Practical computation of simple paths with length and diversity constraints in complex and multimodal networks

Ali Al Zoobi

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Calcul pratique de chemins simples avec des contraintes de longueur et de diversité dans des réseaux complexes et multimodaux

Ali AL ZOOBI

Inria Sophia Antipolis Méditerranée & Laboratoire I3S

Présentée en vue de l'obtention du grade de docteur en Informatique **de l'**Université Côte d'Azur

Dirigée par : David COUDERT, Directeur de Recherche, Inria, Sophia-Antipolis
Co-encadrée par : Nicolas NISSE, Chargé de Recherche Inria, Sophia-Antipolis
Soutenue le : 25 Novembre 2021

Devant le jury, composé de :
Jean-Charles RÉGIN, Professeur, Université Côte d'Azur,
David ILCINKAS, Chargé de Recherche CNRS, LaBRI, Université de Bordeaux
Laurent VIENNOT, Directeur de Recherche, Inria, IRIF, Paris
Christian LAFOREST, Professeur, Université Clermont Auvergne
Mohammed Amine AIT OUAHMED, Ingénieur R&D, Instant-System

CALCUL PRATIQUE DE CHEMINS SIMPLES AVEC DES CONTRAINTES DE LONGUEUR ET DE DIVERSITÉ DANS DES RÉSEAUX COMPLEXES ET MULTIMODAUX

Practical computation of simple paths with length and diversity constraints in complex and multimodal networks

Ali AL ZOOBI

Jury :

Président du jury

Jean-Charles RÉGIN, Professeur, Université Côte d'Azur,

Rapporteurs

David ILCINKAS, Chargé de Recherche CNRS, LaBRI, Université de Bordeaux
Laurent VIENNOT, Directeur de Recherche, Inria, IRIF, Paris

Examinateurs

Christian LAFOREST, Professeur, Université Clermont Auvergne
Mohammed Amine AIT OUAHMED, Ingénieur R&D, Instant-System

Université Côte d'Azur

Ali AL ZOOBI

Calcul pratique de chemins simples avec des contraintes de longueur et de diversité dans des réseaux complexes et multimodaux

Ce document a été préparé avec LaTeX2e et la classe `these-ISSS` version v. 2.10.

Impression : output.tex – 9/2/2022 – 11:06

Révision pour la classe : these-ISSS.cls,v 2.10 2020/06/24 14:16:37 mpelleau

À ma précieuse maman, merci de m'avoir laissé partir

Calcul pratique de chemins simples avec des contraintes de longueur et de diversité dans des réseaux complexes et multimodaux

Résumé

Le problème du plus court chemin est l'un des problèmes les plus étudiés en théorie des graphes et en recherche opérationnelle. Une généralisation classique de ce problème est le problème de trouver k plus courts chemins simples (kSSP). C'est-à-dire, le problème de trouver le plus court, le deuxième plus court, etc. jusqu'au k-ième plus court chemin simple d'une source à une destination dans un graphe orienté pondéré. Yen (1971) a proposé l'algorithme avec la meilleure complexité théorique connue pour résoudre le kSSP dans un graphe orienté pondéré à n sommets et m arcs, avec une complexité en $O(kn(m + n \log n))$. Depuis, le problème a été largement étudié du point de vue de l'ingénierie algorithmique.

Dans cette thèse, nous étudions également le problème kSSP sous cet angle, c'est-à-dire que nous proposons des algorithmes exacts offrant de meilleures performances en pratique que l'état de l'art, en termes de temps d'exécution, de consommation mémoire ou offrant de meilleurs compromis espace-temps. Nous montrons aussi comment étendre nos algorithmes au cas des graphes avec des poids arbitraires sans cycles négatifs.

De plus, nous étudions le problème de trouver k plus courts chemins simples qui sont mutuellement dissimilaires. Plus précisément, nous étudions la complexité du problème en fonction de quatre mesures de similarité différentes, et nous montrons dans quels cas le problème est NP-Complet ou peut être résolu en temps polynomial.

Enfin, nous montrons comment adapter le problème kSSP à un modèle de réseau de transport public multimodal. Nous adaptons certains de nos algorithmes pour le kSSP au problème de trouver, dans un réseau de transport public multimodal, les k itinéraires d'une station source à une station destination arrivant au plus tôt.

Mots-clés : Théorie des graphes, plus court chemin, Ingénierie algorithmique.

Practical computation of simple paths with length and diversity constraints in complex and multimodal networks

Abstract

The shortest path problem is one of the most studied problem in graph theory and operations research. A classic generalization of this problem is the problem of finding k shortest simple paths (kSSP for short). That is, the problem of finding a shortest, a second-shortest, etc. until a k-th shortest simple path from a source to a destination in a directed weighted graph. Yen (1971) proposed the state-of-the-art kSSP algorithm, with theoretical time complexity in $O(kn(m + n \log n))$, where n is the number of vertices and m is the number of arcs of the input digraph. Since then, the problem has been widely studied from an algorithmic engineering perspective, that is designing exact algorithms offering better performances in practice.

In this thesis, we study the kSSP problem from an algorithm engineering perspective. More precisely, we design new kSSP algorithms allowing to outperform the state-of-the-art algorithms in terms of running time, memory consumption, or offering a better space-time trade-off. We also show how to apply our algorithms in graphs with arbitrary arc weights without negative cycles.

Then, we study the problem of finding paths respecting dissimilarity constraints. Precisely, we study the complexity of the problem according to four different similarity measures, and we show in which cases the problem is NP-Complete or polynomial time solvable.

Finally, we show how to adapt the kSSP problem to a multimodal public transportation network model, i.e., combining metro, tram, buses, and walk. Precisely, we design some kSSP algorithms to solve a related problem, which is, the problem of finding k earliest arrival journeys from a source station to a destination station in a public multimodal transportation network.

Keywords: Graph theory, shortest path, algorithm engineering.

Remerciements

Je remercie d'abord les membres de mon jury de thèse. Merci à David ILCINKAS et Laurent VIENNOT pour avoir eu la gentillesse de rapporter ma thèse. Je remercie également Jean-Charles RÉGIN, Christian LAFOREST et Mohammed Amine AIT OUAHMED d'accepter d'être membre de mon jury de thèse.

Un très grand merci à mon encadrant, David COUDERT, d'avoir accepté de m'encadrer pour ces trois ans, d'avoir écouté mes idées avec un son esprit ouvert, sa patience en lisant mon code pourri et en corrigeant mes bugs débiles, et les efforts de qualité qu'il a mis afin que cette thèse se déroule bien malgré les conditions particulières. Merci aussi à mon co-encadrant, Nicolas NISSE de m'avoir appris à écrire (formellement), ce qui est la chose la plus importante que j'ai apprise au cours de ces trois ans. Je tiens bien sûr à remercier tous les autres membres de l'équipe COATI et apparentés : Patricia (sa majesté), à mes co-bureau Nataza et Hicham, à Fréderic HAVET (un jour je serais titulaire et je te battrai au ping-pong), à Stéphane PÉRENNES, le génie discret, pour ses idées intéressantes qui m'ont aidé à finir mes preuves, pour Luc HOGIE pour son aide technique indispensable pour mes expérimentations, merci à Michelle, Joanna, Fréderic (Minus), Emilio, Ha, Foivos, Francesco, Arthur, Thomas et tous les COATIs. Un grand merci pour Arthur FINKELSTEIN pour sa patience pendant notre projet "dure et long". Finalement, je remercie mes anciens profs Ahmad AL ZOGHBI et Bilal SAÏD pour leurs conseils scientifiques et humains pendant ces trois ans.

Des remerciements à ma famille d'être là malgré leurs conditions difficiles. Merci encore une fois à ma mère pour son amour inconditionnel et de m'avoir laissé partir, à mon père d'avoir partagé ses morceaux de poésie avec moi, et pour ses précieux conseils. Je remercie spécialement ma tante Alima et ma nièce Salam (mes amies en même temps) d'avoir la curiosité de découvrir ce que je fais pendant ma thèse, je ne peux pas décrire ce que je ressens quand je vois leurs yeux fiers.

Finalement, je remercie mes amis "originaux" : Abbass, Nour, Hussein et Ali. Et tous les amis que j'ai rencontrés pendant ces trois ans : Nader, Hassan, Yahya, Clara, Julia, Salwa, Rima, Riam et Arij.

Table of contents

Introduction

Travelling from one place to another is a primitive action. Animals move to reach food resources, migrate to better circumstances *etc*. Humans travel in a transportation network to discover new places, to go to work, *etc*. Also, materials are sent from a source to a destination via the post network. Even virtual object like data may move from a device to another via Internet.

Let $G = (V, A)$ be a graph, i.e, a set of vertices connected by a set of edges. Most of the networks can be seen as a graph. For instance, a road network can be seen as a graph where crossroads are represented by vertices and roads by edges, the Internet network can be modeled as a graph where vertices represent routers and each edge represent a link between two routers, *etc*. Analogously to real life networks, a path is a sequence of vertices that one can take to go from one vertex to another in a graph. The length of a path indicates how long it takes (how much it costs, *etc*.) to traverse such path. Finally, a shortest path from one vertex s to another vertex t is a path with minimum length among all the paths starting from s and ending at t.

A classical problem in graph theory, is the problem of finding shortest paths from a source to a destination in a graph. Precisely, today's challenge is to design performant algorithms, in terms of running time, memory consumption *etc*. allowing to answer shortest path queries from a source to a destination. This classical problem has various interesting generalization, like the one-to-all (finding a path from one vertex to all the others), all-to-one, all-to-all (known as All Pairs Shortest Paths, APSP, problem), the multi-criteria shortest path (finding a short, fast and cheap path for example), shortest path in dynamic graphs and the k shortest paths problem.

The k shortest paths problem is one of the most fundamental shortest path problems. It asks to find a shortest path, a second-shortest path, *etc*. until a $k - th$ shortest path from a source to the destination in a graph. Depending on the context, some constraints may be added to the output paths. For instance, the output paths may be required to be simple, i.e, do not visit a vertex more than once, mutually dissimilar, *etc*.

In this thesis, we first study the k shortest simple paths problem, where we propose new algorithms outperforming the state-of-the-art in terms of running time, memory consumption or offering space-time trade-offs. Then, we add constraints of dissimilarity between paths, where we study the complexity of the problem regarding various similarity measures, and we show in which case the problem is NP-complete or can be solved in polynomial time. Finally, we show how to adapt some of the k shortest simple paths algorithms to extract k-best journeys in a public transit networks using a well-known public transit model.

1.1 Overview

Shortest path. The shortest path problem is the problem of finding a shortest path from one vertex to another in a digraph. This problem is extensively studied in the literature, and various shortest path algorithms are proposed. Let $D = (V, A)$ be a directed weighted graph with length function $\ell : A \rightarrow \mathbb{R}^+$. In 1959, Dijkstra proposed a polynomial time algorithm allowing to find a shortest path from a given vertex to all other vertices in a positively weighted directed graph. With the help of advanced data structures (like Fibonacci heap), Dijkstra's algorithm can

reach a complexity bound in $O(m + n \log n)$ where n is the number of vertices and m is the number of arcs. Meanwhile, another algorithm known as Bellman-Ford)Moore was discovered with a time complexity of $O(nm)$, Bellman-Ford-Moore's algorithm allows to correctly compute a shortest path from a vertex to all other vertices in a directed weighted graph without negative cycles (defined below) [106].

Today, with the help of preprocessing routine, one can answer shortest paths queries in milliseconds or less even at continental scale (Section 1.3.2). In addition, several trade-offs between the computation time of the preprocessing routine, the query time and the space requirements are proposed [21].

k shortest (simple) paths. The k shortest paths problem (kSP) is the problem of finding a shortest path from a source vertex (s) to a destination vertex (t) in a graph, a second-shortest s-t path, *etc*. until a k^{th} shortest path (formally defined in Section 1.4). Henceforth, several versions of this problem are defined and studied in the literature.

The original version of the k shortest paths problem can be solved by Eppstein's algorithm [50] in $O(m + n \log n + k)$. That is, only $O(k)$ plus the time needed to compute a single shortest path from a source to a destination. If we want to extract k shortest paths (and not an implicit description), we need an extra $O(n)$ time per path, i.e, Eppstein's algorithm extracts, explicitly, k shortest paths in $O(m + n \log n + kn)$ time complexity.

The k shortest simple paths (kSSP for short) problem is a special version of the kSP where output paths are required to be simple (paths that do not visit a vertex more than once). The first algorithm solving the kSSP problem has been proposed by Yen [127] in 1971 and has a worst-case time complexity in $O(kn(m + n \log n))$, that is the best theoretical complexity bound to solve the kSSP so far. Since then, several solutions have been proposed to improve the efficiency of the algorithm in practice [67, 79, 111, 70, 59, 54, 83, 82]; they all feature the same worst-case time complexity as Yen's algorithm, that is, $O(kn(m + n \log n))$ but are much faster in practice.

k shortest dissimilar simple paths. Two distinct paths may differ by a single arc, share half of their edges, or can be completely different (disjoint). In order to measure how similar / different they are, the path-similarity is studied. The similarity between two paths can be expressed according to the proportion of arcs they share. A path-similarity measure is a function between zero and one indicating how much two given paths are similar. Various path-similarity measures are studied in the literature. For instance, a similarity measure can be the portion of the length of the intersection of the two given paths over the length of their union / the length of the longest / the length of the shortest path among them, *etc*. So, two paths are called "dissimilar" if their similarity, w.r.t. a similarity measure, does not exceed a certain threshold.

The k shortest dissimilar (simple) paths problem is a special version of the k shortest (simple) paths problem where output paths are required to be mutually dissimilar. This problem is proved to be NP-complete for several similarity measures, and various exact algorithms (path enumeration, pseudo-polynomial algorithm) and heuristics (like the additive penalty heuristic) were proposed to solve it [1, 3, 32, 33].

Journey planning in public transit networks In contrast to road networks that can be modeled using graphs with edges that can be traversed anytime, public transit networks are more complex, as they are time dependent and contain various transportation modes.

A *transit network*, also referred to as a (multimodal) transportation network or a *timetable*, is a set of stops (such as bus stops or train stations), a set of routes (such as bus, tramway, ferries, metro or train lines), and a set of trips. Trips correspond to individual vehicles that visit the stops along a certain route at a specific time of the day. Trips can be further subdivided into sequences of elementary connections, each given as a pair of (origin/destination) stops and (departure/arrival) times between which the vehicle travels without stopping. In addition, footpaths model walking transfers between nearby stops. A *journey* is a sequence of trips and footpaths one can take within a transit network.

Journeys planning in (schedule-based) public transit networks and accelerating queries for efficient journey planning is a long-standing problem [21]. In the last decade, many algorithms have been developed not only to answer efficiently basic queries like a quickest or an earliest arrival journey, but also to optimize additional criteria like the number of transfers, the cost of the trip, *etc.* or even to offer Pareto optimal solutions combining several criteria [21, 43, 46].

1.2 Model

To model a problem, to solve a riddle or to express ideas, we begin spontaneously to draw circles and connect them by lines as a graph. Graphs offer a formal and expressive structure enabling to model networks. In this section, we present few examples of how a graph can be used to model networks, then we formally present several graph related concepts that are used in this chapter.

Travelling in a map is usually done via routes and crossroads. So, from a routing perspective, a map can be seen as a road network, i.e, is restricted to crossroads and routes. Also, a straightforward model of a road network is a graph (a set of vertices and edges) model. As shown in Figure 1.3, a road network can be seen as a graph where vertices represent crossroads, with an edge between two vertices if there is a route linking the two corresponding crossroads. Indeed, this basic model can be further improved using directed weighted graphs where edges are directed (called arcs), and each edge is labelled with a value corresponding to its length, travel-time, cost, *etc*.

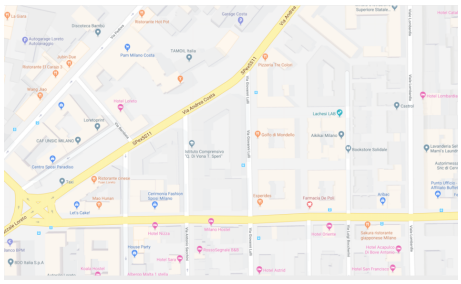


Figure 1.1 – A road network

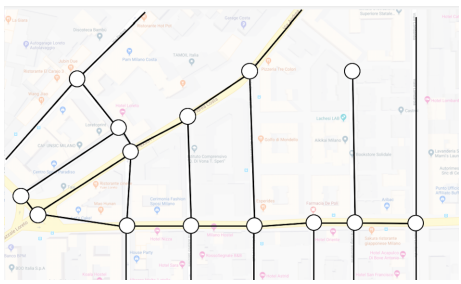


Figure 1.2 – A graph model

Figure 1.3 – A graph model of a road network

More generally, most of the well-known networks can be seen as a graph. For instance, the Internet network can be seen as a graph where vertices are the routers and edges are the links between them. A social network can be seen as a graph where vertices represent the people accounts

and the edges (arcs) represent the friendship (follow) relationship between the accounts. In a scheduling context, the vertices of a graph can represent the tasks and the arcs represents precedence constraints between them, *etc*.

1.2.1 Definitions and Notation

Let $D = (V, A)$ be a directed graph (digraph for short) with vertex set V and arc set $A \subseteq V \times V$. Let $n = |V|$ be the number of vertices and $m = |A|$ be the number of arcs of D. For the sake of simplicity, we restrict ourselves to simple graphs, i.e, self loops and multi-arcs are not allowed. For each arc $a = uv$, the vertex u is called the tail of a and v is called the head of a. Let $\ell_D : A \to \mathbb{R}^+$ be a length function over the arcs. For every $u, v \in V$, a *(directed) path* from u to v in D is a sequence $P = (u = v_0, v_1, \cdots, v_r = v)$ of vertices with $v_i v_{i+1} \in A$ for all $0 \leq i < r$. Note that vertices may be repeated, i.e., paths are not necessarily simple. A path is *simple* if, moreover, $v_i \neq v_j$ for all $0 \leq i < j \leq r$. The *length* of the path P equals $\ell(P) = \sum_{0 \leq i < r} \ell(v_i, v_{i+1})$.

Let s, t be two vertices of V. An *s-t path* $P = (v_0 = s, v_1, \cdots, v_{r-1}, v_r = t)$ is a path starting with s and ending with t. A *shortest path* from s to t is a path such that no other path from s to t has a strictly smaller length. The *distance* $d(s, t)$ between two vertices $s, t \in V$ is the length of a shortest s-t path. Note that there might be several shortest s-t paths, all of them having the same length which is equal to the distance $d(s, t)$.

Given a vertex $v \in V$, $N^+(v) = \{w \in V \mid vw \in A\}$ denotes the out-neighbors of v in D. Given two paths $P = (v_0, \cdots, v_r)$ and $Q = (w_0, \cdots, w_p)$ and an edge $v_r w_0 \in A$, we denote by $P.Q$ the v_0-w_p path obtained by the concatenation of P and Q. i.e, $P.Q = (v_0, \cdots, v_r, w_0, \cdots, w_p) = (v_0, \cdots, v_r, Q) = (P, w_1, \cdots, w_p)$.

Let $t \in V$. An *in-branching* T rooted at t is any sub-digraph of D that induces a (not necessarily spanning) tree containing t, such that every $u \in V(T) \setminus \{t\}$ has exactly one out-neighbor (that is, all paths go toward t) and t has no out-neighbors. An in-branching T is called a *shortest path (SP) in-branching* rooted at t if, for every $u \in V(T)$, the length of the (unique) u-t path P^T_{ut} in T equals $d_D(u, t)$. Note that an SP in-branching is sometimes called *reversed shortest path tree*. Similarly, A *shortest path (SP) out-branching* T rooted at s is any sub-digraph of D inducing a tree containing s, such that every $u \in V(T) \setminus \{s\}$ has exactly one in-neighbor and the length of the (unique) s-u path P^T_{su} in T equals $d_D(s, u)$.

A digraph is called arbitrarily weighted if it has negative weighted arcs, i.e, $\ell : A \to \mathbb{R}$. A *cycle* $C = (v_0, v_1, \cdots, v_{r-1}, v_0)$ is a path starting and ending with the same vertex. So, the *length* of a cycle follows the definition of the length of a path. Moreover, if every cycle in a digraph D has a non-negative length, we call D a *digraph with no negative cycle*. For each pair of vertices s, t in V, we define a shortest s-t path in an arbitrarily weighted digraph as a shortest simple path from s to t (if any).

