de k plus courts chemins simples †‡

Ali Al Zoobi [1] et David Coudert [1] et Nicolas Nisse[1]

[1] *Université Côte d'Azur, Inria, CNRS, I3S, France*

Le problème de trouver k plus courts chemins simples (sans répétition de sommets) entre deux sommets dans un graphe a été largement étudié du point de vue de l'ingénierie algorithmique. Kurz et Mutzel (2016) ont proposé l'algorithme SB (pour *Sidetrack Based*) basé sur le concept de déviations, qui est actuellement la méthode la plus rapide en pratique. Dans ce travail, nous proposons deux améliorations de cet algorithme. Nous montrons tout d'abord comment accélérer l'algorithme SB en utilisant des mises à jour dynamiques d'arbres de plus courts chemins. Nos simulations réalisées sur certains réseaux routiers avec environ un demi-million de sommets et un million d'arcs montrent que notre amélioration donnent une accélération moyenne d'un facteur 1,5 à 2 avec une consommation de mémoire similaire à celle de l'algorithme SB. Notre principale contribution est un second algorithme réalisant un compromis entre temps d'exécution et mémoire utilisée. Notre algorithme permet de réduire significativement la mémoire de travail (d'un facteur $1,5$ à 2) au prix d'une légère augmentation du temps d'exécution.

Mots-clefs : k plus courts chemins simples, algorithmes de graphes, compromis complexité espace-temps

1 Introduction

The classical k shortest paths problem (kSP) returns the top-k shortest paths (SP) between a source and a destination in a digraph. This problem has numerous applications in various kinds of networks (road and transportation networks, communications networks, social networks, etc.) and is also used as a building block for solving optimization problems. Let $D = (V,A)$ be a digraph with n vertices and m arcs, and with a weight function $\omega : A \to \mathbb{R}^+$ on its arcs. A (directed) s-t path is a sequence $(s = v_0, v_1, \cdots, v_l = t)$ of vertices starting from s and ending at t, such that $(v_i, v_{i+1}) \in A$ for all $0 \leq i < l$. It is called simple if it has no repeated vertices, i.e., $v_i \neq v_j$ for all $0 \leq i < j \leq l$. The weight of a path is the sum of the weights of its arcs and the distance $d_D(u,v)$ between two vertices $u, v \in V$ is the minimum weight of a u-v path in D. A set $\mathcal{P}$ of s-t paths is a top-k set of s-t paths if $|\mathcal{P}| = k$ and no s-t path not in $\mathcal{P}$ has weight strictly less than some path in $\mathcal{P}$. Eppstein's algorithm [Epp98] has the best time-complexity, $O(m + n\log n + k)$, for this problem.

An important variant of this problem is the k shortest *simple* paths problem (kSSP) which adds the constraint that reported paths must be simple. The algorithm with the best known time complexity for solving the kSP problem has been proposed by Yen [Yen71], with time complexity in $O(kn(m + n\log n))$. Since, several proposals have been made to improve the practical efficiency of the algorithm [Fen14,KM16].

Recently, Kurz and Mutzel [KM16] obtained a tremendous practical running time improvement. The key idea of their Sidetrack Based (SB) algorithm was to postpone as much as possible the computation of trees that are done earlier by Yen's algorithm. We present a slight improvement of SB algorithm and a modification of it allowing to establish a tradeoff between the running time and the memory consumption.

Our contribution. Section 2 presents Kurz and Mutzel's SB algorithm with a particular attention to the used data structures. This allows us, in Section 3, to describe a first slight improvement, using dynamic updates of shortest path trees, and resulting to a notable speed up of SB algorithm, with an average speed up by a factor of 1.5 to 2 and with a similar working memory consumption. Our main contribution, presented in Section 3, is a more involved adaptation of the SB algorithm, enabling to significantly reduce the working memory (with a factor of 1.5 to 2) at the cost of a small increase of the running time. We conclude with the results of our simulations comparing the performances of our algorithms with the previous ones.

†The full version of this paper can be found here: `https://hal.archives-ouvertes.fr/hal-02465317`.

‡This work has been supported by the UCA[JEDI] Investments in the Future project managed by the National Research Agency (ANR-15-IDEX-01), project MULTIMOD (ANR-17-CE22-0016), project Digraphs (ANR-19-CE48-0013), and by Région Sud PACA.