1.3 The shortest path problem

The shortest path problem is probably one of the most studied problems in operational research [21, 91], regarding its various applications in networking, transportation, scheduling, bioinformatics *etc*.

In this section, we start by giving a formal definition of the shortest path problem with its various variants, then we survey the fundamental algorithms solving this problem with an overview of their advantages / drawbacks.

1.3.1 One-to-all

Various shortest path algorithms are proposed. Let $D = (V, A, \ell)$ be a directed positively weighted graph. The one-to-all shortest path problem asks to find a shortest path from a source vertex s in V to all other vertices. As long as D has no negative weighted cycles, the one-to-all problem is the problem of finding an SP out-branching of D rooted at s.

In 1959, Dijkstra proposed a polynomial time algorithm allowing to solve the one-to-all shortest path problem.

Let $a = (u, v)$ be an arc of A, and let $d[u]$ ($d[v]$) be an upper bound on the distance from s to u (v). *Relaxing* the arc a consists of updating $d[v]$ with respect to the path via u, that is, applying the instruction :

$$d[v] \leftarrow min(d[v], d[u] + \ell(u, v))$$

Dijkstra's algorithm starts by labeling each vertex u in V with an upper bound on the distances $d[u]$ from s to u (that is zero for s and ∞ for the other vertices), then it processes the vertices in a non-decreasing order of their labels.

Processing a vertex u in Dijkstra's algorithm consists of adding the distance of the vertex $d[u]$ to the output (as it is proven that $d[u]$ is a valid distance from s to u), and relaxing its neighbors ($v \in N^+(u)$) via u. Dijkstra's algorithm stops once all the vertices in V are processed. If only the distance to a given vertex t is needed (one-to-one), Dijkstra's algorithm stops once the vertex t is processed.

Dijkstra's algorithm needs a data structure to store the unprocessed vertices in a non-decreasing order of their labels (the upper bound of their distance from s), if this data structure is a Fibonacci heap, the complexity of Dijkstra's algorithm is bounded in $O(m+n \log n)$. The correctness of Dijkstra's follows from the non-decreasing order of vertex extractions, and it is guaranteed only for positively weighted digraphs.

Another classical algorithm known as Bellman-Ford's (or Bellman-Ford-Moore's) algorithm was discovered simultanously by Bellman, Ford, Moore and Shimbel. Bellman Ford Moore's algorithm relaxes $n-1$ times all the arcs of A. The proof of its correctness follows from the fact that any s-t simple path (in particular a shortest one) has at most $n-1$ vertices. The time complexity of Bellman Ford Moore's algorithm is in $O(nm)$, Bellman Ford Moore's algorithm allows to correctly compute a shortest path from a vertex to the remaining vertices in a directed weighted graph without negative cycles [106]. Since then, several algorithms were proposed to speed up / improve Bellman Ford Moore's algorithm in practice [93, 106, 42, 40, 73, 115, 61]. A common idea of these improvements is to relax the arc in a specified order (breadth first search BFS fashion, for instance), and avoid relaxing an arc once relaxing it won't improve the current distance value of its head.

Note that both Dijkstra's and Bellman Ford Moore's algorithms can be adapted to solve the all-to-one problem, that is, finding a shortest path from each vertex of the digraph to a destination vertex. This can be done by reversing the orientation of each arc of the digraph.

Finding a shortest simple path in arbitrarily weighted graphs (with negative cycles) is NP-complete, as the Longest Path problem can be reduced to it. However, several advanced integer linear programming (ILP) models were proposed to solve this problem with ac-

ceptable performances only on small networks (one second for a random graph of 100 vertices [85]) [45, 23, 103, 115, 42, 27].

1.3.2 One-to-one query acceleration

In most of the shortest path applications, we only need source-destination paths, i.e, a shortest path from a source vertex s to a destination vertex t. Plethora of methods were designed in order to accelerate the computation of such path. In this section, we briefly present the intuitions of some of these methods.

Most of these speed-up techniques split the path-finding process into two routines : A preprocessing routine where a data structure summarizing information about the graph is constructed, and a query routine where, with the help of the already computed data structure, a one-to-one shortest path is computed. Even though a preprocessing routine could drastically reduce the shortest path query computation, one may have to repeat the whole preprocessing computation if changes occur to the input graph (typically for dynamic and time-dependent graphs) [21].

Each of these query accelerating methods offers a different tradeoff between the preprocessing time, the memory consumption and the query time.

Goal-directed methods (A*, ALT, *etc*.) With a fast preprocessing routine, the goal-directed methods retrieve a lower bound from each vertex to the destination, which can be done via additional information like the euclidean distance (for A*) [63], or by picking, heuristically, a set of vertices called landmarks (8 to 32 landmarks are sufficient for the road network Western US with up to 2 millions vertices), and computing distances from (and to) each vertex to (and from) each one of the landmarks (for the ALT) [63]. The goal-directed methods apply almost the same routine as Dijkstra's algorithm during the query processing, except, they use the accessible lower bound of each vertex as an indicator on how far it is from the destination. So, they prioritize vertices who are close to the destination in order to scan less vertices and to reach, as soon as possible, the destination vertex. Another optimization approach leading to scan less vertices, known as bidirectionnal Dijkstra, consists of running two simultaneous searches : one forward from the source s, and one backward from the target t, and stopping when the two searches meet. This approach can be integrated with the A* or ALT methods in order to achieve even better query time.

Contraction Hierarchies (CH) The intuition of the Contraction Hierarchies (CH) method is that some vertices are more important than others in the process of finding a shortest path, so their distance from / to the other vertices must be computed and accessible. For instance, in road networks, highway junctions are "more important" than a junction leading into a dead end. Roughly, a vertex is more important than another if there are more shortest paths traversing it than the other.

In the preprocessing routine, Contraction Hierarchies (CH) methods select a set of "important" vertices from the graph and computes "shortcuts" between them, that is, the exact distance. This can be done by performing iterative vertex contractions, where contracting a vertex consists of temporarily removing it from the graph and adding "shortcut" arcs between its in-neighbors to its out-neighbors with a corresponding length. So, one can sort the vertices in an increasing order of their "importance" and iteratively contract them until achieving a "small" reduced graph of only important vertices. The query phase of CH consists of a bidirectional exploration (from the source and the target) of the graph plus the shortcuts produced during the contraction and processing only

the more important vertices until the two searches meet, then the algorithm concatenates paths from both explorations and outputs a shortest path [47].

Hub Labelling methods (HL) A Hub Labeling method consists in precomputing, for each vertex u, a Hub set $\mathcal{L}(u)$ containing some vertices along with their distances from (and to) u. Those sets are computed such that, for each pair (s, t) of vertices, there is a vertex $u \in \mathcal{L}(s) \cap \mathcal{L}(t)$ that is on a shortest path from s to t. Therefore, a query consists of picking the vertex x in $\mathcal{L}(s) \cap \mathcal{L}(t)$ minimizing $d(s, x) + d(x, t)$.

The query running time of a HL method is extremely fast (see below), as it is bounded by the size of the label set of a vertex. However, the preprocessing routing consists of picking the appropriate set of labels for each vertex, computing and storing the distance from (and to) each vertex to (and from) each one of its label.

In order to achieve satisfying performance in practice, the size of a hub-set should be small (as its elements must be scanned during each query). However, the problem of minimizing the size of the hubs (on average or minimizing the size of the bigger Hub) is NP-complete [122]. So, the hub-sets are usually computed using heuristics based on experimental and theoretical observations of the topology and the metrics of the networks [2, 80, 5].

Even though the required average label size can be $\Theta(|V|)$ in general [62], it can be significantly smaller for some graph classes. For instance, an empirical observation for road networks, is the existence of small hub sets, and consequently efficient shortest path query algorithms. Several theoretical explanations for this phenomenon were proposed by studying structural parameter as the highway dimension [2] or the skeleton dimension of the graph [80]. The study of these parameters also provides an efficient way of computing labels and helps in finding bounds on the size of a hub set.

Note that, other query-accelerating methods are proposed in the literature. They either offer various preprocessing-query-space tradeoffs, like Reach, Lookup (PHAST), Arc Flags, *etc*, or have a context dependent motivation, like Customizable Contraction Hierarchies (CCH), Contraction Hierarchy, Arc flagS and highway nodE (CHASE), Pruned Landmark Labelling PLL, *etc*. Several shortest path survey explain these methods in details, see [21, 91].

Performance in practice In order to evaluate the main query-accelerating methods, a detailed experiment has been conducted by Bast et al. [21]. As shown in Table 1.1, for the Western Europe road network, with up to 18.10^6 vertices and 42.10^6 arcs, Dijkstra's algorithm running time is around 2 seconds on average, while one of the ALT-like methods achieved an average query time of 40 (ms) for 4 (min) preprocessing time (note that its memory consumption is not explicitly written in the paper, but it is not supposed to be far bigger than the memory consumption of Dijkstra's algorithm). Moreover, CH query time achieved 0.1 (μs) for 5 (min) preprocessing time. On the extreme case, the hub-labelling methods could achieve a query running time of 0.56 (μs) for up to 37 (mns) of preprocessing time (with about 19 GiB of memory consumption).

To sum up, one can compute a shortest path from one vertex to another in milliseconds or less, even at continental scale graphs. Today's researches aims at finding better trade-offs between preprocessing effort, space requirements and query time, by designing special variants of these methods (like customization for CH [47] and skeleton dimension for HL [80]).

Algorithm	Preprocessing time	Query time	Memory consumption (GiB)
Dijkstra	0	2 (s)	0.4
ALT	4 mns	40 (ms)	-
CH	5 mns	0.1 (ms)	0.4
HL	37 (mns)	0.56 (μs)	18.8

Table 1.1 – Practical performance on average of some shortest path algorithms on Western Europe network ($n \approx 18.10^6$ and $m \approx 42.10^6$) [21]

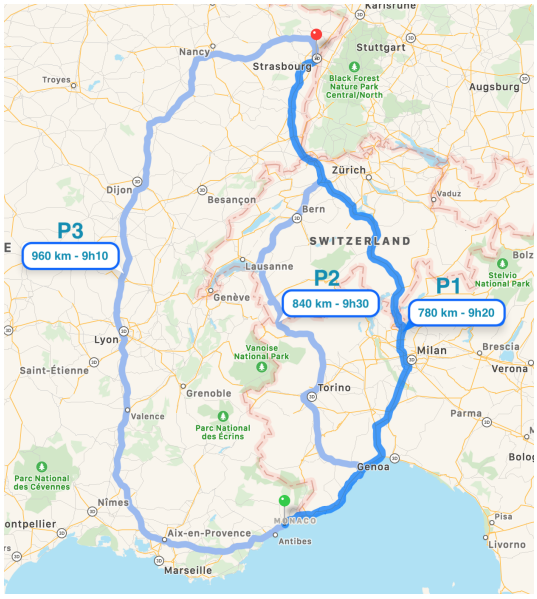


Figure 1.4 – Three alternative paths from Nice to Strasbourg

1.3.3 Multicriteria shortest paths

In various scenarios, the choice of a best path does not depend only on a single criterion. In a transportation context, for example, one may like to optimize the travelling distance, the travelling time, the tolls payments, *etc.* (Figure 1.4). In this case, the graph model could be updated to be a multi-label graph. That is, several length functions are associated to the arcs. Formally, we define $\ell_1, \ell_2, \cdots, \ell_\alpha$ length functions where $\ell_i : A \to \mathbb{R}^+$ for $0 \leq i \leq \alpha$ (we suppose there are α criteria). Following the single-criteria path length definition, the length of the path with respect to the i^{th} criteria is the sum of the length of its arcs $\ell_i(P) = \sum_{0 \leq j \leq r} \ell_i(v_j, v_{j+1})$, for $P = (v_0, \cdots, v_r)$.

The Resource Constraint Shortest Path (RCSP) problem takes as input a directed multi-label graph with length functions $\ell_1, \cdots, \ell_\alpha$, with constants $C_1, \cdots, C_\alpha$, two vertices s and t, and asks whether there is an s-t path P s.t. $\ell_i(P) \leq C_i$ for $0 \leq i \leq \alpha$.

The Resource Constraint Shortest Path problem is known to be NP-complete [77]. However, there exists a pseudo-polynomial time algorithm solving it. Moreover, various linear programming models, heuristics, and approximations were studied. See [69, 72] for details.

Note that, the problem of finding a Pareto optimal solution considering several criteria, i.e, constraints is also studied and various methods are proposed to compute / approximate the set of Pareto-optimal solutions [69].

1.4 The k shortest (simple) paths problem

1.4.1 Motivations

In various shortest path applications, proposing a single path may not be enough, as it might be blocked, not satisfying some properties or not meeting the desire of diversity. This may be illustrated in a transportation context, where a user likes to choose the itinerary he likes among those proposed by the GPS. For this, the k shortest paths problem (kSP) is defined and studied. That is, the problem of finding a shortest path from a source s to a destination t, a second-shortest s-t path, *etc*. until a k^{th} shortest path. Several versions of this problem are defined and studied in the literature.

Apart from its diverse applications in transportation routing [67, 76, 125], the k shortest (simple) paths is also used as a building block in other important problems. For instance, it is successfully used for the (0,1)-knapsack problem [126], natural language processing and speech recognition [24, 30, 31, 36, 108, 114]. And the Viterbi decoding technique for Markov model [96]. Also, it is used in different areas like biological sequence alignment [26, 94, 121], gene regulation networks [112], reconstruction of metabolic pathways [17], telecommunications networks [18, 117] and scheduling [48, 71], as well as in subroutines for several operational research problems [74, 58, 39, 29]. See Eppstein's recent comprehensive survey on k-best enumeration for more applications [51].

1.4.2 The k shortest paths kSP problem

In this section, we define formally the k shortest paths problem, then we briefly describe two main algorithms solving it.

Definition The k shortest paths problem (kSP) takes as input a directed weighted graph with a positive length function ℓ, two vertices s and t, and asks to find a set S of k pairwise different paths from s to t such that no path outside S has a length strictly less than a path in S. Formally, $S = \{P_1, P_2, \cdots, P_k\}$ such that, for $1 \leq i \leq k$, P_i is an s-t path, $\ell(P_i) \leq \ell(P)$ for every s-t path $P \notin S$ and $P_i \neq P_j$ for $1 \leq i < j \leq k$

Eppstein's algorithm The original version of the k shortest paths can be solved by Eppstein's algorithm [50] in $O(m + n \log n + k)$ time complexity. That is, only $O(k)$ plus the time needed to compute a single shortest path from a source to the remaining vertices. Note that, Eppstein's algorithm (1997) output only a description of the k shortest paths, i.e, a data structure allowing to extract k shortest paths in linear time each. In other words, one can use Eppstein's algorithm to extract k shortest paths in $O(m + n \log n + kn)$.

Eppstein's algorithm starts by computing a shortest path in-branching T rooted at t, i.e, an all-to-one shortest path query, together with a shortest s-t path P (the one extracted from T). Then, each arc tailing at P (each arc with a tail in P) and not in T is called a *sidetrack*. Each sidetrack sequence $(a_1, a_2, \cdots, a_\alpha)$ encodes the path starting at s, following P until the tail of a_1, taking

a_1, then following T from the head of a_1 until reaching the tail of a_2, *etc*. until the head of a_α, then following T until reaching t. Each sidetrack $a = (u, v)$ is associated with a corresponding *residual length* δ, that is, the length of deviating from T via a. Formally, $\delta(a) = \ell(u, v) + \ell(P^T_{vt}) - \ell(P^T_{ut})$.

Clearly, a second-shortest path is a path encoded by a sidetrack $a_1 = (u_1, v_1)$ with minimum residual length. Similarly, a third-shortest path is the shorter path among the path encoded by a sidetrack a_2 tailing at P (different from a_1), and a path encoded by a_1, a'_2 where a'_2 is a sidetrack with minimum residual length among the extensions of a_1, i.e, the sidetracks tailing at $P^T_{v_1 t}$.

In [50], Eppstein designed a heap of sidetrack sequences, where each element represents a sequence of sidetracks encoding an s-t path with the length of the corresponding path as a key. Eppstein shows that this designed heap allows to extract the k minimum element only in $O(k)$ time (for a constant k). Therefore, one can extract k sequences of sidetracks with minimum associative residual length in $O(k)$ time. That is, a data structure allowing to extract, in linear time, each of the k shortest s-t paths.

Eppstein's algorithm takes $O(m + n \log n)$ time to compute a shortest path out-branching rooted at t, and $O(n + k)$ time to build and extract the elements from the heap. That is, $O(m + n \log n + k)$ time to find the k shortest paths. An additional time of $O(kn)$ is needed to extract the paths themselves.

Few years later, Jiménez and Marzal proposed a lazy version of Eppstein's algorithm [75] allowing to speed it up in practice while preserving the same worst case time complexity.

Hub labelling for the kSP Recently, Akiba et al. [4] show how to adapt one of the hub labelling methods to answer k shortest paths query from a source to a destination. Similarly to the hub labelling methods, the algorithm of Akiba et al. computes and stores, for each vertex u and its labels $L(u)$, data related to the k shortest paths from u to each vertex in $L(u)$ and vice versa.

Precisely, Akiba et al. algorithm computes with a modified BFS or Dijkstra (called top-k BFS / top-k Dijkstra), for each vertex u, a *Distance Label* $L(u)$, that is, for each v in $L(u)$, it computes and stores a set of α shortest u-v paths for $1 \leq \alpha \leq k$. Then, it computes a *Loop Label* $C(v)$, constituting a sequence of k cycles containing v. Therefore, an *Index* of the algorithm is a pair $I = (L, C)$.

A kSP query algorithm to find k shortest s-t paths computes the k smallest concatenations of labels of s, self loops (cycles), and labels of t. Precisely, let $\Delta(I, s, t) = \{\delta_{sv} + \delta_{vv} + \delta_{vt} \mid (v, \delta_{sv}) \in L(s), \delta_{vv} \in C(v), (v, \delta_{vt}) \in L(t)\}$. That is, we first move from s to v, then loop back to v several times, and finally move from v to t. Then, a set of k shortest s-t paths is a set of k smallest elements in $\Delta(I, s, t)$.

Akiba et al. also show by conducting experiments on large graphs (complex networks) of millions of vertices and tens of millions of edges that their algorithm, with a reasonable preprocessing time (less than one hour), can answer kSP queries in few microseconds, that is, up to six orders of magnitude faster than Eppstein's algorithm.

1.4.3 The k shortest simple paths problem (kSSP)

In various applications (typically in transportation contexts), we are rather interested in *simple* paths (paths that does not visit a vertex more than once). The k shortest simple paths problem is a special version of the kSP where output paths are required to be simple. This variant of the problem was first introduced by Clark et al. [37] (1963) while the first algorithm solving it has been proposed by Yen [127] in 1971 and has a worst-case time complexity in $O(kn(m + n \log n))$,

that is the best theoretical complexity bound to solve the kSSP so far. Recently, Williams and Williams [118] show that a subcubic kSSP algorithm would also result in a subcubic algorithm for APSP (All Pairs Shortest Paths), which seems unlikely at the moment.

Baseline solution. A baseline solution to find k shortest simple paths is to adapt, naively, Eppstein's algorithm to extract only simple paths, i.e, run Eppstein's algorithm on the input digraph to extract $k' > k$ shortest paths, then select the simple paths among them.

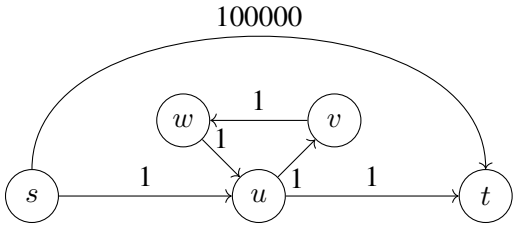


Figure 1.5 – A second-shortest s-t path is $(suvwut)$, a third-shortest path is $(suvwuvwut)$ *etc*.

Unfortunately, this approach is unpractical. First, this algorithm is not polynomial, as shown in Figure 1.5, the number of paths that are shorter than the k shortest simple paths can be arbitrarily large, i.e, not bounded by the size of the input. Second, such scenario is very usual in practice (in transportation context, a roundabout can form such cycle). So, apart from the fact that this algorithm is not polynomial, it won't be a practical heuristic as its running time will be too long. Moreover, this method is too memory consuming [104] as using Eppstein's algorithm require storing in the memory a description of all the k' paths.

Yen's algorithm In 1971, Yen proposed the first algorithm designed to solve the k shortest simple path problem. The intuition of Yen's algorithm is the following, it starts by computing a shortest s-t path. Then, it computes "shortest simple detours" (described below) of the shortest path in order to pick a smallest detour among them, that is, a second-shortest path. Then, it applies almost the same routine on the chosen second-shortest path in order to find a third-shortest path, *etc*. until finding a k^{th} shortest path. Precisely, Yen's algorithm (and most of the Yen-Based algorithms) apply the following routine :

1. Compute a shortest s-t path and add it to a set $Candidates$
2. For $i = 1, \cdots, k$:
3. Extract a path P with minimum length from $Candidates$ and add it to the output
4. Compute the shortest simple detours of P and add them to $Candidates$

A shortest (simple) detour of an s-t path P on a vertex v is a shortest (simple) path deviating from P at v, i.e, a path $Q.Q'$ where Q is the subpath starting with s and ending with v in P and Q' is a shortest v-t path such that the second vertex in Q' is different from the vertex right after v in P. Each shortest simple detour of P at v can be computed by a v-t shortest path query after removing some vertices and arcs of the input digraph to guarantee that it deviates from P and it is simple. The shortest simple detours of an s-t simple path P is the set of simple shortest simple detour of P at each of its vertices.

Yen's algorithm uses Dijkstra's algorithm to compute each shortest simple detour, each of these calls is done independetly from the others. As the number of vertices of an s-t path is bounded by $O(n)$, Yen's algorithm takes $O(n(m + n\log n))$ per path, i.e, $O(kn(m + n\log n))$ to find k shortest simple paths.

Note that, the most expensive part (in terms of practical running time) in Yen's algorithm is the phase of computing the shortest simple detours.

From an algorithm engineering perspective, several works have been proposed to improve the efficiency of the algorithm in practice [67, 79, 111, 70, 59, 54, 83, 82]; they all feature the same worst-case running time as Yen's algorithm i.e, $O(kn(m+n\log n))$. All of these algorithms (except the Sidetrack-Based, SB, algorithm [83]) are Yen-Based algorithms, i.e, they only differ on how they compute the shortest simple detours. In what follows, we present two of these methods and their practical performances.

Node Classification (NC) algorithm In 2014, Feng [54] * proposed a new kSSP algorithm called the Node-Classification (NC) algorithm. The NC algorithm starts by computing a shortest path in-branching T rooted at t, and uses T to extract a first shortest path. Then, in contrast with Yen's algorithm where shortest simple detours are computed independently of each others. The NC algorithm classifies (with the help of T) at each shortest simple detour computation, the nodes (vertices) of the digraph into three parts :

— Red (forbidden vertices) : The removed vertices in Yen's algorithm,

— Green (bypass vertices) : Vertices reaching t via T without passing through a Red vertex,

— Yellow (scan-vertices) : All the remaining vertices.

So, as the bypass vertices, i.e, the green area remains valid, the NC algorithm restricts the search space of Dijkstra's call to the scan-vertices, i.e, the Yellow area, which is, in practice, considerably smaller than the whole digraph. Feng showed, experimentally on several real networks, that the NC algorithm outperforms all of preceding kSSP algorithms. Another advantage of the NC algorithm is its low memory consumption in practice, as it saves only a single SP in-branching in the memory (and the candidate paths †). In fact, the NC algorithm is, so far, the fastest kSSP algorithm with low memory consumption in practice.

Sidetrack Based (SB) algorithm Recently, Kurz and Mutzel [83, 82] proposed the Sidetrack Based (SB) algorithm. In contrast with other kSSP algorithms, the SB is not a Yen-based algorithm. Actually, the SB algorithm can be seen as the first non-obvious and polynomial time adaptation of Eppstein's kSP algorithm to extract only simple paths. We will start by describing a data structure representing a path in the SB algorithm, then we will briefly describe the SB algorithm.

The SB algorithm uses a data structure generalizing the representation of a path proposed by Eppstein [50]. Such compact representation uses sequences of shortest path in-branchings rooted at t, $T_0, T_1, \cdots, T_h$ and sidetracks $a_0, a_1, \cdots, a_h$ (this time, a sidetrack of a path P is any arc not in P but tailing at P).

Precisely, the sequence $\varepsilon = (T_0, a_0, T_1, a_1, \cdots, T_h, a_h, T_{h+1})$ with $a_i = v_iw_i$ for all $0 \leq i \leq h$, represents the path P starting at s, following T_0 until the tail v_0 of a_0, then the sidetrack a_0, then T_1 from the head w_0 of a_0 until it reaches the tail v_1 of a_1, *etc.* until it reaches the head

∗. Even though it is called Feng's algorithm in the literature [83, 82], almost the same algorithm is proposed by Gao et al. [59]

†. A path among the candidate paths does not exceed 1000 vertices in practice and saving the candidate paths is not an issue

w_h of a_h, plus (possibly) the path from w_h to t in T_{h+1}. That is, P is the sequence of vertices of the paths $P^{T_0}_{sv_0}, P^{T_1}_{w_0v_1}, \cdots, P^{T_h}_{w_{h-1}v_h}$ followed by the vertices of $P^{T_{h+1}}_{w_ht}$ if this latter path exists. Note that, two consecutive shortest path in-branchings T_i and T_{i+1} are not necessarily distinct (see Figure 1.10 for example).

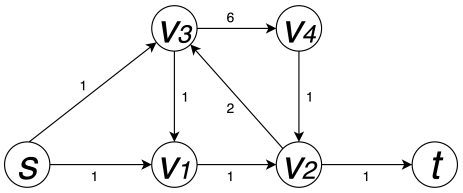


Figure 1.6 – A graph G

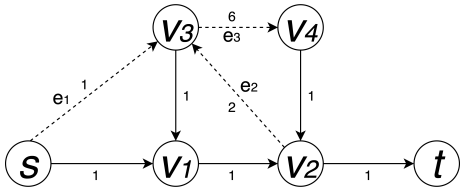


Figure 1.7 – An SP in-branching T_0 of G rooted at t

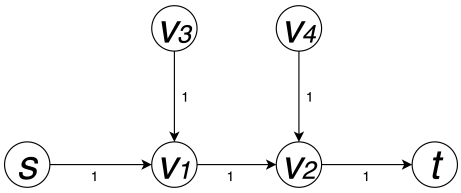


Figure 1.8 – T_0

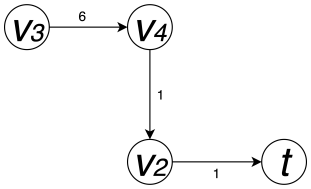


Figure 1.9 – An SP in-branching T_1 of $G \setminus \{s, v_1\}$ rooted at t

Figure 1.10 – An exemple of a path representation in the SB algorithm, the path $P = \{s, v_3, v_1, v_2, v_3, v_4, v_2, t\}$ can be represented as $(T_0, e_1, T_0, e_2, T_1)$, note that P is not simple.

The SB algorithm also uses a set $Candidates$ to manage candidate paths that are encoded using the above data structure. Sequentially, it extracts a shortest element ε from $Candidates$. If ε represents a simple path, this path is added to the output and the representations of its shortest, but not necessarily simple, detours (that will be found using the last in-branching in the representation of ε) are added to $Candidates$. Otherwise, the SB algorithm attempts to modify ε by instantiating its last in-branching. If this computation leads to a representation of a simple path, then it is added to $Candidates$. Otherwise, ε is discarded. The SB algorithm goes on iteratively until k paths are found. The initialization consists in computing a first shortest path in-branching T_0 rooted at t in D (using Dijkstra's algorithm) and so a shortest s-t (simple) path $P^{T_0}_{st}$ and adding its representation to $Candidates$.

Note that, for each element $\varepsilon = (T_0, a_0, T_1, a_1, \cdots, T_h, a_h, T_{h+1})$, a set of its corresponding shortest path in-branchings (up to $h+1$ in-branching if they are all distinct) is stored in the memory, and this may become hard to manage for large graphs and large requested number k of paths. In other words, a main drawback of the SB algorithm is its large memory consumption.

Kurz and Mutzel proved that their SB algorithm has the same complexity bound of Yen's algorithm, that is $O(kn(m+n\log n))$. Moreover, the SB algorithm is much faster than the previous algorithms in practice.

To conclude, the fastest known algorithm with low working memory consumption (i.e, the same working memory as Yen's algorithm) is the Node Classification (NC) algorithm, proposed independently by [54] and [59]. With larger memory consumption, the Sidetrack Based algorithm (SB) [83] can achieve an impressive speed up. For instance, the SB algorithm computes $k = 1000$

shortest paths in 50 ms for the DC network [44] ($n \approx 10,000; m \approx 30,000$) while it required about 5 s with Yen's algorithm and about 0.3 s by NC algorithm.

1.4.4 On arbitrarily weighted graphs

The problem of finding a shortest path from a source to a destination in an arbitrary weighted digraph has concrete applications, as negative arc lengths arise in several applications, typically in network survivability, in arbitrage problems in foreign exchange markets, in job scheduling with deadlines and so on [28, 109]. Similarly to digraphs with positive arc lengths, a single shortest path may not be sufficient for the same reasons. Despite the fact that the problem of computing a shortest path in an arbitrary weighted digraph has been intensively studied [106, 42, 40, 73, 115, 61], and the extensive number of researches tackling the kSP on positively weighted digraphs, we are not aware of any study on arbitrarily weighted digraphs, not even on digraphs with no negative cycles, dedicated directly to the kSP or kSSP. Finding a shortest s-t path in arbitrarily weighted digraph (known as shortest elementary path problem SEP) is NP-complete, and polynomial if the digraph has no negative cycles. So, solving the kSP or the kSSP seems to be more difficult on such digraphs.

1.4.5 Contributions

Here, we describe the contributions of this thesis, related to the k shortest paths problem. The details of these contributions are presented in Chapter 2.

On positively weighted digraphs We propose a new algorithm with low working memory consumption, called Postponed Node Classification (PNC), that is faster, in practice, than the NC algorithm while preserving the same working memory. Our experimental results show that the PNC algorithm is the fastest, in practice, kSSP algorithm with low working memory among all considered algorithms. Furthermore, it competes with the algorithms with large memory consumption on road networks.

Considering large working memory, we show how to speed up the SB algorithm using dynamic updates of shortest path trees resulting with the SB* algorithm, that is, in practice on road networks, the fastest kSSP algorithm (on median) with large memory consumption among the considered algorithms. Moreover, we propose a new algorithm called Parsimonious Sidetrack Based (PSB), that is based on the SB algorithm. The PSB algorithm gives, on road network, a space-time trade off between SB and NC algorithms. That is, it enables to significantly reduce the working memory of SB at the cost of an increase of the running time. Nonetheless, on complex networks, the PSB algorithm gives the best running time among all the considered algorithms.

We further propose parameterized variants of PSB (called PSB-v2 and PSB-v3) that improve its performances, both in terms of working memory consumption and of running time, on road networks.

We analyze the behavior of all the aforementioned algorithms on different types of networks (road, biological, Internet and social networks). We have also investigated the relationships between the performances of the algorithms and some properties of the queries, such as the number of hops and the stretch from the center. Finally, we end up with a first empirical framework for selecting the most suitable kSSP algorithm for a given use case.

On arbitrarily weighted digraphs Also, we have studied the problem of finding k shortest simple paths from a source to a destination in arbitrary weighted digraphs. We first prove that the classical framework of Yen remains valid on arbitrarily weighted digraphs. Precisely, Yen's algorithm remains valid after replacing the shortest path algorithm used in Yen's algorithm, i.e, Dijkstra's, by any other exact shortest path algorithm able to compute a shortest simple path in arbitrary weighted digraphs. This leads to several improvements.

Considering arbitrary weighted digraphs with no negative cycles, the kSSP can be solved naively by replacing Dijkstra's algorithm by Bellman-Ford-Moore's algorithm. This gives a polynomial time algorithm, referred to as Y-BFM, with time complexity in $O(kn^2m)$. Furthermore, we show how to use shortest path tree update [90] in order to improve the time complexity to $O(kn(m + n \log n))$, which is, surprisingly, equal to the complexity of Yen's algorithm. Moreover, we propose a new algorithm, called Postponed Yen (PY-BFM), that is, more than one order of magnitude faster than the Y-BFM algorithm in practice. Finally, we design a special variant of PY-BFM called PY-BFM* that is, up to twice as fast as PY-BFM.

In contrast to the previous context, the problem of finding a shortest simple path on arbitrary weighted digraphs (with negative cycles), also called shortest elementary path (SEP) [92], is NP-Hard as the Hamiltonian path problem can be reduced to it. However, as we are also interested in efficient solutions, we give several initiatives to solve the problem. This is done by proposing several mixed integer linear programming (MIP) models.

1.5 The k (shortest) dissimilar paths problem

Motivation A major drawback of the kSSP is that, in practice, the k shortest paths are somehow "similar". As shown in Figure 1.11, solving naively the k shortest simple path may lead to very similar paths, that are the same from a user's perspective. For instance, except in the case of a car accident or a demonstration, no one is usually interested in taking tiny local detours of a path.

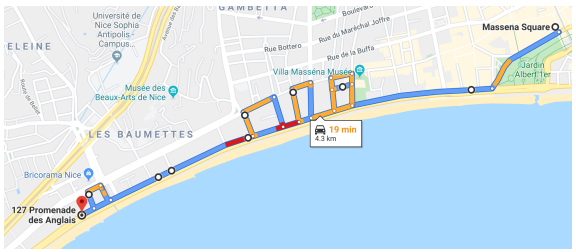


Figure 1.11 – Six shortest paths from a source to a destination in a road network.

State of the art The similarity between two paths can be measured according to the proportion of arcs they share and the problem of computing "dissimilar" (shortest) paths has been investigated.

The first related study, we are aware of, is proposed by Erkut and Verter [53], motivated by the transportation of hazardous materials where it is recommended to avoid residential areas and crowded routes. In [53], Erkut and Verter introduced several definitions of the similarity between two paths (including the Jaccard and the Max measures also studied later in [34]). Few years later, Akgün *et al.* [3] proposed and analysed the first basic solution, consisting in computing a huge set of shortest paths and then selecting a subset of paths that are mutually dissimilar (for

one of the similarity measures defined in [53]). In their experiments, this method scaled only on small transportation networks (about 300 vertices). The first scalable solutions were proposed by Abraham *et al.* [1] where a shortest path P is fixed, and "locally shortest" paths with limited intersection with P are requested (this corresponds to the Asymmetric measure defined below). However, except for the initial path P, this definition does not guaranty any mutual dissimilarity between the computed paths. A noticeable heuristic proposed in [1] is the penalty based approach. This heuristic adds a penalty on the arcs of the already chosen paths in order to limit the chances of falling back on them again.

Several recent studies by Chondrogiannis *et al.* (see [35]) offer both theoretical and empirical study of the problem of finding k shortest dissimilar paths. First, they formally proved that finding k shortest dissimilar paths is weakly NP-complete for both the Asymmetric measure and a new dissimilarity measure that they define (referred to as Min measure below). For these two measures, they proposed an exact pseudo polynomial time algorithm with several pruning techniques enabling to find 4 dissimilar paths in the road network of Rome (3,000 vertices) in less than one second. They also proposed advanced heuristics enabling to scale on a road network with one million vertices and two millions arcs while achieving acceptable paths "quality".

As shown in Table 1.5, each of the considered similarity measures, and those of the state of the art that we are aware of [53, 3, 35], are a function between zero and one indicating how much two given paths are similar. All of them are a ratio between the length of the common arcs over the length of one of the two paths or their union. Each of these measures has its own advantage. For instance, Min and Asymmetric measures are interesting in transportation scenarios, especially in the process of finding shortest paths. However, the Jaccard measure gives a natural and well known set similarity measure. Finally, the Max measure gives a restricted and constrained measure.

Contributions In Chapter 3 of the thesis, we further study the computational complexity of computing (shortest) dissimilar paths for four of the main measures. More formally, let P, P' be two paths of D and let $\ell(P \cap P') = \sum_{e \in A(P) \cap A(P')} \ell(e)$. The four considered measures are defined as follows.

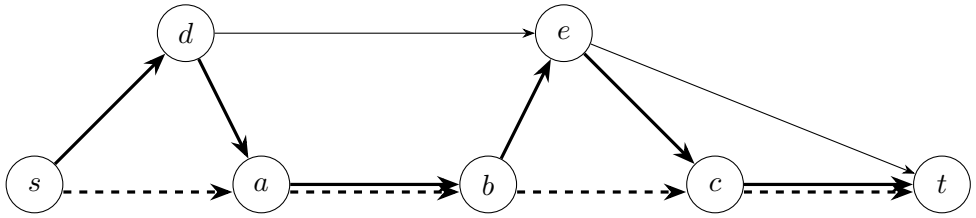


Figure 1.12 – Two s-t paths, $Q = (s, a, b, c, t)$ (dotted) and $Q' = (s, d, a, b, e, c, t)$ (thick). Every arcs have length 1.

Name (Z)	Jaccard [53]	Asymmetric [1]	Min [34]	Max [53]
$S_Z(P, P') =$	$\frac{\ell(P \cap P')}{\ell(P \cup P')}$	$\frac{\ell(P \cap P')}{\ell(P)}$	$\frac{\ell(P \cap P')}{Min\{\ell(P), \ell(P')\}}$	$\frac{\ell(P \cap P')}{Max\{\ell(P), \ell(P')\}}$
[Fig.1.12] $S_Z(Q, Q')$=	0.25	0.5	0.5	0.33

Let $\mathcal{S} = \{Asymmetric, Jaccard, Min, Max\}$. Given one of the similarity measures $Z \in \mathcal{S}$ and a threshold $0 \leq \theta \leq 1$, two paths P and P' are said *θ-dissimilar* (or P' is said *θ-dissimilar to* P in the case of asymmetrical measure) for a measure Z if $S_Z(P, P') \leq \theta$.

In this thesis, we study the problem of finding k shortest pairwise dissimilar paths. We give a unified and simple proof of the NP-completeness of this problem for each of the four similarity measures defined above.

Nevertheless, in many practical scenarios, a part of the solution is generally already given or it can be easily determined. For instance, a first path i.e, a shortest path can be computed in polynomial time, an alternative path could be an earliest arrival path, *etc*. Therefore, a natural question is whether one can find a (shortest) path dissimilar to a set of given paths.

We study the problem from this perspective. More precisely, we show that if only one path P is initially given, computing a second path that is dissimilar to P for the Asymmetric measure can be done in polynomial time while it is NP-complete for the remaining measures (Min, Max and Jaccard). Moreover, we prove that finding a path dissimilar (for each of the considered four measures) to a given set of $k \geq 2$ paths is NP-complete on DAGs. Finally, for each of these four measures, we show that computing a shortest path among those dissimilar to a given path is NP-complete on DAGs.

Finally, we propose an alternative pseudo polynomial time algorithm allowing to find k shortest dissimilar paths for the Asymmetric and the Min measures. Our alternative algorithm has the advantage to work on arbitrarily weighted graphs with no negative cycles.

1.6 The k earliest arrival time journeys in public transit networks

As mentioned before, a road network can easily be modeled as a weighted digraph, and finding k "best" (shortest, fastest or cheapest) paths from a given origin to a given destination in a road network is straightforward using any kSSP algorithm. Unfortunately, this problem becomes harder in a *public transit network* (also referred to as a transportation network, a multimodal network or a timetable). First, because public transit networks are time dependent, i.e., certain segments of the network can only be traversed at specific times. Second, several additional optimization criteria are considered in public transit network such as the arrival time, the departure time, the number of transfers, *etc*. In this section, we present the state of the art of journey planning in public transit networks, and our adaptation of the k shortest paths problem and some of its algorithms to such context.

1.6.1 Public Transit models

In this section, we review three main models and several variants of the shortest path problem in these public transit models.