2 Kurz and Mutzel's algorithm

We give a short presentation of Kurz and Mutzel's Sidetrack Based (SB) algorithm in order to explain our contributions in the next section. Let $(D = (V,A), \omega)$ be an arc-weighted digraph, $s,t \in V$ and $k \in \mathbb{N}$. The goal of SB algorithm is to compute a top-k set $\mathcal{P}$ of simple shortest s-t paths (assuming they exist).

Compact representation of a path. The SB algorithm is based on a data structure generalizing the representation of a path proposed by Eppstein [Epp98]. Let us present this data structure briefly below.

An *in-branching* T rooted at t is a sub-digraph of D that induces a tree containing t and such that every $u \in V(T) \setminus \{t\}$ has exactly one out-neighbor (that is, all paths in T go toward t) and, for every $u \in V(T)$, the weight of the (unique) u-t path $P_{ut}(T)$ in T equals $d_D(u,t)$, i.e, $P_{ut}(T)$ is a shortest u-t path (both in T and D). Let $P = (v_0, v_1, \cdots, v_r)$ be any path in D and $i < r$. Any arc $a = v_i w \neq v_i v_{i+1}$ is called a *deviation* of P at v_i. Any path $Q = (v_0, \cdots, v_i, w, w_1, \cdots, w_\ell = t)$ where $w, w_1, \cdots, w_\ell = t$ is a shortest wt path in D is called an *extension* of P at a (or at v_i). Note that neither P nor Q is required to be simple.

Kurz and Mutzel's proposed a compact representation of paths using sequences of in-branchings and deviations. Precisely, the sequence $\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1})$ (with $e_i = (v_i, w_i)$ for all $i \leq h$) represents the path P starting at s, following T_0 until the tail v_0 of e_0, then the deviation e_0, then T_1 until it reaches the tail v_1 of e_1, *etc.* until it reaches the head w_h of e_h, plus (possibly) a path from w_h to t in T_{h+1}. That is, P is the sequence of vertices of the paths $P_{sv_0}(T_0), P_{w_0v_1}(T_1), \cdots, P_{w_{h-1}v_h}(T_h)$ followed by the vertices of $P_{w_ht}(T_{h+1})$ if this latter path exists. Two consecutive in-branchings T_i and T_{i+1} are not necessarily distinct. SB algorithm ensures that, if P is an s-t path (i.e., if $P_{w_ht}(T_{h+1})$ exists), then the subpath $Pref$ of P going from s to w_h $(v_0, \cdots, w_h)$ is always simple and P is not simple only if $P_{w_ht}(T_{h+1})$ intersects $Pref$.

Sidetrack Based (SB) Algorithm. We are now ready to present SB algorithm. Roughly, SB algorithm uses a set $\mathcal{C}$ to manage candidate paths that are encoded using the above data structure. Sequentially, it extracts a shortest element ε from $\mathcal{C}$. If ε represents a simple path P, this path is added to the output $\mathcal{P}$ and the representations of its extensions are computed and added to $\mathcal{C}$. Otherwise, SB algorithm attempts to modify ε by instantiating its last in-branching (this is one bottleneck both for the time and space complexities since an in-branching is actually computed by applying Dijkstra's algorithm, and is stored). If this computation leads to the representation of a simple path, then it is added to $\mathcal{C}$. Otherwise, ε is discarded. SB algorithm goes on iteratively until it has found k paths. The initialisation consists in computing a first in-branching T_0 in D (using Dijkstra's alg.) and so a shortest s-t (simple) path $P_{st}(T_0)$ and adding its representation to $\mathcal{C}$.

More precisely, the set $\mathcal{C}$ is a min-heap in which the weight of an element is a lower bound on the weight of the path it represents. Each element μ in $\mathcal{C}$ has the form $\mu = (\varepsilon = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), lb, \zeta)$ where each in-branching $T_{h'}$ (with $h' \leq h$) is already computed and lb is a lower bound of the weight of the path represented by ε. The value ζ is a boolean indicating whether the path represented by ε is known to be simple. If so, it will follow from the construction that T_{h+1} has already been computed, else T_{h+1} must be first computed to know if ε represents a simple path. For the initialization, the in-branching T_0 is computed and the element $((T_0), \omega(P_{st}(T_0)), \zeta = 1)$ is inserted in $\mathcal{C}$.

SB algorithm iteratively extracts elements from $\mathcal{C}$ by minimum weight (with a priority to representation of simple paths to break ties) until k paths are obtained or $\mathcal{C}$ is empty. When an element $\mu = (\varepsilon = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), lb, \zeta)$ is extracted from $\mathcal{C}$, two cases are distinguished :

- **Case $\zeta = 1$.** Then, ε represents a simple path $P = (v_0 = s, \cdots, v_i = w_h, \cdots, v_r = t)$ and all the in-branchings T_i's have already been computed. In this case, the path P is added to the output $\mathcal{P}$. Then, for every $i \leq j < r$, and for every deviation $e = (v_j, w)$ at v_j, let $P_{wt}(T_{h+1})$ be the shortest path from w to t in T_{h+1} and let $Q(v_j, e) = (v_0, \cdots, v_j, w, P_{wt}(T_{h+1}))$. If $Q(v_j, e)$ is simple (this can be verified efficiently using a trick of Feng [Fen14]), the representation $\mu' = ((T_0, e_0, \cdots, e_h, T_{h+1}, e = (v_j, w), T_{h+1}), lb(e), \zeta = 1)$ is added to $\mathcal{C}$ with $lb(e) = \omega(Q(v_j, e))$ as a key (note that the computation of $lb(e)$ is done in constant time since, in particular, T_{h+1} is already computed). Otherwise ($Q(v_j, e)$ is not simple), the representation $\mu'' = (\varepsilon'' = (T_0, e_0, \cdots, e_h, T_{h+1}, e = (v_j, w), T'), lb(e), \zeta = 0)$ is added to $\mathcal{C}$, where T' is the *name* of the in-branching of $D \setminus \{v_0, \cdots, v_j\}$ whose actual computation is postponed, and $lb(e) = \omega(Q(v_j, e))$ is a lower bound on the weight of the path represented by ε''.
- **Case $\zeta = 0$.** In this case, the algorithm checks for the existence of a w_h-t path $P_{wt}(T_{h+1})$. To do so, the in-branching T_{h+1} (whose computation has been postponed) is computed. Note that T_{h+1} is an

in-branching in $D \setminus \{v_0, \cdots, v_h\}$, which ensures that, if $P_{wt}(T_{h+1})$ is found, the path $P_{new} = (s = v_0, \cdots, v_h, P_{w_h t}(T_{h+1}))$ is guaranteed to be simple. Moreover, P_{new} has weight $\omega(P_{new}) = \omega((s = v_0, \cdots, v_h, w_h)) + \omega(P_{w_h t}(T_{h+1}))$. Then, the representation $\mu' = (\varepsilon' = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), \omega(P_{new}), \zeta = 1)$ is added to $\mathcal{C}$. Finally, if no w_h-t path can be found in T_{h+1}, μ is discarded.

So far, SB algorithm has out-performed all other algorithms for solving the k-Shortest Simple Paths problem in terms of running time (e.g., it is ten times faster than Feng's algorithm [Fen14] in large networks). However, it has the drawback to have an important consumption of working memory (much more than Feng's algorithm that stores a single in-branching, while keeping the whole description of paths it computes).