Time-expanded model The main idea of the time-expanded model is to build a space-time graph (often also called an event graph) that encodes time. Roughly, suppose there are T time slots (during a day), a time-expended model creates T copies of the station-connections graph, where each copy G_t correspond to the timetable at a time slot t for $t \in [0; T]$. Moreover, for each $t \leq t' \leq T$, each vertex u_t of G_t is connected to each u_{t+1} and each vertex $v_{t'}$ if there is an elementary connection (defined below) from u to v starting at t and ending at t'. For example, suppose there are two stops u and v with an elementary connection starting from u at $9h00$ and reaching v at $9h05$, the Time-expended model creates two copies u_{9h00}, v_{9h00} and u_{9h05}, v_{9h05} with an arc (u_{9h00}, v_{9h05}) (the possibility to take the connection), and an arc (u_{9h00}, u_{9h05}) (waiting at the same stop) if there is no other event at u between $9h00$ and $9h05$.

Time dependent model The main disadvantage of the time-expanded model is that making T copies (for a time interval T) of the network results a large graph. Besides, an alternative model called the time-dependent approach can be used to achieve smaller graphs. In a time-dependent model, vertices correspond to stops (bus stops for instance), and an arc is added from a vertex u to another v if there is at least one elementary connection serving the corresponding stops. In other words, departure and arrival times are encoded by the travel time function associated with the arc (u, v). For example, two stops u and v with three elementary connections between them (at $9h00 - 9h05$, $9h10 - 9h15$ and $9h30 - 9h40$) are connected by three arcs, each of them labelled with its departure and arrival time.

It is worth mentioning that an alternative model called a **Frequency-Based Model** is proposed by Bast and Storandt [22]. This model exploits the fact that in real-world timetables trips often operate according to specific frequencies at times of the day. For instance, a bus may run every 5 minutes during rush hour, and every 10 minutes otherwise, *etc*. This model can be seen as a time-dependent model with an extra advantage of compacting the data structure encoding the whole set of connections into a basic function of time intervals and frequencies. For example, two stations u and v with an an elementary connection between them passing each 5 mns from $9h00$ to $19h00$ can be modeled by a single arc from u to v labelled with $(f = 5mns, T = 9h00 - > 19h00)$.

Multimodal A multimodal network combine time-constrained means of transport (bus, metro *etc*.) and unconstrained one (walking, bike, *etc*.), i.e., static graphs. Pajor [98] discusses how to model the union of these two types of networks in order to adapt the existing algorithms to such cases. Similarly, Wagner and Zündorf [120] (improved by Phan and Viennot [100]) presented an example of how connections between vertices can be found in an underlying static graph during the exploration of a temporal graph, typically for unrestricted walking journey planning.

In this thesis, we only consider time-dependent models, which are not necessarily frequency-based.

A *timetable* (or public transit network) represents for one specific day the vehicles that exist (train, bus, tram, ferry, ...), when they travel, where they travel and how a passenger can go from one vehicle to another.

Formally, a timetable is a quadruple $\mathcal{T} = (S, T, C, F)$ of stops S, trips T, connections C and footpaths F :

- A *stop* is a position outside a vehicle where a passenger can wait. At a stop (and only at a stop) a vehicle can halt and passengers can leave or get on.
- A *trip* is defined by a vehicle going through stops at fixed times. Precisely, a trip is a scheduled vehicle, i.e, a journey done by a unique vehicle from a starting stop to a last stop at a fixed time.
- A *connection* is a vehicle going from one stop to another with no intermediate stops.
- A *footpath* is used to model a transfer, i.e, how to get from one vehicle to another. We suppose that footpaths are transitively closed, i.e, if there is a footpath from a to b and from b to c then there is a footpath from a to c.

A *journey* is a sequence of trips and footpaths one can take within a transit network.

1.6.2 Problem Variants

In contrast to road networks where focus is on computing a shortest path according to a single length function (distance, travel-time, *etc.*), there are various variants of this problem for public transit networks.

As mentioned in [21], the *earliest arrival problem* is the simplest variant. That is, given a source stop o, a target stop d, and a departure time t_0. It asks to find a journey departing from o after t_0 and reaching d as soon as possible. Similarly, the latest departure problem takes an arrival time t_{max} and asks to find a journey from o, reaching d before t_{max} and departing as late as possible. A related variant is the *profile* (or *range*) problem, which replaces the departure time by a time range $[t_0, t_{max}]$ (a whole day, for instance), and asks to find a set of journeys of minimum travel time that depart within that range. In other words, it asks to find a set of Pareto optimal solutions between the earliest arrival time (after t_0) and the latest departure time (before t_{max}). Note that, other criteria may be considered, as the number of transfers or the cost of the journey, this leads to the *multicriteria problem*.

1.6.3 Earliest arrival journey planning

Journeys planning in (schedule-based) public transit networks and accelerating queries for efficient journey planning is a long-standing problem [21]. In the last decade, many algorithms have been developed to answer efficiently basic queries like an earliest arrival journey, and to optimize additional criteria like the number of transfers, the cost of the trip, *etc.* or even to offer Pareto optimal solutions combining several criteria [21, 43, 46].

Plethora of algorithms were proposed to efficiently answer queries of optimal journeys from a given origin o to a given destination d after a departure time t_0 in a public transit network. For instance, the *Connection Scan Algorithm (CSA)* [46] is the fastest known algorithm, without any preprocessing routine, enabling to find an earliest arrival journey from o to d departing after a departure time t_0, i.e., finding an earliest arrival time journey. With the help of a heavy preprocessing routine, the Transfer Patterns algorithm [20] can achieve a tremendous speed up with respect to the CSA. Besides, *Round Based Public Transit Routing (RAPTOR)* [43] is the fastest known algorithm (also without any preprocessing routine) enabling to compute a Pareto optimal set of journeys optimizing the arrival time and the number of transfers of a journey. Recently, Bast et al. [21] presented an extensive survey on the topic of journey planning in road and public transit networks.

Connection Scan Algorithm Similarly to Dijkstra's algorithm, the CSA stores an earliest arrival time for each stop in an array. A connection c is considered *reachable* if a passenger can sit in the public transit vehicle of the connection, i.e, a traveler starting from the origin stop o, can reach the departure stop of c before its departure time (the departure time of c). However, the main difference between Dijkstra's algorithm and the CSA is the fact that the CSA does not use a priority queue. Instead, the CSA iterates over all the connections sorted by their departure time (the same ordering is used for all queries). The CSA checks whether a connection is reachable or not. If so, it improves the arrival time at the arrival stop of the connection. Once all the connections have been scanned, the earliest arrival time to a stop is the current arrival time stored for the stop. The main advantage of avoiding the use of a priority queue is that, while more connections are scanned, the amount

of work per connections is significantly reduced. This is why the CSA is significantly faster than Dijkstra's algorithm in this context [46].

Profile Connection Scan Algorithm The result of the Profile Connection Scan Algorithm (PCSA) is a mapping between a departure time from a departure stop onto the earliest arrival time at the arrival stop. In other words, the profile problem solves simultaneously the earliest arrival problem for all departure times.

Compared with the CSA, the PCSA iterates on the connections sorted decreasingly by departure time, which leads to the fact that it solves the all-to-one problem. The PCSA constructs journeys from late to early and exploits the fact that an early journey can only have later journeys as subjourneys. It has been reported in [46] that the PCSA is one order of magnitude slower than the CSA, which is acceptable considering the fact that it solves the all-to-one problem.

RAPTOR The *Round Based Public Transit Routing (RAPTOR)* algorithm [43] is based on dynamic programming. Precisely, it computes the travel times from the source to any vertex of the graph using a single trip, then, with the help of this solution it computes the travel time to any vertex using at most two trips, *etc*. until it finds the travel time from the source to the remaining vertices using at most k trips (such that, the travel times are the same as those of the $k - 1^{th}$ iteration. As discussed in [21], RAPTOR is one of the most efficient algorithm without preprocessing.

1.6.4 k earliest arrival journeys query algorithms

Despite the extensive research on finding optimal journeys in public transit networks, we are aware of only two initiatives of finding k optimal journeys in such context. Precisely, there are two known papers where the problem of finding k shortest (or fastest) arrival time journeys in public transit networks is studied, and concrete algorithms were proposed.

First, Vo et al. [119] proposed a time dependent graph model of a bus network. Then, they adapt Yen's algorithm to find alternative journeys in this network model. Precisely, they select a set of alternative journeys (journeys sharing only a limited part of their common edges) among those given by Yen's adaptation.

As shown above, Yen's algorithm uses Dijkstra's algorithm as a basic brick to compute shortest detours of a given path. Analogously, Vo et al. [119] used a standard time-dependent shortest path (TDSP) algorithm [107] to compute earliest detours of a journey in a bus network. They evaluated their method on a single network of around $4\,000$ stops and $8\,000$ connections, resulting in an average running time of around 1 second to find 5 journeys.

On the other hand, Scano et al. [104] proposed a labelled directed graph model of a transportation network where a label is an object composed of the transportation mode (foot, car, bus, *etc*.) and a travel time. This model merges a road and a public transport network together. Then, it is shown how the k shortest paths algorithms can be adapted for this model. Specifically, they adapted Yen's and Eppstein's algorithm to work on their model. In both algorithms, a Dijkstra-like algorithm called Dijkstra Regular Language Constraint (DRegLC) [19] is used to answer earliest arrival journeys queries. Moreover, an Iterative Enumeration Algorithm (IEA) is proposed to extract only simple journeys using Eppstein's algorithm. i.e, using Eppstein's k shortest paths algorithm as an iterator and then selecting the simple corresponding journeys (a journey is *simple* if it does not visit a stop more than once).

Experimentally, Scano et al. showed that their IEA is faster than Yen's straightforward adaptation on the transportation network of Toulouse (75 000 nodes, 500 000 road edges and 43 000 public transport edges). On this network, the average running time of Yen's adaptation to find 100 journeys is 250 seconds while it is 0.6 seconds using their refined IEA. However, IEA is not a polynomial-time algorithm, and its memory consumption is too high [104]. In addition, using the labelled directed graph model described in [104] may cause a duplication of the public transit part in practice, i.e, many journeys given by the algorithms proposed in [104] may only differ on the foot-path part while sharing the exact same public transit part. This is undesirable in applications requesting diverse public transit journeys.

Our contribution In this thesis (Chapter 4) [‡], we study the problem of finding k earliest arrival journeys from a given origin to a given destination in a public transit network. For this purpose, we use the timetable model of public transit networks, i.e, the well-known common model used in [21, 46, 43]. First, we propose a performant adaptation of Yen's k shortest simple paths algorithm to public transit networks (Yen - Public Transit, Y-PT algorithm). In contrast with [104, 119], we use the Connection Scan Algorithm (CSA) to answer earliest arrival journey queries in our algorithm.

Our main contribution is a novel algorithm, called Postponed Yen's algorithm for Public Transit networks (PY-PT). With the help of a lower bound on the arrival time of a detour journey (a journey that could be one of the k earliest arrival journeys), PY-PT postpones the effective computation of such detour (and so the corresponding earliest arrival journey queries using CSA) with the aim of skipping it.

Our experimental results on several train and public transit networks show that the running time of our adaptation of Yen's algorithm is acceptable in practice. Moreover, on the same dataset, the PY-PT algorithm performs 10 to 30 times faster than the Y-PT algorithm on average.

1.7 Techniques used

In this section, we describe briefly the technologies and the datasets used in the experiments conducted during the thesis.

k shortest simple paths in a graph. In order to evaluate our algorithms and their speed-up with respect to the state-of-the-art algorithms, we implemented in C++ (using the STL library) the most performant state-of-the-art kSSP algorithms and our algorithms. We run experiments on road networks of the 9^{th} DIMAC's challenge and various complex networks : biological, social and Internet networks from [89]. In order to evaluate algorithms on negative weighted digraphs, we conducted experiments on complex networks with negative length functions like cryptocurrency exchanges and social trust network, and road networks of DIMAC's challenge with small modifications, i.e., flipping the sign of the length of some arcs. We made our code publically available on Gitlab [13].

We have considered the running time and the number of shortest path calls, i.e, Dijkstra's algorithm calls for positively weighted digraphs and Bellman-Ford-Moore algorithm for negatively weighted digraphs with no negative cycle. Note that the number of shortest path algorithm

‡. The contributions related to journey planning in multimodal transportation networks are the outcome of collaborations of the ANR-MULTIMOD members, where we manage to exchange the industrial need and afford an appropriate tool to our industrial partner PME Instant-System.

calls is an indication of the running time and the working memory, which is independent of the implementation and the architecture of the machine.

k earliest arrival time journeys in public transportation networks. We have implemented Y-PT and PY-PT algorithms in Java and we made our code publicly available [16]. We have evaluated the performances of our k earliest arrival journey algorithms on two train networks (Germany and Switzerland) and three public transit networks (Paris, Berlin and Stockholm). This dataset is publicly available (`https://transitfeeds.com/`).

Similarly to the kSSP algorithms evaluations, we have considered the execution time and the number of CSA calls. Note that the number of CSA calls is also a machine and implementation independent indication of the running time.

1.8 Summary of the contributions

The k shortest simple paths problem. This problem has different flavors depending on each context and the considered constraints. In this thesis, we outperform the state-of-the-art algorithms in practice, we study the problem with dissimilarity constraints, and we adapt some of these algorithms to work with public transportation networks. The contribution of this thesis can be summarized as follows :

— **On the k shortest simple paths** : First, we study the original k shortest simple path problem (kSSP). The following initiatives are outlined :
  — New kSSP algorithms, outperforming the state-of-the-art algorithms either in the running time in practice, in the practical space requirement or in offering space-time trade-offs.
  — Analysis of the performance of the considered kSSP algorithms on road and complex networks in order to conclude the best algorithm for each use case and end up with an empirical framework.
  — Initialization of the study of the kSSP on digraphs with arbitrarily weighted arcs (possibly negative), and show how and where some algorithms can be adapted to such case.

  Some of these results have been presented at SEA'20 [10], AlgoTel'20 [8] and JGA'20 [7] (full version submitted to a journal [12]).

— **On the k (shortest) dissimilar paths** : Second, we study the k shortest dissimilar paths problem. For four of the most studied measures in the literature, we give a unified and simple proof of the fact that finding k shortest dissimilar paths is NP-complete. Moreover, we consider the problem of finding a (shortest) path that is dissimilar to one or more given paths. This study results with a complexity classification of several variants of the problem with respect to the similarity measure, the number of the given paths and the optimization criteria. In addition, we propose an alternative pseudo-polynomial time algorithm allowing finding k shortest dissimilar paths for two of the considered measures.

  These results have been presented at AlgoTel'21 [11] and ROADEF'20 [9] (full version submitted to a journal [15]).

— **On the k earliest arrival journeys** : Finally, we study the k earliest arrival path problem ($kEAT$) in public transit networks. In contrast with the state of the art, we use a well-known public transportation network model, and we show how to extend Yen's algorithm to public transit networks. Moreover, we propose a more refined algorithm enabling a considerable speed up in practice.

These results have been obtained in collaboration with Arthur Finkelstein from Instant-System and are submitted to a conference [6].

We aim, at the end of this thesis, at building an empirical framework allowing to recommend the best kSSP algorithm to each use case, regarding various aspects starting from the model (graph, transportation network), the additional criteria (dissimilarity of the paths), the topology of the input graph (complex or road network), the length function of the arc (positive, arbitrarily, arbitrarily weighted with no negative cycles), and the resources' availability (typically the memory consumption).

1.9 Overview of the manuscript

The manuscript is organized as follows. In Chapter 2, first, we explain in details the state-of-the-art algorithms of the kSSP problem. Then we describe our new algorithm and their evaluations on road and complex networks. Finally, we show how to adapt these algorithms to arbitrarily weighted digraphs (with or without negative cycles). Chapter 3 is dedicated to the k (shortest) dissimilar paths, where the notion of similarity between paths is formally defined and discucced with respect to each motivation, then we study the complexity of finding (shortest) dissimilar paths from a source to a destination, for each case and similarity measure. Finally, we describe a pseudo-polynomial time algorithm allowing to correctly find k shortest dissimilar paths for two of the given similarity measures. The study of journey planning in public transit networks is presented in Chapter 4, where our algorithms for finding the k earliest arrival journeys are proposed and analyzed on different public transportation and train networks. Finally, in Chapter 5, we conclude with a generic framework to find k best paths in a network with respect to the type, topology, length-function of the network, the additive constraints and resources availability of the machine.

1.10 Publications

International Conferences

— A. Al Zoobi, D. Coudert, N. Nisse. Space and time trade-off for the k shortest simple paths problem. SEA 2020-18th International Symposium on Experimental Algorithms 160 (18), 13, 2020.

— C. Gou, A. Al Zoobi, A. Benoit, M. Faverge, L. Marchal, G. Pichon, P. Ramet, Improving Mapping for Sparse Direct Solvers, European Conference on Parallel Processing (EuroPar 2020), 167-182, 2020. (contributions established during my master's internship at the ENS of Lyon)

— A. Al Zoobi, D. Coudert, A. Finkelstein. On finding earliest arrival time journeys in public transit networks (ICORES 2022).

National Conferences

— A. Al Zoobi, David Coudert and Nicolas Nisse. De la difficulté de trouver des chemins dissimilaires, AlgoTel 2021

— A. Al Zoobi, David Coudert and Nicolas Nisse. Compromis espace-temps pour le problème de k plus courts chemins simples, AlgoTel 2020

— A. Al Zoobi, David Coudert and Nicolas Nisse. On the Top-k Shortest Paths with Dissimilarity Constraints (ROADEF 2020)

Submitted

— A. Al Zoobi, D. Coudert, N. Nisse. Finding the k Shortest Simple Paths : Time and Space trade-offs. (submitted at JEA).

— A. Al Zoobi, D. Coudert, N. Nisse. On the complexity of finding shortest dissimilar paths in a graph. (submitted at DMTCS).

This thesis has been done in the context of ANR project MULTIMOD with the reference number ANR-17-CE22-0016 and with the financial support of Région Sud PACA.

On finding k shortest simple paths in a graph

The k shortest simple path problem (kSSP) asks to compute a set of top-k shortest simple paths from a source to a destination in a digraph. Yen (1971) proposed an algorithm with the best known polynomial time complexity for this problem, that is in $O(kn(m + n \log n))$ where n is the number of vertices and m the number of arcs. Since then, the problem has been widely studied from an algorithm engineering perspective. The most noticeable proposals are the node-classification (NC) algorithm (Feng, 2014) and the sidetracks-based (SB) algorithm (Kurz and Mutzel, 2016). The latest offers the best running time at the price of a significant working memory.

We first show how to speed up the SB algorithm using dynamic updates of shortest path trees resulting in a faster algorithm (SB) with same working memory. We then propose the parsimonious SB (PSB) algorithm that significantly reduces the working memory of SB at the cost of a small increase of the running time. Furthermore, we propose the postponed Yen (PY) algorithm that combines the best of NC and SB. It offers a significant speed up compared to NC while using the same amount of working memory of NC.*

Our experimental results on complex networks show that all of the considered algorithms have low working memory, and that the PSB algorithm is the fastest. On road networks, the SB algorithm is the fastest (on median) among the considered algorithms, but it suffers from a large working memory. The PNC algorithm has comparable running time to SB* on road networks while using the same working memory as NC.*

We also initiate the study of the kSSP problem on arbitrarily weighted digraphs. First, we consider digraphs with no negative cycles where we show that the problem can be solved with the same time complexity as Yen's algorithm, i.e, $O(kn(m+n \log n))$. Then, we show how to adapt some of the original kSSP algorithms to work with arbtitrarily weighted digraphs with no negative cycles. We propose three algorithms, called Yen-Bellman Ford Moore (Y-BFM), Postponed Yen - Bellmman Ford Moore (PY-BFM) and Postponed Node Classification - Bellman Ford Moore (PNC-BFM). Our experiments on road and complex networks (where weights have been arbitrary modified to include negative weights) show that the PNC-BFM algorithm is between one and two orders of magnitude faster than Y-BFM in practice. Finally, we propose several Mixed Integer linear Programming (MIP) models to solve the kSSP on arbitrarily weighted digraphs, i.e, with and without negative cycles.

2.1 Introduction

The classical k shortest paths problem (kSP) aims at finding a set of top-k shortest paths between a pair of source and destination nodes in a directed weighted graph. 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 weighted digraph with a length function $\ell : A \to \mathbb{R}^+$ (or in $\mathbb{R}$ for arbitrarily weighted digraphs), an s-t path is a sequence $(s = v_0, v_1, \cdots, v_l = t)$ of vertices starting with s and ending with 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 length of a path is the sum of the length of its arcs. A *cycle* is a path with same starting and ending nodes. A negative weighted cycle is a cycle s.t. the sum of the lengths of its arcs is negative. A top-k shortest paths is a set containing a shortest s-t path, a second shortest s-t path, *etc.* until a k^{th} shortest s-t path.

Several algorithms for solving kSP have been proposed. In particular, Eppstein [50] proposed an exact algorithm that computes k shortest paths (not necessarily simple) in $O(m + n \log n + k)$-time, where m is the number of arcs and n the number of vertices of the graph. An important variant of this problem is the k shortest *simple* paths problem (kSSP) introduced in 1963 by Clarke *et al.* [37] which adds the constraint that reported paths must be simple. This variant of the problem has various applications in transportation network when paths with repeated vertices are not desired by the user. It is also a subproblem of other important problems like constrained shortest path problem, vehicle and transportation routing [67, 76, 125]. It also can be applied successfully in bio-informatics [17], especially in biological sequence alignment [110] and in natural language processing [24]. For more applications, see Eppstein's recent comprehensive survey on k-best enumeration [51].

The most famous algorithm for solving the kSSP problem has been proposed by Yen [127] and has time complexity in $O(kn(m + n \log n))$. It has been proved that the kSSP problem can be solved in $O(k)$ iterations of APSP (All Pairs Shortest Paths) [64]. This means that the kSSP problem can be solved in $O(kn(m + n \log \log n))$-time on sparse digraphs and $O(kn^3(\log \log n)/\log^2 n)$-time on dense digraphs using the fastest APSP algorithms [99, 68]. Vassilevska Williams and Williams [118] show that a subcubic kSSP algorithm would also result in a subcubic algorithm for APSP, which seems unlikely at the moment. Recently, Eppstein and Kurz [52] proved that on digraphs with bounded treewidth, the kSSP problem can be solved in time $O(n + k \log n)$.

On the other hand, several works have been proposed to improve the efficiency of the algorithm in practice [67, 79, 111, 70, 59, 54, 83, 82]; they all feature the same worst-case running time as Yen's algorithm i.e, $O(kn(m + n \log n))$.

In particular, Feng [54] proposed an improvement of Yen's algorithm called the Node Classification (NC) algorithm. With the help of a precomputed shortest path tree of the digraph, the NC algorithm classifies the vertices of the digraph with respect to their validity. Thus, computing a shortest simple detour will be restricted to the non-valid sub-digraph that is smaller than the original one. By this restriction, the running time of computing a shortest path remarkably decreases. An interesting fact about the NC algorithm is that its memory consumption is almost the same as Yen's algorithm (only a shortest path tree is kept in the memory).

Recently, Kurz and Mutzel [83, 82] obtained a tremendous improvement of the practical running time, designing an algorithm called the Sidetrack Based (SB) with the same flavor as Eppstein's algorithm.

The key ideas are to define a path using a sequence of shortest path trees and *deviations*, and to postpone as much as possible the computation of shortest path trees. With this new algorithm, they were able to compute hundreds of paths in graphs with million vertices in about one second, while previous algorithms required an order of tens of seconds on the same instances. For instance, Kurz and Mutzel's algorithm computed $k = 1000$ shortest paths in 50 (ms) for the DC network [44] while it required about 5 (s) with Yen's algorithm and about 0.3 (s) by the improvement proposed by Feng [54]. The drawback of the SB algorithm is the need for storing all computed shortest path trees, thus leading to a large usage of working memory.

Finally, the fastest algorithm with low working memory consumption (i.e, the same working memory as Yen's algorithm) is the Node Classification (NC) algorithm, proposed independently by [54] and [59]. With larger memory consumption, the Sidetrack Based algorithm (SB) [83] can achieve a tremendous speed up.

In the case of arbitrarily weights, negative arc lengths arise in several applications, typically in network survivability, in arbitrage problems in foreign exchange markets, in job scheduling with deadlines and so on. So, the problem of finding a shortest path from a source to a destination in an arbitrarily weighted digraph has concrete applications [28, 109]. Similarly to digraphs with non-negative arc lengths, a single shortest path may not be enough, as it may be blocked, unsatisfying some constraint or simply does not meet the user's desire for diversity. While the problem of computing a shortest path in an arbitrarily weighted digraph has been intensively studied [106, 42, 40, 73, 115, 61], we are not aware of any study dedicated to the problem of finding k shortest simple paths in an arbitrarily weighted digraphs.

Our contributions. We propose a new algorithm with low working memory consumption, called the Postponed Node Classification (PNC), that is much faster than the NC algorithm while using the same working memory. Our experimental results show that the PNC algorithm is the fastest kSSP algorithm with low working memory among all considered algorithms. Furthermore, it competes with the algorithms with large memory consumption on road networks.

Considering large working memory, we show how to speed up the SB algorithm using dynamic updates of shortest path trees resulting with the SB* algorithm, that is, the fastest kSSP algorithm (on median) with large memory consumption among the considered algorithms on road networks. Moreover, we propose a new algorithm called Parsimonious Sidetrack Based (PSB), that is based on the SB algorithm. The PSB algorithm gives, on road network, a space time trade off between SB and NC algorithms. That is, it enables to significantly reduce the working memory of SB at the cost of an increase of the running time. Nonetheless, on complex networks, the PSB algorithm gives the best running time among all the considered algorithms.

We further propose parameterized variants of PSB (called PSB-v2 and PSB-v3) that improve its performances, both in terms of working memory consumption and of running time, on road networks.

We analyse the behavior of all the aforementioned algorithms on different types of networks (road, biological, Internet and social networks). We have also investigated the relationships between the performances of the algorithms and some properties of the queries, such as the number of hops and the stretch from the center. Finally, we end up with a first empirical framework for selecting the most suitable kSSP algorithm for a given use case.

Moreover, we study the kSSP in arbitrarily weighted digraphs, where we show that the classical framework of Yen remains valid on arbitrarily weighted digraphs. Precisely, Yen's algorithm remains exact after replacing the shortest path algorithm used in Yen's algorithm (Dijkstra's al-

gorithm) by any other exact shortest path algorithm able to compute a shortest simple path in arbitrarily weighted digraphs. This leads to several improvements.

Considering arbitrarily weighted digraphs with no negative cycles, the kSSP can be solved naively by replacing Dijkstra's algorithm by Bellman-Ford-Moore's algorithm. This gives a polynomial time algorithm, referred to as Yen - Bellman-Ford-Moore (Y-BFM), with time complexity in $O(kn^2m)$. Furthermore, we show how to use shortest path tree update [90] in order to improve the time complexity to $O(kn(m + n \log n))$, which is surprisingly equal to the complexity of Yen's algorithm on non negative weighted digraphs. In addition, we propose a new algorithm, called the Postponed Yen - Bellman-Ford-Moore (PY-BFM), and an improved variant called Postponed Node Classification - Bellman-Ford-Moore (PNC-BFM). Our experiments show that the PNC-BFM is the fastest kSSP algorithm among the considered ones, and that it outperforms the Y-BFM algorithm by one to two orders of magnitude.

In contrast, the problem of finding a shortest simple path, also called shortest elementary path (ESP) [92], is NP-hard on arbitrarily weighted digraphs (with negative cycles allowed), as the Hamiltonian path problem can be reduced to it. However, as we are also interested in efficient solution, we give several initiatives to solve the problem by proposing mixed integer linear programming (MIP) models.

This chapter is organized as follows. We start recalling the Yen and NC algorithms in section 2.2. Then, we present in Section 2.3 the SB algorithm with our improvement SB*. In Section 2.4, we present the PSB algorithm and its two variants PSB-v2 and PSB-v3. Section 2.5 is devoted to the presentation of the PY algorithm. We present our experimental evaluation of all these algorithms on various networks in Section 2.6. The adaptations of some kSSP algorithms to arbitrarily weighted digraphs, with their experimental evaluations are presented in Section 2.7 and the MIP models for arbitrarily weighted digraphs are described in Section 2.8.

2.2 Preliminaries

2.2.1 Definition and Notation

We use the same graph definitions and notations of section 1.2.1 page 16.

Given $s, t \in V$, a *top-k set of shortest s-t paths* is any set S of (pairwise distinct) simple s-t paths such that $|S| = k$ and $\ell(P) \leq \ell(P')$ for every s-t path $P \in S$ and s-t path $P' \notin S$.

The k shortest simple paths problem takes as input a weighted digraph $D = (V, A)$, $\ell : A \to \mathbb{R}^+$ and a pair of vertices $(s, t) \in V^2$ and asks to find a top-k set of shortest s-t paths (if they exist).

Recall that, given a vertex $t \in V$. An *in-branching* T rooted at t is any sub-digraph of D that induces a (not necessarily spanning) tree containing t, such that every $u \in V(T) \setminus \{t\}$ has exactly one out-neighbor and t has no out-neighbor (that is, all paths go toward t). An in-branching T is called a *shortest path (SP) in-branching* rooted at t if, for every $u \in V(T)$, the length of the (unique) u-t path P^T_{ut} in T equals $d_D(u, t)$. As we consider directed weighted digraphs in this chapter, we use Dijkstra's algorithm to find an SP in-branching. However, it is possible to use any suitable shortest path algorithm instead.

In the forthcoming algorithms, the following procedure will often be used (and the key point when designing the algorithms is to limit the number of such calls and to optimize each of them). Given a sub-digraph H of D and $u, t \in V(H)$, we use an SP algorithm to compute an SP in-branching rooted in t that contains a shortest u-t path in H. Note that, the execution of an SP

algorithm may be stopped as soon as a shortest u-t path has been computed (when u is reached), i.e., the in-branching may only be partial (not necessarily spanning H). The key point will be that this way to proceed not necessarily only returns a shortest u-t path in H (if any) but an SP in-branching rooted in t, containing u. Recall that any such call has worst-case time complexity $O(m + n \log n)$.

Let $P = (v_0, v_1, \cdots, v_r)$ be any path in D and $i < r$. Any arc $a = v_i v' \neq v_i v_{i+1}$ is called a *deviation* of P at v_i. Moreover, any path $P' = (v_0, \cdots, v_i, v', v'_1, \cdots, v'_\ell = v_r)$ is called a *detour* of P at a (or at v_i). Note that neither P nor P' is required to be simple. However, if P' is simple, it will be called a *simple detour* of P at a (or at v_i). In addition, P' is called a shortest (simple) detour at v_i (or at a) if and only if P' is a detour with minimum length among all (simple) detours of P at v_i (or at a).

2.2.2 Yen's algorithm

We start by describing Yen's algorithm [127] trying to give its main properties and drawbacks.

All the algorithms described below start by computing a shortest s-t path $P_0 = (s = v_0, v_1, \cdots, v_r = t)$, and assume that there is always at least one such path. This is done by applying an SP algorithm from s. Note that P_0 is simple since weights are non-negative. A second shortest s-t simple path must be a shortest simple detour of P_0 at one of its vertices. Yen's algorithm computes a shortest simple detour of P_0 at v_i for every vertex v_i in P_0 as follows. For every $0 \leq i < r$, let $D_i(P_0)$ be the graph obtained from D by removing the vertices $v_0, \cdots, v_{i-1}$ (this is to avoid non simple detours) and the arc $v_i v_{i+1}$ (to ensure that the computed path is a new one, i.e., different from P_0). For every $0 \leq i < r$, an SP out-branching in $D_i(P_0)$ rooted at v_i is computed using an SP algorithm until it reaches t and therefore returns a shortest path Q_i from v_i to t. For every $0 \leq i < r$, the detour $(v_0, \cdots, v_{i-1}, Q_i)$ of P_0 at v_i is added to a set $Candidate$ (initially empty). Note that the index i (called below *deviation-index*) where the path $(v_0, \cdots, v_{i-1}, Q_i)$ deviates from P_0 is kept explicit (The deviation index of P_0 is 0) *, i.e, the path is stored with its deviation index. Once $(v_0, \cdots, v_{i-1}, Q_i)$ has been added to $Candidate$ for all $0 \leq i < r$, by remark above, a path with minimum length in $Candidate$ is a second shortest s-t simple path.

More generally, by induction on $0 < k' < k$, let us assume that a top-k' set S of shortest s-t paths has been computed and the set $Candidate$ contains a set of simple s-t paths such that there exists a shortest path $Q \in Candidate$ with $S \cup \{Q\}$ a top-$(k'+1)$ set of shortest s-t paths. Moreover, let us assume by induction that, for every path R in $Candidate$, with deviation index j, all detours of $R = (v_0, \cdots, v_{|R|})$ at vertices v_i for $0 \leq i < j$ have already been computed and added to $Candidate$. Yen's algorithm pursues as follows. Let $Q = (v_0 = s, \cdots, v_r = t)$ be any shortest path in $Candidate$ † and let $0 \leq j < r$ be its deviation-index. First, Q is extracted from $Candidate$ and it is added to S (as the $(k'+1)^{th}$ shortest s-t path). Then, for each vertex v in Q, a shortest simple detour of Q at v is added to $Candidate$ (since potentially one of these detours is a next shortest s-t path). For this purpose, for every $j \leq i < r$, let $\pi_i = (v_0, \cdots, v_{i-1})$ ($\pi_i = \emptyset$ if $i = 0$) and let $D_i(Q)$ be a subdigraph of D containing a shortest v_i-t path Q_i in D such that $Q_i \cap \pi_i = \emptyset$ and the path $\pi_i.Q_i$ is new ($\pi_i.Q_i \notin S$). After the construction of $D_i(Q)$ (described below), an SP out-branching of $D_i(Q)$ rooted at v_i is computed using an SP algorithm until it

∗. The deviation-index is not kept explicitly in Yen's algorithm. But, since it is a trivial improvement already existing in the literature [86], we mention it here.

†. Actually $Candidate$ is implemented using a heap, in such a way that extracting a shortest path in it takes logarithmic time and insertions are done in constant time.

reaches t and therefore returns a shortest path Q_i from v_i to t in $D_i(Q)$. For every $j \leq i < r$, the shortest simple detour $\pi_i.Q_i$ of Q at v_i (together with its deviation index i) is added to the set $Candidate$. This process is repeated until k paths have been found, i.e., when $k' = k$. Indeed, the computed detours of Q are distinct from every previously computed paths as they have different prefixes (this is the reason to keep explicitly the deviation-index).

The procedure of constructing $D_i(Q)$ is the following. First, to avoid non simple detours, i.e, any intersection between Q_i and π_i, the vertices $v_0, \cdots, v_{i-1}$ (if $i > 0$) are removed from D. Second, to ensure that the computed path $(\pi_i.Q_i)$ is new (different from those in S), each arc v_iv' such that S already contains a path with prefix $(v_0, \cdots, v_i, v')$ is removed from $D_i(Q)$.

Therefore, for each path Q that is extracted from $Candidate$, $O(|V(Q)|)$ calls of an SP algorithm are done. This gives an overall time-complexity of $O(kn(m + n \log n))$ which is the best theoretical (worst-case) time-complexity currently known (and of all algorithms described in this manuscript) to solve the kSSP problem.

For completeness, let us formally prove the correctness of Yen's algorithm in digraphs with non negative weights. It will allow us to show its correctness in digraphs with arbitrarily weights.

Claim 2.2.1. *Yen's algorithm correctly computes k shortest simple paths from s to t in a positive arc weighted directed graph.*

Proof. We proceed by induction on $0 < i \leq k$. For $i = 1$, the algorithm returns a shortest path from s to t. So, the algorithm is valid for $i = 1$.

Now suppose by induction that the algorithm returns correctly i shortest simple paths $\mathcal{P} = \{P_1, P_2, \cdots, P_i\}$. Let P_{i+1} be the $i + 1^{th}$ path reported by the algorithm and Q be a $i + 1^{th}$ shortest simple path in the digraph. Let us show that $\ell(Q) \geq \ell(P_{i+1})$.

Among the paths already returned by the algorithm until this step, let $P \in \mathcal{P}$ be a path that maximizes the length of a common prefix with Q, and then minimizes its deviation-index.

Let $\pi = Q \cap P = (s = u_0, \cdots, u_p)$ and let Q' be the subpath of Q from u_p to t, i.e, $Q = \pi.Q'$. When the algorithm iterates on each path in $\mathcal{P}$, it will iterate on the vertex u_p of P (as $P \in \mathcal{P}$), and a shortest u_p-t path Q^* is computed. Note that $\ell(Q^*) \leq \ell(Q')$ as Q^* is a shortest u_p-t path in a digraph where Q' lies. So, by maximality of the length of two common prefix with Q and by simplicity of Q, a path $P' = \pi.Q^*$ is added to the set $Candidate$. Since P_{i+1} is a smallest element in $Candidate$, we have $\ell(P_{i+1}) \leq \ell(P') = \ell(\pi.Q^*) = \ell(\pi) + \ell(Q^*)$ as $\pi.Q^*$ is simple. It follows that $\ell(P_{i+1}) \leq \ell(\pi) + \ell(Q') = \ell(Q)$. This proves the induction hypothesis for $i + 1$. □

Let us denote by Yen's framework the extension of Yen's algorithm when using any SP algorithm able to compute a shortest simple path in a digraph with arbitrary arc weights (i.e., when it is not possible to use Dijkstra's algorithm). Since no restriction is made on the length of the arcs in the proof of Claim 2.2.1, it follows that Yen's framework is valid for arbitrarily weighted digraphs.

Corollary 2.2.2. *Yen's framework solves the kSSP in digraphs with arbitrary arc weights.*

2.2.3 A Node Classification algorithm

In this section, we present the Node Classification (NC) algorithm, an improvement of Yen's algorithm proposed independently by Feng [54] and Gao et al. [59]. Same as Yen's algorithm, the NC algorithm is designed for digraphs with non-negative arc weights.

The most expensive part of Yen's algorithm is its large number of calls to an SP algorithm. The NC algorithm aims at reducing the computing time of each of these calls, and possibly to avoid some of them.

Precisely, during the process of finding a detour, the search area of an SP algorithm is restricted to a digraph that is smaller than D with the help of a precomputed shortest path in-branching. The NC algorithm starts by computing a shortest path in-branching T of D rooted at t (used to extract a first shortest path P_0). Then, when a path $Q = (v_0, \cdots, v_r)$ with deviation-index j is extracted, its detours are computed from $i = j$ to $r - 1$. The NC algorithm classifies the vertices as `red`, `yellow`, and `green` : a vertex on the prefix (i.e., the path $(v_0, \cdots, v_{i-1})$) is colored `red`, a vertex u that can reach t through T without visiting a `red` vertex (i.e, $P^T_{ut} \cap (v_0, \cdots, v_{i-1}) = \emptyset$) is colored `green`, and all other vertices are colored `yellow`. This coloring can be computed in linear time using a DFS in T. Moreover, the coloring used to compute the detour at v_{i+1} can be obtained faster by updating the coloring for the detour at v_i.

Another important ingredient of the NC algorithm is the notion of *residual length*. For each arc $e = uv$, the residual length of e is the cost of deviating from T at e. Precisely, it is the length of the path $u.P^T_{vt}$ minus the length of the path in P^T_{ut}. Formally, the residual length of arc uv is $\delta(u, v) = \ell(u, v) + \ell(P^T_{vt}) - \ell(P^T_{ut})$. The residual length is computed only once (after computing T) and remains valid till the end of the execution of the algorithm. Similarly, a residual length of a path is the sum of the residual length of thier arcs. Note that an arc in T has residual length equals 0, and so the residual length of the path P^T_{ut} from any `green` vertex u to t in T equals 0.

Recall that to compute a detour of Q at v_i, Yen's algorithm executes an SP algorithm to compute a shortest path from v_i to t in $D_i(Q)$. In the case of NC algorithm, Feng proved that it is sufficient to execute an SP algorithm with the residual lengths of the arcs and to stop its execution as soon as a green vertex is reached. This results in restricting the execution of the SP algorithm to the `yellow` subgraph that is expected to be smaller than $D_i(Q)$, and so to speed up the computation of the detours.

In Section 2.5, we propose an adaptation of the NC algorithm (using ideas of SB algorithm presented in the next section) that allows us to speed it up.