3 Our Contributions

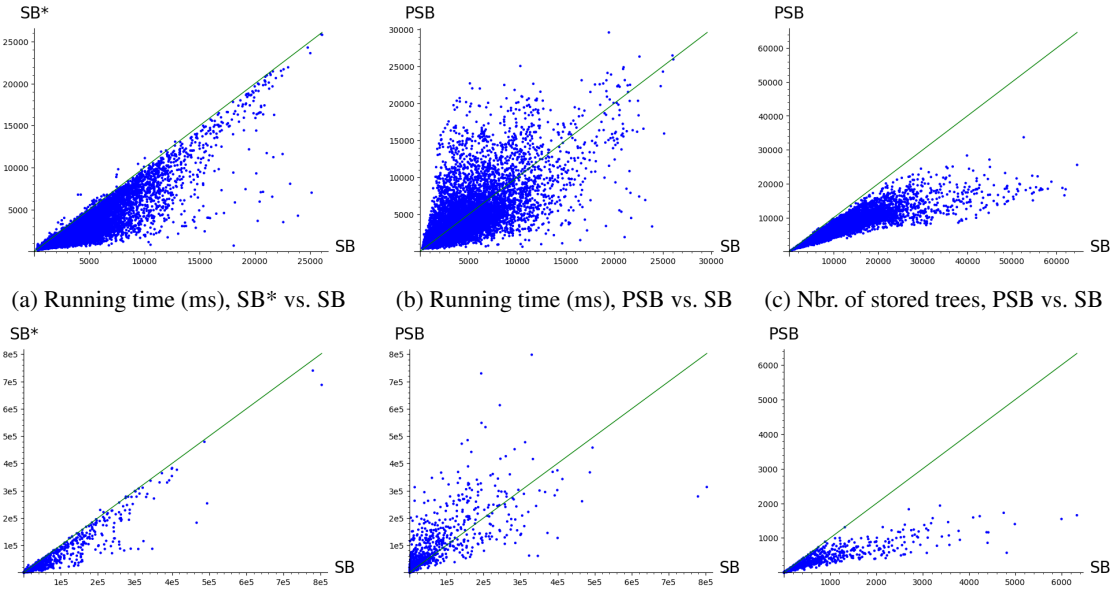
SB* Algorithm. First, we propose a slight modification of SB algorithm (called SB*). Precisely, each time a representation $(T_0, e_0, T_1 \cdots, e_{h-1} = (u_{h-1}, v_{h-1}), T_h, e_h = (u_h, v_h), T_{h+1})$ of a non simple path is extracted from $\mathcal{C}$ with T_{h+1} not computed yet (i.e., it is only a *name*), while SB algorithm computes T_{h+1} from scratch our algorithm does not. Instead, SB* algorithm creates a copy T of T_h, discards vertices of the path from v_{h-1} to u_h in T_h, and updates the in-branching T using standard methods for updating an in-branching.Then, the *name* T_{h+1} is instantiated as the new in-branching T. It is clear that SB* algorithm computes (and store) exactly the same number of in-branchings as SB algorithm. As demonstrated by the simulations below, SB* algorithm is up to twice faster than SB algorithm with the same memory consumption.

Parsimonious SB Algorithm. Now, let us present our main algorithm, called PSB, which is an adaptation of SB algorithm allowing to reduce the memory consumption due to the storage of all in-branchings computed by SB algorithm. Here, we only focus on the differences between SB and PSB algorithms.

The main difference is that PSB algorithm stores two types of elements in $\mathcal{C}$. The first type, of the form $(\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h = (v_h, w_h), T_{h+1}), lb)$, represents a simple s-t path P of weight lb. Contrary to SB algorithm, the in-branching T_{h+1} has not necessarily been computed yet. The second type, of the form (ε, Dev, lb), contains an extra field Dev (explained below) and, in this case, all of the in-branchings $T_1, \cdots, T_{h+1}$ are already computed.

Let us start considering a step of PSB algorithm when an element $\mu = (\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h = (v_h, w_h), T_{h+1}), lb)$ representing a simple path P is extracted from $\mathcal{C}$. T_{h+1} is computed at this step (if not already done) which allows to output P. Then, PSB algorithm adds $P = (s = v_0, \cdots, v_i = w_h, \cdots, v_r = t)$ to $\mathcal{P}$ and (as SB algorithm) for every $v \in \{v_i, \cdots, v_r\}$, and every deviation e with tail v, the extension $Q(v,e)$ is considered. If $Q(v,e)$ is simple (again, checking whether an extension is simple or not is done using the trick of Feng), then $\mu' = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1}), \omega(Q(v,e)))$ is added to $\mathcal{C}$. Otherwise, the deviation e is added to a set Dev (initially empty). Once all deviations have been considered, the (unique) element (ε, Dev, lb') is added to $\mathcal{C}$, where $lb' = \min_{f_j = (u_j, u'_j) \in Dev} \omega(Q(u_j, f_j))$. That is, Dev is the set of all "non simple deviations" of P at the vertices between w_h and t, and lb' is a lower bound on the weight of the extensions at a deviation in Dev. The important difference between SB and PSB algorithms comes from the fact that non simple extensions are considered as a unique object by PSB.