2.3 Sidetrack Based (SB) algorithm

We now present the Sidetrack Based (SB) algorithm, proposed by Kurz and Mutzel [83] solving the kSSP with the same settings of Yen's and NC algorithms, i.e, on digraphs with non-negative arc weights. We start by describing the data structure used in the SB algorithm. Then, we explain it and provide a pseudo code (Algorithm 2.1). Finally we analyse few aspects of it. Note that our contributions in Section 2.4 strongly rely on this algorithm and that is why we describe it in detail.

2.3.1 Compact representation of a path

The SB algorithm is based on a data structure generalizing the representation of a path proposed by Eppstein [50]. Such compact representation uses sequences of in-branchings $T_0, T_1, \cdots, T_h, T_{h+1}$ and deviations $e_0, e_1, \cdots, e_h$ (recall that a deviation of a path P is any arc not in P but with tail in P).

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 $0 \leq i \leq h$, 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 from the head w_0 of e_0 until it reaches the tail v_1 of e_1, *etc.* until it reaches the head w_h of e_h, plus (possibly) the 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 π 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 π.

2.3.2 The SB algorithm

We are now ready to present the SB algorithm, whose pseudocode is presented in Algorithm 2.1. Roughly, the SB algorithm uses a set $Candidate$ to manage candidate paths that are encoded using the above data structure. Sequentially, it extracts a shortest element ε from $Candidate$. If ε represents a simple path, this path is added to the output and the representations of its detours (that will be found using the last tree in the representation of ε) are added to $Candidate$. Otherwise, the SB algorithm attempts to modify ε by instantiating its last in-branching (see below). If this computation leads to a representation of a simple path, then it is added to $Candidate$. Otherwise, ε is discarded. The SB algorithm goes on iteratively until it has found k paths. The initialization consists in computing a first in-branching T_0 rooted at t in D (using an SP algorithm) and so a shortest s-t (simple) path $P_{st}^{T_0}$ and adding its representation to $Candidate$.

More precisely, the set $Candidate$ is a min-heap in which the length (the key) of an element is a lower bound on the length of the path it represents. Each element μ in $Candidate$ 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 length 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 $((T_0), \ell(P_{st}^{T_0}), \zeta = 1)$ is inserted in $Candidate$.

The SB algorithm iteratively extracts elements from $Candidate$ by minimum length (or lower bound) ‡ until k paths are obtained or $Candidate$ 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 $Candidate$, two cases are distinguished. Let i be the index of w_h in the path P represented by ε (note that i plays the same role as the deviation-index in the Yen's algorithm).

Case $\zeta = 1$. Then, ε represents a simple path $P = (v_0 = s, \cdots, v_i = w_h, \cdots, v_r = t)$ and all of its in-branchings have already been computed. In this case, the path P is added to the output. Then, for every deviation from P tailing at the suffix, i.e, for every deviation $e = (v_j, w)$ at v_j with $i \leq j < r$ (i.e, e is tailing at the suffix $(v_i, \cdots, t)$ of P), let $P_{wt}^{T_{h+1}}$ be a shortest path from w to t in T_{h+1} (if any) and let $Q(v_j, e) = (v_0, \cdots, v_j, P_{wt}^{T_{h+1}})$. If $Q(v_j, e)$ is simple, 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 $Candidate$ with $lb(e) = \ell(Q(v_j, e))$ 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 $Candidate$, where T' is the *name* of the in-branching of $D \setminus \{v_0, \cdots, v_j\}$ whose actual

‡. with a priority to representation of simple paths to break ties, i.e, if there is several elements with minimum lengths and one of them represents a simple path, we choose to extract it first.

computation is postponed, and $lb(e) = \ell(Q(v_j, e))$ is a lower bound on the length of the path represented by ε''.

Case $\zeta = 0$**.** In this case, the algorithm checks for the existence of a w_h-t path $P_{w_h t}^{T_{h+1}}$. To do so, the in-branching T_{h+1} (whose computation had 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 $P_{new} = (s = v_0, \cdots, v_h, P_{wt}^{T_{h+1}})$ is guaranteed to be simple. Moreover, P_{new} has length $\ell(P_{new}) = \ell((s = v_0, \cdots, v_h, w_h)) + \ell(P_{wt}^{T_{h+1}})$. Then, the representation $\mu' = (\varepsilon' = (T_0, e_0, \cdots, e_h = (v_h, w_h), T_{h+1}), \ell(P_{new}), \zeta = 1)$ is added to $Candidate$. Finally, if no w_h-t path can be found in T_{h+1}, μ is discarded.

Require: A digraph $D = (V, A)$, a source $s \in V$, a sink $t \in V$, and an integer k
Ensure: k shortest simple s-t paths

1: Let $Candidate \leftarrow \emptyset$, $\mathcal{T} \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
2: $T_0 \leftarrow$ an SP in-branching of D rooted at t containing s
3: Add $((T_0), \ell(P_{st}(T_0)), \zeta = 1)$ to $Candidate$
4: **while** $Candidate \neq \emptyset$ and $|Output| < k$ **do**
5: $\mu = (\varepsilon = ((T_0, e_0, \cdots, T_h, e_h = (v_h, w_h), T_{h+1}), lb, \zeta) \leftarrow$ an element in $Candidate$ with minimum length (or minimum lower bound lb)
6: **if** $\zeta = 1$ **then**
7: Extract μ from $Candidate$ and add ε to $Output$
8: **for** every deviation $e = w_j v'$ with $w_j \in P_{w_h t}^{T_{h+1}}$ **do**
9: $\varepsilon' \leftarrow (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1})$
10: $lb' \leftarrow lb - \ell(P_{w_j t}^{T_{h+1}}) + \ell(e) + \ell(P_{v' t}^{T_{h+1}})$
11: **if** ε' represents a simple path **then**
12: Add $\mu' = (\varepsilon', lb', \zeta = 1)$ to $Candidate$
13: **else**
14: $T' \leftarrow$ the name of an SP in-branching of $D_j(P)$ *// T' is not computed yet*
15: Add T' to $\mathcal{T}$
16: Add $\mu'' = (\varepsilon'' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T'), lb', \zeta = 0)$ to $Candidate$
17: **else**
18: **if** T_{h+1} has not been computed yet **then**
19: Compute T_{h+1}, an SP in-branching of $D_j(P)$ and add it to $\mathcal{T}$
20: Let $\mu' = (\varepsilon' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb + \ell(P_{v' t}^{T_{h+1}}) - \ell(P_{v' t}^{T_h}), \zeta = 1)$
21: Add μ' to $Candidate$
22: **return** $Output$

Algorithm 2.1 – Sidetrack Based (SB) algorithm for the kSSP [83]

Analysis. In a worst case scenario, each first extraction of a path from $Candidate$ leads to a non simple detour, and then a call of an SP algorithm. Note that no more than one call to an SP algorithm is done per vertex on a path, thanks to the checks of line 18. So, the complexity of the SB algorithm is bounded by $O(kn(m + n \log n))$ as the number of vertices of a simple path is bounded by n and the algorithm stops once k paths are added to $Output$.

There are two key improvements for which the SB algorithm has, in practice, out-performed all other algorithms for solving the kSSP problem so far. First, it saves an SP algorithm call if the detour is simple. Second, if the detour is not simple it is inserted with a lower bound on its length and the corresponding call to an SP algorithm is postponed. This way, if this detour leads to a long path (path with length, or length's lower bound, larger than the k^{th} shortest path), the call to an SP algorithm will never be performed.

However, as the SB algorithm stores complete in-branchings into memory, it has the drawback to possibly have an important consumption of working memory (much more than the NC algorithm that stores a single in-branching, while keeping the whole description of paths it computes).

2.3.3 The SB* algorithm

Here, we propose the SB* algorithm, a variant of the SB algorithm that is a tiny modification of the SB algorithm but leading to a significant speed up (see Section 2.6.2).

In fact, each time a representation $(T_0, e_0, T_1 \cdots, e_{h-1} = (u_{h-1}, v_{h-1}), T_h, e_h = (u_h, v_h), T_{h+1})$ is extracted from $Candidate$ with $\zeta = 0$ and that T_{h+1} has not been computed yet (i.e., it is only a pointer), our algorithm does not compute T_{h+1} from scratch as the SB algorithm does. Instead, the 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 SP in-branching T using standard methods for updating a shortest path tree [57]. Then, the pointer T_{h+1} is associated with the new in-branching T.

It is clear that the SB* algorithm computes (and stores) exactly the same number of in-branchings as the SB algorithm. The computational results presented in Section 2.6.2 show that this update procedure gives an average speed up by a factor of 1.5 to 2 on road networks.

2.4 Space - time tradeoffs

2.4.1 The Parsimonious Sidetrack Based algorithm

Here, we present the Parsimonious Sidetrack Based (PSB) algorithm, which is an adaptation of the SB algorithm allowing to reduce the memory consumption due to the storage of all in-branchings computed by the SB algorithm. Here, we only focus on the differences between the SB and the PSB algorithm.

The main difference is that PSB algorithm stores two types of elements in $Candidate$. 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 length lb. Contrary to the 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 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 $Candidate$. T_{h+1} is computed at this step (if not already done) which allows to get P explicitly (note that in this case lb equals the length of P). Then, PSB algorithm adds $P = (s = v_0, \cdots, v_i = w_h, \cdots, v_r = t)$ to $Output$ and (as the SB algorithm), for every $v \in \{v_i, \cdots, v_r\}$, and every deviation e with tail v, the detour $Q(v, e)$ of P at e is considered. If $Q(v, e)$ is simple, then $\mu' = ((T_0, e_0, T_1, e_1, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1}), \ell(Q(v, e)))$ is added to $Candidate$. 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 $Candidate$, where

$lb' = \min_{f_j=(u_j,u'_j)\in Dev} \ell(Q(u_j, f_j))$. That is, Dev is the set of all "non simple deviations" of P at the vertices between w_h and t, ordered with respect to the index of their tail on P, i.e, for two deviations $f_i = (u_i, u'_i), f_j = (u_j, u'_j) \in Dev$, $f_i \leq f_j$ if and only if $i \leq j$. Finally, let lb' be a lower bound on the length of the detours at a deviation in Dev. The important difference between SB and PSB algorithms comes from the fact that non simple detours are considered as a unique object by PSB algorithm.

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 $Candidate$. 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 $lb = \ell(Q(u_{min}, f_{min}))$.

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 $P^{T'_j}_{u'_j t}$ from u'_j to t is discovered (if no such path exists, j is decreased by one). If $P^{T'_j}_{u'_j t}$ 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 length $lb_j = \ell((v_0, \cdots, v_{i_j})) + \ell(f_j) + \ell(P^{T'_j}_{u'_j t})$. Then, the element $\mu_j = (\varepsilon_j, lb_j)$ is added to $Candidate$, but T'_j is not stored (PSB algorithm might have to recompute it later). A second key improvement is that to speed up the computation, T'_j is actually computed by updating T'_{j+1}. which is done using standard tools from [57]. Then, only when $j = min$, the in-branching T'_{min} is stored and $\mu_{min} = (\varepsilon_{min}, lb_{min})$ is added to $Candidate$. The reason why T'_{min} is stored (while other T'_j are not) is that μ_{min} is expected to be extracted soon from $Candidate$ (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 $Candidate$, where lb' is the minimum length over the non simple detours in Dev'.

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.

2.4.2 Special variants of PSB

A better space and time trade-off than PSB can be achieved if each computed and stored in-branching is going to be used in the future steps, i.e, it is used to extract a simple candidate path that is going to be extracted from $Candidate$ (before the k^{th} shortest path). Unfortunately, such information cannot be afforded as the length of the k^{th} path is not previously known. However, if computing an in-branching leads to constructing a path with length relatively short, say less than a threshold value times the current outputted path, then storing such in-branching is meaningful as the extraction of its corresponding element form $Candidate$ is expected soon and saving it leads to save its redundant computation. Here, we present two variants of the PSB ; PSB-v2 and PSB-v3. PSB-v2 is a tiny improvement of PSB leading to consume less memory by storing less in-branching while PSB-v3 gives an adaptable trade off depending on the value of the threshold.

Let us consider, again, a step when PSB algorithm extracts an element $\mu = (\varepsilon, Dev = \{f_1, \cdots, f_j = (u_j, u'_j), \cdots, f_l\}, lb)$ from $Candidate$ with $1 \leq min \leq l$ the smallest integer such that $lb = \ell(Q(u_{min}, f_{min}))$. The PSB algorithm algorithm iterates on j, decreasing from l to min as explained above, a corresponding in-branching T'_j is computed for each j (but not stored). Then, only when $j = min$, the in-branching T'_{min} is stored.

The PSB-v2 algorithm does not naively store T'_{min}. Instead, T'_{min} is stored only if the length of its corresponding detour is less than a threshold value θ, times the length of the shortest simple path in $Candidate$. That is, $lb_{min} = \ell((v_0, \cdots, v_{i_{min}})) + \ell(P^{T'_{min}}_{u_{min}t}) \leq \theta * \ell(P_{next})$, where P_{next} is the shortest simple path in $Candidate$. As a result, if the in-branching T'_{min} leads to a (relatively) long path that is not expected to be extracted very soon from $Candidate$, then it is freed from the memory.

PSB-v3 behaves on every deviation in Dev between min and l the same way PSB-v2 behaves with f_{min}. Precisely, for each deviation f_j with $min \leq j \leq l$, PSB-v3 computes its corresponding in-branching T'_j. This in-branching (T'_j) is stored only if the length of its corresponding detour is less than a threshold θ, times the length of the shortest simple path in $Candidate$.

The value of the threshold θ could change dynamically during the execution. For instance, it could be related to the ratio between the two upcoming paths i.e, the two elements in $Candidate$ with minimum length.

2.5 Postponing the detours's computation

In this section, we present the Postponed Yen (PY) algorithm. The PY algorithm have a $O(kn(m + n \log n))$ time complexity with a similar working memory as the NC algorithm. However, it is faster in practice than the NC algorithm.

Even though the NC algorithm consumes less time during each SP algorithm call than Yen's algorithm, the total number of calls remains equal to Yen's algorithm. Here, with the help of a lower bound on the length of a simple detour, we propose (using a similar idea as the SB algorithm) to postpone the calls in order to avoid some of them. We prove that such postponement does not hurt the correctness of the algorithm.

Let us describe our PY algorithm.

As in the NC algorithm, our algorithm starts by computing an SP in-branching T_0 rooted at t that will be used throughout the execution of the algorithm. Then, as the NC algorithm, the PY algorithm proceeds by phases where a new path is added to the output and its detours are computed and added in the set $Candidate$. Our algorithm differs from the NC algorithm in how and when it computes the detours of the paths but also in the structure of the elements in the heap $Candidate$.

Let us consider a phase when a s-t path $P = (s = u_0, u_1, \cdots, u_i, \cdots, u_{r-1}, u_r = t)$ is extracted from $Candidate$. Let $0 \leq i < r$ and consider the step when a shortest simple detour of P at u_i is computed. Let N be the set of neighbors v of u_i such that paths with prefix $(u_0, \cdots, u_i, v)$ have already been added to $Output$. Let $D' = D \setminus \{(u_i, v) \mid v \in N\}$ and let $D_i(P) = D' \setminus (u_0, \cdots, u_{i-1})$.

Let us describe how the PY algorithm finds a new shortest simple detour P' of P at u_i. Recall that a detour P' is said new if P' has not been added to the $Output$ yet. Let $v_{LB} \in D_i(P)$ be the neighbor of u_i (neither in N nor in the prefix of P) such that the residual length of (u_i, v_{LB}) is minimum, i.e., $\delta(u_i, v_{LB}) \leq \delta(u_i, v')$ for every $v' \in N^+_{D_i(P)}(u_i)$ (recall that $\delta(u, v) = \ell(u, v) + \ell(P^{T_0}_{vt}) - \ell(P^{T_0}_{ut})$ denotes the residual length of arc uv as defined in Section 2.2.2). Note that, by definition of the residual length, the path $P_{LB} = (s, u_1, \cdots, u_i, P^T_{v_{LB}t})$ is a shortest new detour (not necessarily simple) of P at u_i, so in particular :

Claim 2.5.1. *$\ell(P_{LB}) \leq \ell(P')$ for any new simple detour P' of P at u_i*

Input A digraph $D = (V, A)$, source $s \in V$, sink $t \in V$ and an integer k
Output k shortest simple s-t paths
Let $Candidate \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
$T \leftarrow$ an SP in-branching of D rooted at t
Add $(P_{st}(T), \ell(P_{st}(T)), 0, 1)$ to $Candidate$
while $Candidate \neq \emptyset$ and $|Output| < k$ **do**
    $(P = (s, u_1, \cdots, t), \ell(P), i, \zeta) \leftarrow$ extract a shortest element from $Candidate$
    Color the vertices `yellow`, `red`, `green` with respect to P and u_i
    $\pi \leftarrow (s, u_1, \cdots, u_{i-1})$
    $Dev_{old} = \{e = (u_i, v)$ s.t there is a path in $Output$ having $\pi.e$ as prefix$\}$
    if $\zeta = 1$ (P is simple) **then**
        add P to $Output$
        for each vertex u_j in $(u_i, \cdots, t)$ **do**
            $(u_j, v_{LB}) \leftarrow$ an arc in $A \setminus Dev_{old}$ with minimum δ among those tailing at u_j
            $P_{LB} \leftarrow (s, \cdots, u_j, v_{LB}, P^T_{v_{LB}t})$
            $\zeta' \leftarrow 0$
            if v_{LB} is `green` **then**
                $\zeta' \leftarrow 1$
            add $(P_{LB}, \ell(P_{LB}), j, \zeta')$ to $Candidate$
    else
        Compute a shortest u_i-t path Q in $D' = (V \setminus \pi, A \setminus Dev_{old})$
        if Q exists **then**
            Add $(P' = \pi.Q, \ell(P'), i, 1)$ to $Candidate$
return $Output$

Algorithm 2.2 – Postponed Yen Algorithm (PY)

Similarly to the SB algorithm (and in contrast to the NC algorithm), PY algorithm may add non-simple paths to the set $Candidate$. Precisely, each element in $Candidate$ has the form $(P = (s = u_0, \cdots, u_r = t), \ell(P), i, \zeta)$ where i is its deviation index and ζ is a boolean flag indicating whether the path P is simple or not.

The main idea of the PY algorithm (Algorithm 2.2) is the following. Instead of computing naively all the shortest simple detours of P, i.e, a shortest simple detour at u_j for all $i \leq j < r$, the following procedure is applied. For each vertex $u_j \in P$, the SP in-branching T_0 is colored (`yellow`, `red`, `green`, as in the NC algorithm) with respect to P and u_i. If the color of v_{LB} is `green`, it implies that the path P_{LB} is simple, and a shortest simple detour is found (by the remark above). In this case, $(P_{LB}, \ell(P_{LB}), i, \zeta = 1)$ is added to the set $Candidate$. Otherwise, i.e., if v_{LB} is `yellow`, the detour P_{LB} is added to the set $Candidate$ (even though it is not simple) with its length $\ell(P_{LB})$ as a key, i.e, the element $(P_{LB}, \ell(P_{LB}), i, \zeta = 0)$ is added to $Candidate$. The idea is that, in the latter case, the non-simple path added to $Candidate$ may never be extracted from $Candidate$ and so a call to an SP algorithm is saved.

When an element $(P, \ell(P), i, \zeta)$ is extracted from $Candidate$. If $\zeta = 1$, the simple path P is added to the $Output$ and its detours are added to $Candidate$ as explained above. If not, i.e., P is not simple, it will be "repaired" into a simple path and re-added to $Candidate$. More precisely, after the extraction of P from $Candidate$, an SP algorithm is called to find a shortest (simple) path Q from u_i to t in $D_i(P)$ and P is replaced by $P' = (s, u_1, \cdots, u_{i-1}, Q)$. Claim 2.5.1 ensures that the order of extraction of the simple paths from $Candidate$ remains valid. And finally, such postponement of this SP algorithm call may end up by skipping it.

2.6 Experimental evaluation

In this section we describe our experimental evaluation. First, we start by describing our implementation and settings (Section 2.6.1), then we discuss our experimental results on road and complex networks (Section 2.6.2). Finally, section 2.6.3 contains an experimental study to some parameters related to the queries.

2.6.1 Experimental settings

Here we specify the details of the implementation and the setting used in our experiments.

We have implemented [§] all the algorithms presented in this thesis (Yen, NC [54], PY, SB [83], SB* and PSB) in C++ and our code is publicly available [13].

Following [83], we have implemented a pairing heap data structure [56] supporting the decrease key operation, and we use it for Dijkstra's shortest path algorithm. Our implementation of the Dijkstra shortest path tree algorithm is lazy, that is, it stops computation as soon as the distance from query node v to t is proved to be the shortest one. Further computations might be performed later for another node v' at larger distance from t starting from this partial shortest path tree already computed. Our implementation of Dijkstra's algorithm supports an update operation when a node or an arc is added to the graph. Moreover, we have implemented a special copy operation that enables to update the in-branching when a set of nodes is removed from the graph. This corresponds to the operations performed when creating an in-branching T_{h+1} from T_h in SB*.

§. Despite several queries, we have not been granted access to the code used for experiments in [54, 83].

Observe that in our implementations the parameter k is not part of the input, and so the sets of candidates are simply implemented using pairing heaps. This choice enables to use these methods as iterators, able to return the next shortest path as long as one exists. Note that, if k is part of the input, the data structure used to store candidates could be changed in order to contain only the k best candidates, but the algorithm would only return exactly k paths even if more exist. Moreover, for the SB, SB*, PSB, PSB-v2 and PSB-v3 algorithms, following [83], we store the candidates into two heaps, the first one to store the simple candidates ($Candidate_{simple}$) and the second one to store the non-simple candidates ($Candidate_{not-simple}$). Then, we extract candidates from $Candidate_{simple}$ as long as the length of the shortest simple path is smaller or equal to the length of the shortest non-simple path in $Candidate_{not-simple}$. This way, we prioritize the extraction of simple paths.

Concerning the PSB-v2 and PSB-v3 algorithms, based on preliminary experiments, we choose to update the value of θ dynamically with respect to the ratio between the pair of the upcoming paths. Recall that when looking for the i^{th} path, a corresponding in-branching will be stored only if the length of that path is at most θ times the length of the $(i-1)^{th}$ path.

Precisely, let ℓ_s be the length of the smallest element in $Candidate_{simple}$ and let ℓ_{ns} be the length of the smallest non-simple element in $Candidate_{not-simple}$. These values are both set to 1 if any of the corresponding sets is empty. Let also $c = \max(\frac{\ell_s}{\ell_{ns}}, \frac{\ell_{ns}}{\ell_s})$ and $\rho = c - 1$. The value of θ is set to $1 + \alpha\rho$, for some constant $\alpha > 0$. The intuition is to store an in-branching only if it is expected to be used soon, that is, while extracting one of the upcoming paths. The choice of these values (ℓ_s and ℓ_{ns}) gives us a meaningful indication, and the value of θ is easy to compute. Observe that in our experiments we have set the factor α in the formula for computing θ to $\alpha = 11$, based on preliminary experiments.

Besides, a special implementation of the PY algorithm is proposed, this implementation is inspired by the NC algorithm and it is called the Postponed Node Classification (PNC) algorithm. As shown in Algorithm 2.2, the PY algorithm computes a shortest simple detour (line 21) with an SP algorithm call that may visit the whole sub-digraph. Similarly to NC algorithm, the PNC algorithm tries to reduce the size of this sub-digraph in order to speed up these calls. For this purpose, it proceeds the same way as the NC algorithm. That is, it gives a color to each node (vertex) with respect to its end in the first pre-computed in-branching and call an SP algorithm visiting only the `yellow` vertices.

Networks setting We have evaluated the performance of our algorithms on some road networks from the 9th DIMACS implementation challenge [44] and on several complex networks.

A road network of a city is the digraph modeling its roads, i.e, a vertex is associated to each crossroad, and there is an arc of length w between two vertices if and only if there is a road of physical length w (in km) between their corresponding crossroads. Road networks are known to be sparse, almost planar and to have a bounded degree [124]. We denote by small road networks the road network of ROME, DC and DE while big road networks denote NY, BAY and COL. The characteristics of these graphs are reported in Table 2.1.

On the other side, complex networks model different types of networks. Generally, they are characterized by being small-world, i.e, they have a logarithmic diameter, by a power-law degree distribution and a high clustering coefficient [25]. For instance, BIOGRID system synthetic lethality represents mutation/deletion of genes resulting in lethality when combined in a same cell [97]. DIP represents protein to protein interactions [102]. The FB network represents social circles from Facebook [89]. Likewise, LOC is a graph provided from Brightkite location-based social networ-

network	n	m	D	$\langle d \rangle$	$-\alpha$	$\langle cc \rangle$	Description
ROME	3 353	8 870	57	5.2	-	0.025	Road network of Rome [44]
DC	9 559	29 682	140	6.2	-	0.039	Road network of Washington DC [44]
DE	49 109	119 520	573	4.8	-	0.024	Road network of Delaware [44]
NY	264 346	733 846	664	5.5	-	0.02	Road network of New York [44]
BAY	321 270	800 172	791	4.9	-	0.016	Road network of San Francisco Bay area [44]
COL	435 666	1 057 066	1219	4.8	-	0.017	Road network of Colorado area [44]
BIOGRID	2 318	12 580	7	21.7	1.96	0.20	Mutation/deletion of genes resulting in cell lethality [97]
FB	3 698	85 963	6	93	-	0.61	Social circles from Facebook [89]
P2P	5 606	23 510	8	16.7	-	0.014	Peer-to-peer network of the Gnutella file sharing network [88]
DIP	13 969	60 621	17	17.4	2.38	0.11	Protein-protein interaction network [102]
CAIDA	29 432	143 000	9	19.4	2.06	0.42	Relationships between Autonomous Systems of the Internet [116]
LOC	33 187	188 577	11	22.7	2.25	0.29	Brightkite location-based social networking service provider [88]

Table 2.1 – Characteristics of the graphs used in kSSP experiments : number of nodes (n), number of edges (m), diameter (D), average degree ($\langle d \rangle$), exponent $-\alpha$ of the power-law degree distribution, and average clustering coefficient ($\langle cc \rangle$).

king [88]. Finally, P2P is the peer-to-peer network of the Gnutella file sharing network [88] and CAIDA (2013.11.01) is the graph of the relationships between the autonomous system of the Internet [116]. As these networks are unweighted, we only consider the number of hops as the length of a path. We consider for each network its largest biconnected component. The characteristics of these graphs are depicted in Table 2.1

In our experiments, we have randomly chosen 1000 queries (source-destination pairs of vertices) for each network, and we have run each algorithm for each of these pairs for k up to $1,000$ on road networks and k up to $10,000$ on complex networks. Because of the excessive running time of Yen's algorithm, we have chosen to run it only on small road networks.

We have measured the execution time and the number of stored SP in-branchings. Note that the number of stored in-branchings gives an indication of the memory consumption that is independent of the implementation and the architecture of the machine [77].

We also attempt in Section 2.6.2 to explain the performance of the algorithms with respect to the structure of the network and / or some properties of the queries (e.g., the hop distance between the source and the destination, their stretch from the "center" of the graph...).

All reported computations have been performed on computers equipped with 2 quad-core 3.20GHz Intel Xeon W5580 processors and 64GB of RAM.

2.6.2 Experimental results

In this section, we first describe and analyze our experimental results on road networks and then on complex networks. We will see that the behavior of the algorithms' differ from one type of network to the other.

First, we have measured the average and the median of the algorithms' running time in all considered networks. The results on road networks are reported in Table 2.2 and the ones on complex networks are in Table 2.4. The data in Tables 2.2, 2.4 and 2.3 corresponds to the biggest experienced value of k ($k = 1,000$ for road networks and $10,000$ for complex network).

Then, we have measured the number of stored trees for all networks. As we will discuss below, in the case of complex networks, there are very few differences in the number of in-branchings. The results on road networks are described in Table 2.3. Moreover, in Figures 2.3 and 2.4, we report the evolution of the average and median running times of the algorithms when the number

k of reported paths increases. For the latter figures, the results we obtained differ depending on the class (road or complex) of the networks but among a same class, they do not differ much depending on the considered networks, so we only report them in the case of the networks COL, DC (see Figures 2.1 and 2.2), BIOGRID and CAIDA (see Figure 2.4).

We have then performed a refined comparison of the algorithms on the road networks. The results we obtained does not differ much depending on the considered networks, so we only report the comparison of the algorithms on the DC and COL networks in Figures 2.1 and 2.2. More precisely, we have plot pairwise comparisons of the running times and number of stored trees for each source-destination pairs on DC and COL network.

Finally, some statistics about the queries' properties and their impact on the algorithms' performances are depicted in Figures 2.5 and 2.6.

Road Networks. Note that the algorithms can be classified as three sets : Algorithms with low memory consumption i.e, the ones storing no more than a single in-branching, that are Yen's, NC, PY and PNC. Algorithms storing big number of in-branchings (SB and SB*). And algorithms establishing a space - time tradeoff (PSB and its two variants). We first compare the experimental results of the first class together, then we do the same with the second class. We finally compare the PSB algorithm with the others, and we give a conclusion on our experiments on road networks.

— We first observe, based on Table 2.2 and Figures 2.3a and 2.3b, that all algorithms are faster than Yen's algorithm on small road networks (the average speed up is between one and two orders of magnitude). Similar experiments described in [54, 83] leads to the same conclusion on big road networks.

  Let us now compare NC, PY and PNC algorithms together, as they all store no more than a single in-branching in the memory. The average and median running times reported in Table 2.2 show that the PY algorithm is significantly faster than the NC algorithm for all road networks (up to 5 times faster on average). Moreover, a refined comparison of the NC and PY algorithms on DC and COL networks (Figures 2.1b and 2.2b) show that this is true for all queries.

  Surprisingly, PNC algorithm is slower than PY (based on Table 2.2 and Figures 2.1d and 2.2d) even though its was expected to be faster (see Section 2.6.1). This is due to the extra time consumed during the coloring procedure that is computed from scratch each time (unlike PY where no such operation is needed).

— The simulation results reported in Table 2.2 and Figure 2.3 confirm that the use of shortest path tree update procedures in SB* helps to significantly reduce the running time compared to SB. More precisely, the average and median running times of the SB* algorithm are significantly smaller than for the SB algorithm on all road networks (SB* is up to twice faster than SB). A more refined comparison on DC and COL network (Figures 2.1a and 2.2a) show that SB* is faster than SB for almost all queries (more than 90% of the queries). The same behavior was observed on all road networks. Finally, we recall that by design, the number of stored in-branchings is the same in both algorithms.

— Concerning the PSB algorithm, as shown in Table 2.2 and Figures 2.1e, 2.1f, 2.2e and 2.2f, PSB gives a space time tradeoff between SB and NC i.e, it is faster than NC and consumes less memory than SB. However, PSB algorithm is, sometimes, beaten by the PY algorithm as it is faster while consuming less memory (only one in-branching is stored). As a result, PSB algorithm could give a space time tradeoff only on DC and BAY. Note that, the special two variants of PSB (PSB-v2 and PSB-v3) lead to a small improvement of the running time

(up to 5% time reduction) but to a significant reduction of the memory consumption (up to 30% memory reduction), see Tables 2.2 and 2.3.
An unexpected observation (in Table 2.2) is the gap between the average and the median running time of the SB like algorithms (SB, SB*, PSB, PSB-v2 and PSB-v3). That is, the median could be up to 5 times smaller than the average, while it is not the case with the remaining algorithm. We discuss this observation further in Section 2.6.3.
Among the considered algorithms, none of them clearly outperforms all the others on road network. As shown in Table 2.2 and Figure 2.3, on most of the road networks (Rome, DE, BAY and COL), PY algorithm is the fastest on average (slightly faster than SB*). However, SB* is, by far, the fastest on median on DC, NY, BAY and COL. This is justifiable by the big difference between the average and the median running time of SB* algorithm and by the fact that, on a small number of queries, SB* algorithm is extremely slow, as shown in Figures 2.1c and 2.2c where a small number of points are very far from the majority of the other points. Note that the results may change with respect to the value of k (Figure 2.3).

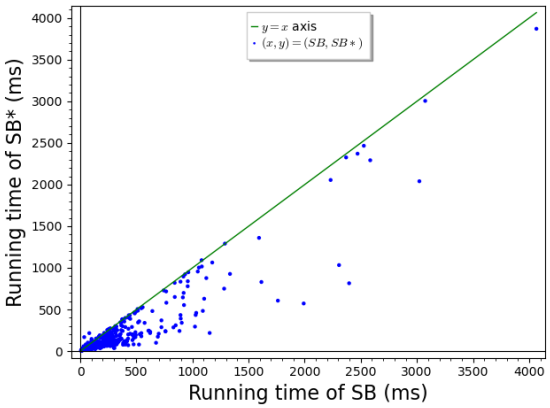
To conclude, among the considered algorithms, PY and SB* algorithms are the fastest on road networks. SB* has a better median running time while PY algorithm has a running time that is more stable.

		Rome	DC	DE	NY	BAY	COL
Yen	avg	3389	11316	159785	-	-	-
	med	1439	4945	59463	-	-	-
NC	avg	407	823	10620	99521	94149	146025
	med	178	404	6129	64465	56136	99540
PY	avg	**181**	326	**1972**	41923	**24970**	**35265**
	med	**155**	299	**1644**	39389	24064	35039
PNC	avg	203	336	2434	69913	28481	40045
	med	173	305	1997	58446	26552	38952
SB	avg	451	184	8469	53423	38390	68077
	med	356	75	4321	30300	8783	20535
SB*	avg	282	**117**	5428	**33704**	28693	49859
	med	199	**43**	2139	**18659**	**4977**	**11060**
PSB	avg	447	269	7939	106040	53286	81321
	med	340	117	6148	81927	23574	40640
PSB-v2	avg	447	265	7513	100377	49683	76766
	med	347	117	5849	75914	21812	38732
PSB-v3	avg	446	265	7471	100390	49653	77185
	med	346	115	5785	75681	21770	38709

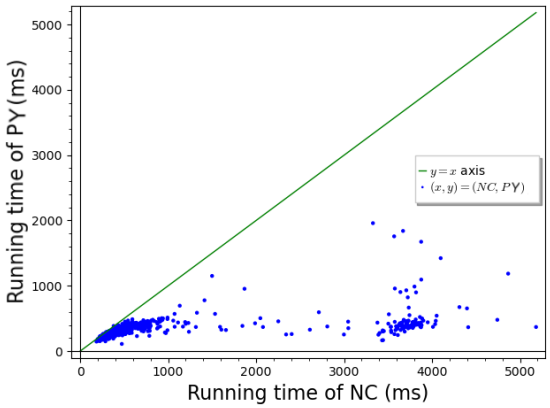
Table 2.2 – Running time (ms) of the algorithms on road networks, ($k = 1,000$)

Complex Networks. Here we analyze, and we try to explain the behavior of the algorithms on complex networks, except for Yen's algorithm because of its excessive running time (as already mentioned above).

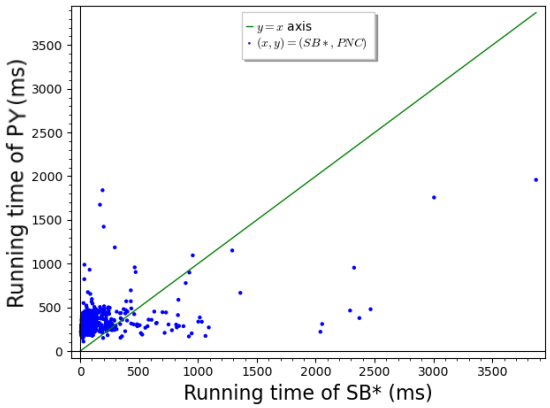
On complex networks, the PSB algorithm is the fastest kSSP algorithm among the considered algorithms (Table 2.4 and Figure 2.4). Considering the space consumption, all the kSSP algorithm



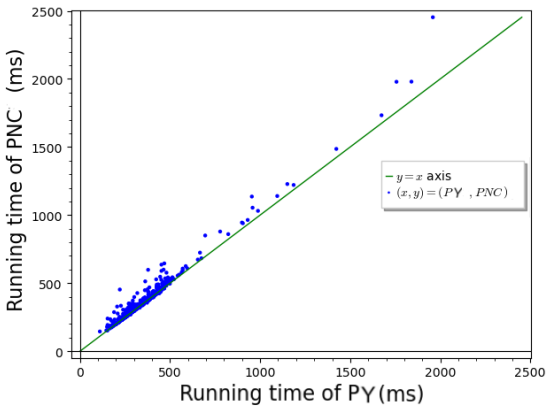
(a) Running time of SB and SB*



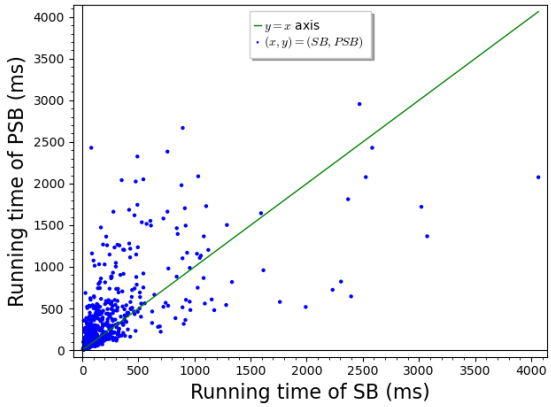
(b) Running time of NC and PY



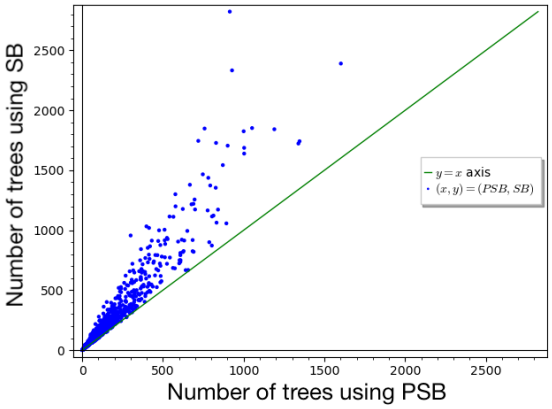
(c) Running time of SB* and PY



(d) Running time of PY and PNC

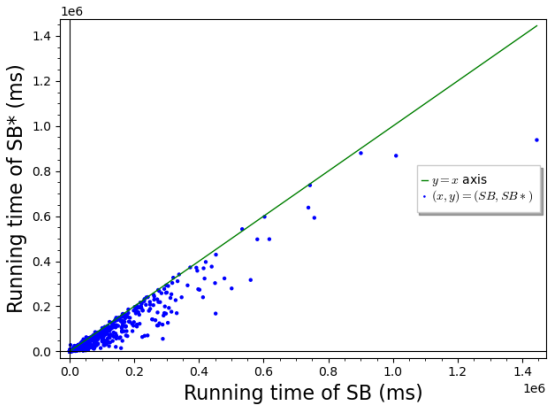


(e) Running time of SB and PSB

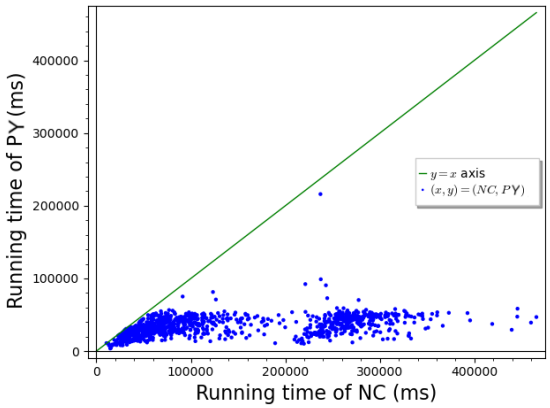


(f) Number of trees of SB and PSB

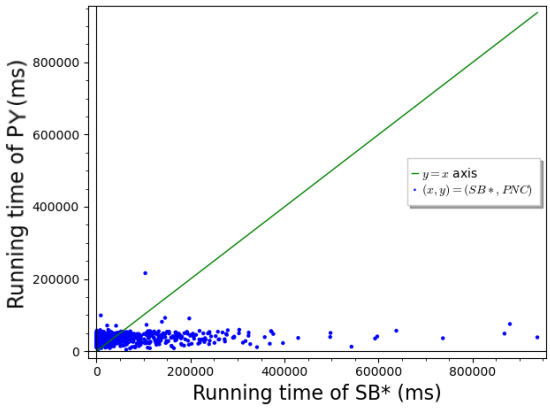
Figure 2.1 – Comparison of the running time and the number of stores trees on DC. Each dot corresponds to one pair source/destination ($k = 1,000$).