Now, let us consider a step when PSB algorithm extracts an element $\mu = (\varepsilon = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}), Dev = \{f_1, \cdots, f_j = (u_j, u'_j), \cdots, f_l\}, lb)$ from $\mathcal{C}$. As mentioned above, in this case, ε encodes a simple s-t path $(v_0, \cdots, v_r)$. Let $1 \leq min \leq l$ be the smallest integer such that $lb = \omega(Q(u_{min}, f_{min}))$. Then, PSB algorithm proceeds as follows. For j decreasing from l to min, an in-branching T'_j in $D \setminus \{v_0, \cdots, v_{i_j} = u_j\}$ is computed (but not stored !) until a path $P_{u'_j t}(T'_j)$ from u'_j to t is discovered (if no such path exists, j is decreased by one). If $P_{u'_j t}(T'_j)$ exists, then $\varepsilon_j = (T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, f_j, T'_j)$ represents a simple s-t path of weight $lb_j = \omega((v_0, \cdots, v_{i_j})) + \omega(f_j) + \omega(P_{u'_j t}(T'_j))$. Then, the element $\mu_j = (\varepsilon_j, lb_j)$ is added to $\mathcal{C}$, but T'_j is not stored (PSB algorithm might have to recompute it later). A 2^{nd} key improvement is that to speed up the computation, T'_j is actually computed by updating T'_{j+1}.Then, only when $j = min$, the in-branching T'_{min} is stored and $\mu_{min} = (\varepsilon_{min}, lb_{min})$ is added to $\mathcal{C}$. The reason why T'_{min} is stored (while other T'_j are not) is that μ_{min} is expected to be extracted soon from $\mathcal{C}$ (because the path represented by ε_{min} is expected to be short) and we want to avoid the recomputation of T'_{min}. Finally, the element $\mu' = (\varepsilon, Dev' = \{f_1, \cdots, f_{min-1}\}, lb')$ is added to $\mathcal{C}$, where lb' is the minimum weight over the non simple deviations in Dev'.



(a) Running time (ms), SB* vs. SB (b) Running time (ms), PSB vs. SB (c) Nbr. of stored trees, PSB vs. SB

(d) Running time (ms), SB vs. SB* (e) Running time (ms), SB vs. PSB (f) Nbr. of stored trees, PSB vs. SB

FIGURE 1: Comparison of the running time of SB versus SB* (Fig. 1a) and SB versus PSB (Fig. 1b) on Rome (resp. Fig. 1d and Fig. 1e on Colorado), and comparison of the number of stored trees for SB versus PSB (Fig. 1c) on Rome, (resp. Fig. 1f on Colorado). Each dot corresponds to one pair source/destination, among 1 000 randomly chosen pairs.

The correctness follows from the one of the SB algorithm. Moreover, since most of the computed in-branchings are not stored, the working memory used by PSB is significantly smaller than for SB algorithm.

Experimental evaluation. We have implemented the algorithms SB, SB* and PSB in C++ and our code is publicly available at `https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths`. We have evaluated the performances of these algorithms on several road networks [DGJ]. Here we present the ones in a "small" network (Rome, $n = 3\,353$, $m = 8\,870$) and in a "large" one (Colorado, $n = 435\,666$, $m = 1\,057\,066$). All computations have been done on a computer equipped with 2 quad-core 3.20GHz Intel Xeon W5580 processors and 64GB of RAM. Our simulations show that our improvement SB* of SB algorithm allows to decrease the running-time by a factor between 1,5 and 2 in average. In particular, for both networks, SB* algorithm is, for most of the queries, faster than SB algorithm (Figures (1a) and (1d)). The simulations comparing PSB and SB algorithms show a significant reduction of the working memory (number of stored trees) when using PSB (Figures (1c) and (1f)). In term of running time, SB algorithm is slightly faster in average but Figures (1b) and (1e) indicate that globally, they are quite comparable.

Conclusion. To obtain a better tradeoff between space and time, a future work consist in determining a threshold τ such that an in-branching T is stored only if it is used to extract a path with weight at most τ.

Références

[DGJ] C. Demetrescu, A. Goldberg, and D. Johnson. 9th dimacs implementation challenge - shortest paths.

[Epp98] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2) :652–673, 1998.

[Fen14] G. Feng. Finding k shortest simple paths in directed graphs : A node classification algorithm. *Networks*, 64(1) :6–17, 2014.

[KM16] D. Kurz and P. Mutzel. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. In *Int. Symp. on Alg. and Comp. (ISAAC)*, volume 64 of *LIPIcs*, pages 49 :1–49 :13, 2016.

[Yen71] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11) :712–716, 1971.