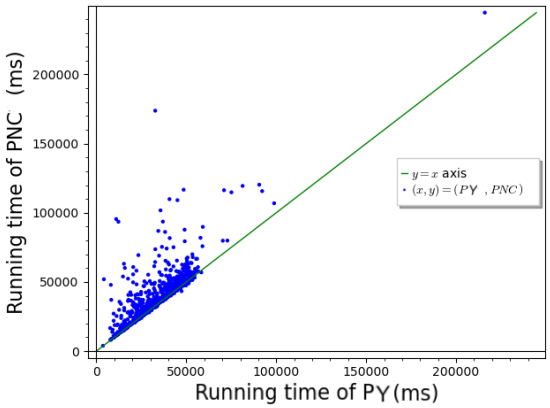
(a) Running time of SB and SB*



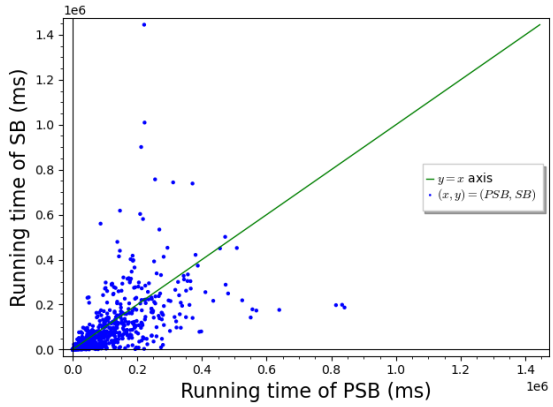
(b) Running time of NC and PY



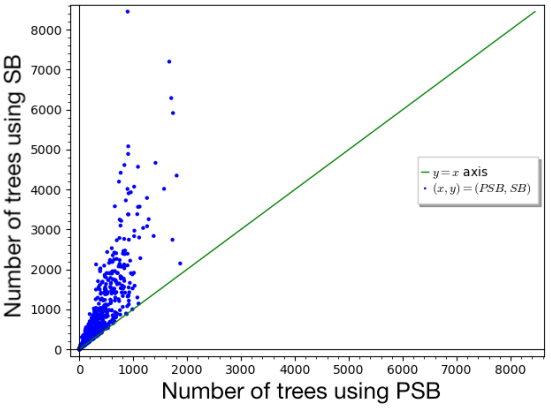
(c) Running time of SB* and PY



(d) Running time of PY and PNC



(e) Running time of PSB and SB



(f) Number of trees of SB and PSB

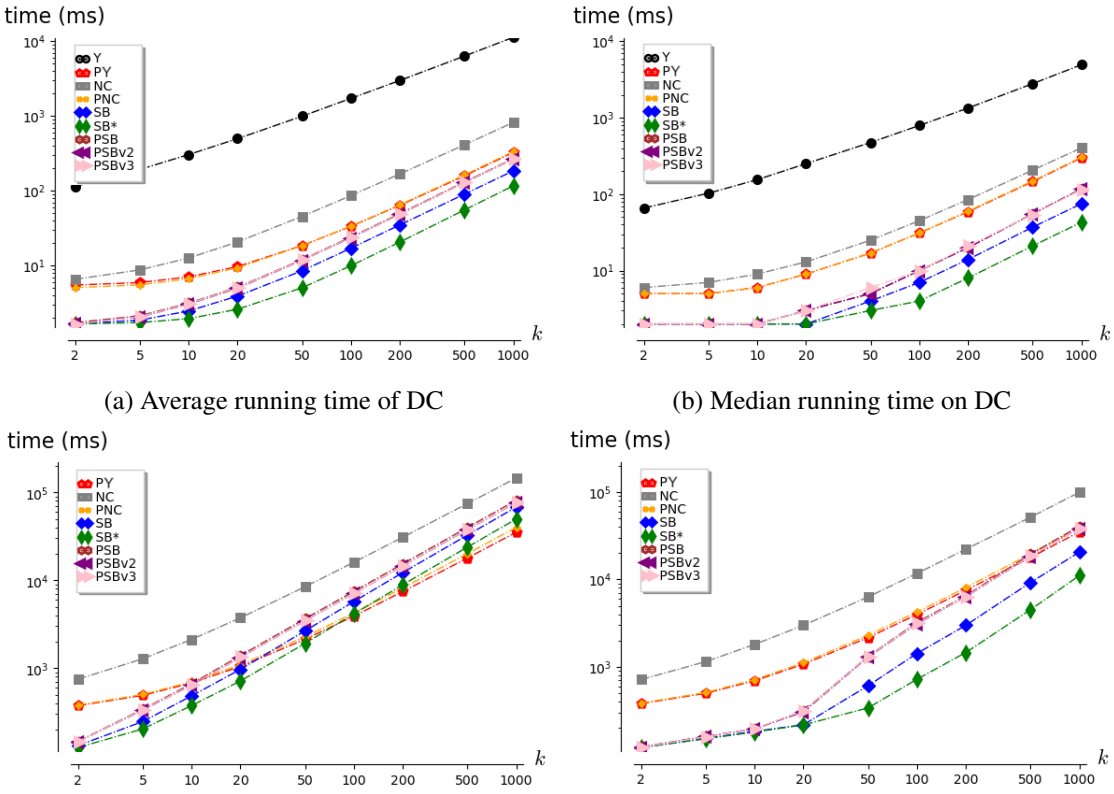
Figure 2.2 – Comparison of the running time and the number of stores trees on COL. Each dot corresponds to one pair source/destination ($k = 1,000$).

	Rome	DC	DE	NY	BAY	COL
NC, PY and PNC	1	1	1	1	1	1
SB and SB*	1135	243	948	1669	541	585
PSB	694	160	335	536	246	249
PSB-v2	**580**	**138**	**274**	**373**	**194**	**208**
PSB-v3	588	147	290	380	200	212

Table 2.3 – Average number of stored trees using some kSSP algorithms on road networks, ($k = 1,000$)

		BIOGRID	FB	P2P	DIP	CAIDA	LOC
NC	avg	1905	1493	3247	8325	25367	26659
	med	1458	1442	3014	7590	18583	23846
PY	avg	1336	1523	2475	6361	22459	21474
	med	1303	1492	2386	6354	21824	21065
PNC	avg	1302	1478	2479	6148	21869	18910
	med	1281	1449	2396	6087	21122	18332
SB	avg	993	2986	596	870	8637	2524
	med	821	2472	575	771	6724	2200
SB*	avg	980	3070	481	802	8506	2508
	med	810	2507	476	725	6661	2173
PSB	avg	431	**1062**	293	456	3644	1237
	med	394	**971**	282	422	3666	1190
PSB-v2	avg	**421**	1082	294	439	**3541**	**1163**
	med	**387**	988	282	410	**3574**	**1122**
PSB-v3	avg	431	1081	**292**	**437**	3654	1210
	med	399	991	**280**	**410**	3664	1167

Table 2.4 – Running time (ms) of the algorithms on Complex networks, ($k = 10,000$)



(a) Average running time of DC

(b) Median running time on DC

(c) Average running time of COL

(d) Median running time on COL

Figure 2.3 – The running time of the kSSP algorithms on road network with respect to the values of k

have a small memory consumption (for $k = 10,000$, the number of stored in-branchings does not exceed 50). It is also shown in Table 2.4 that the running time of PSB and its two variants (PSB-v1 and PSB-v2) is similar.

In what follows, we give some qualitative arguments that may explain the fact that the PSB algorithm is the fastest among all considered algorithms on complex networks.

Suppose P is a shortest path from s to t. On complex networks, it is likely to have a vertex v on P with high degree. Let us study the behavior of the different variants of the PSB algorithm when they compute the detours of P at v, in contrast with the other algorithms.

First, Yen's, NC, PY and PNC algorithms may compute, independently, for each vertex $v' \in N^+(v)$ neighbor of v a shortest path from v' to t resulting with $|N^+(v)|$ shortest path algorithm calls to find the shortest simple detours at the neighbors of v. On the other hand, SB* and PSB algorithms compute at most one shortest path in-branching T at v, that works for each neighbor v' of v. In another word, SB* and PSB are favorable to iterate on v. Moreover, as v has a high degree, it is supposed that a large number of the neighbors v' of v leads to simple candidates, i.e, $P^T_{v't} \cap (s, \cdots, v) = \emptyset$. So, the number of shortest path in-branchings computed and/or stored using SB* and PSB algorithms is expected to be small.

In addition, as the number of hops of a shortest path is "small" (remember that complex network are small-world networks). The number of calls to the shortest path in-branching update is expected to be small for PSB. As this procedure is faster in PSB than SB* and the number of calls is similar, PSB algorithm is faster than SB* on complex networks. This is not valid on road network because the number of hops of a shortest path may be big and the shortest path in-branching update is called many more times.

To conclude, on complex networks, the PSB algorithm is the fastest among the considered algorithms, it has a feasible working memory and this seems to be related to structural properties of complex networks.

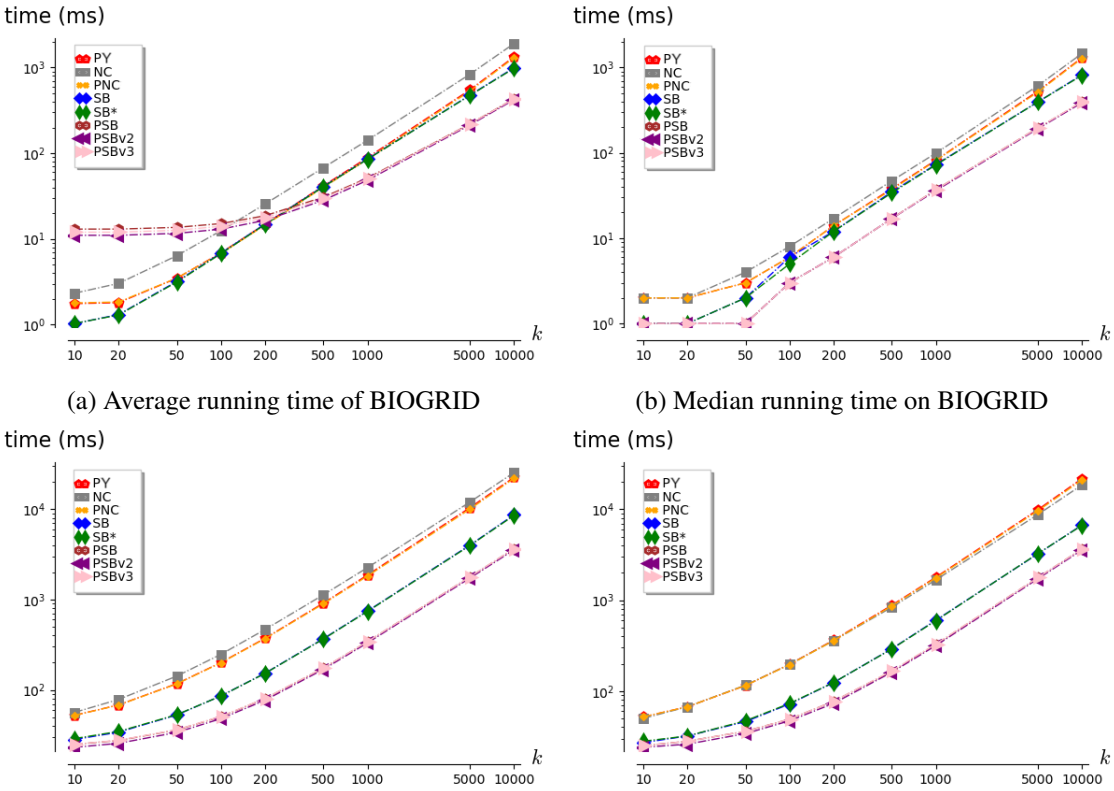
2.6.3 Impact of the properties of the queries

Here we study the impact of other parameters (related to the properties of the queries) on the running time of the algorithms. In particular, the number of hops of a shortest path and the stretch of a shortest path from the "center" (defined below).

As noticed in Section 2.6.2, some algorithms have different behaviors with respect to the structure of the network and the query's properties. In this section, we are investigating whether some properties of an s-t query may explain the variations of the running time of the algorithms. For this purpose, we have considered two criteria of each s-t query, the number of hops between s and t of a shortest s-t path and the stretch of a shortest s-t path from the "center" of the graph.

A similar indicator to the number of hops of a shortest path is the maximum number of hops, that is the number of hops of a path with maximum number of hops among the k shortest paths given by an algorithm. Formally, the maximum number of hops of a kSSP query from s to t using an algorithm A is M if and only if for each path P given by A, $|P| \leq M$. We studied this parameter and the obtained results are almost the same as those obtained while studying the number of hops. Therefore, we only describe the results corresponding to the number of hops.

The number of hops of the shortest path (the one given by an SP algorithm) is a meaningful criterion to be studied, as each of the algorithms has the common routine of iterating and eventually calling an SP algorithm for each vertex on the first shortest path. As the obtained results are similar



(a) Average running time of BIOGRID

(b) Median running time on BIOGRID

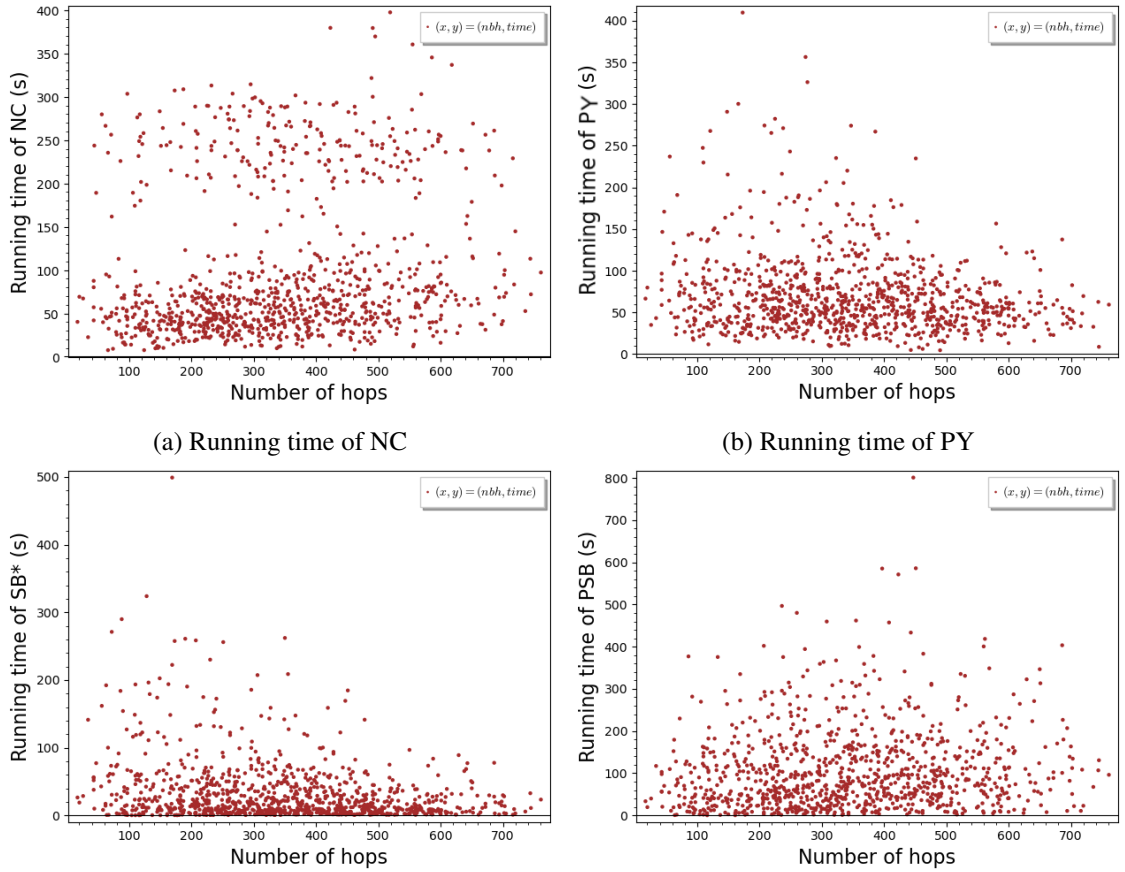
(c) Average running time of CAIDA

(d) Median running time on CAIDA

Figure 2.4 – The running time of the kSSP algorithms on complex network with respect to the values of k

on different road networks, we only add and discuss the results of NC, PY, SB* and PSB algorithm on NY road network.

As shown in Figure 2.5, no clear pattern can lead to a establish a concrete relationship between the running time for one of these algorithms and the number of hops. For instance, the SB* algorithms (Figure 2.5c) has big running time for queries with relatively small number of hops. However, this is not flagrant and it does not hold on the remaining algorithms (Figures 2.5a, 2.5b and 2.5d).



(a) Running time of NC

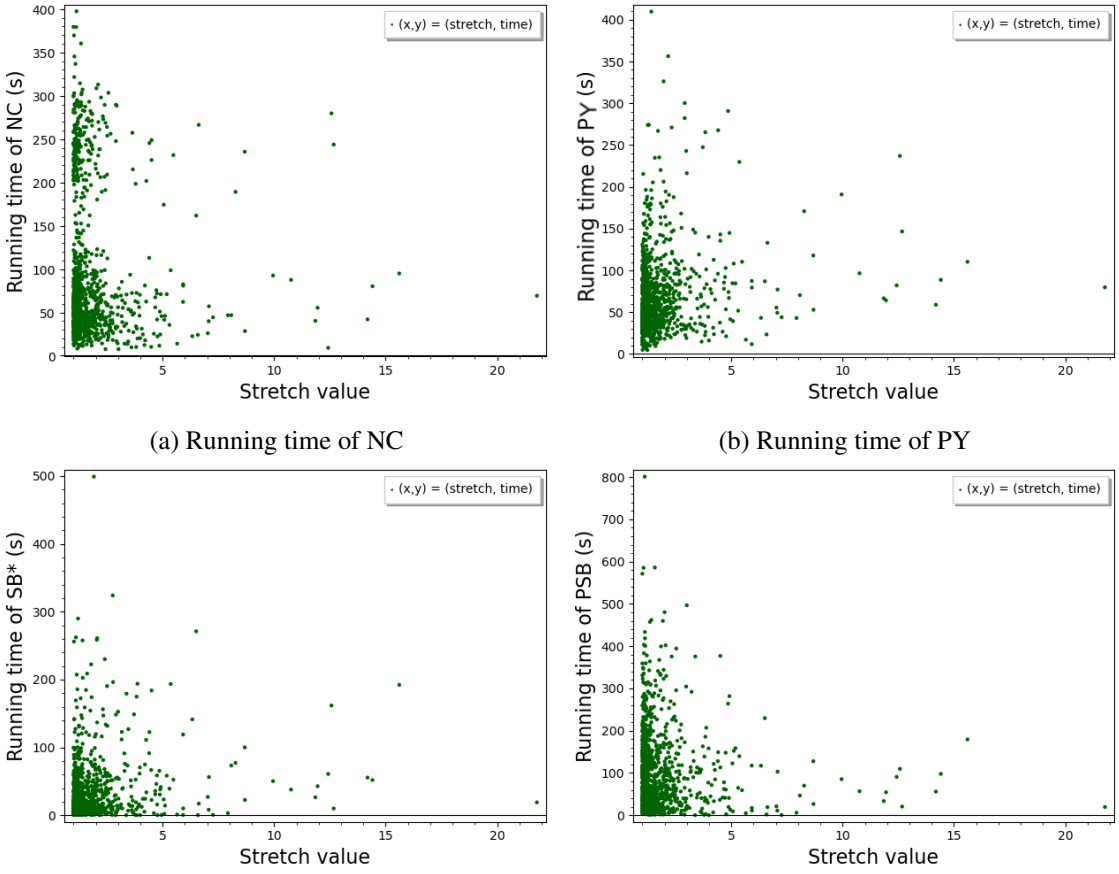
(b) Running time of PY

(c) Running time of SB*

(d) Running time of PSB

Figure 2.5 – The running time with respect to the number of hops of the shortest path of some kSSP algorithms on NY. Each dot corresponds to one pair source/destination ($k = 1,000$).

Another interesting parameter to be studied in road networks is the stretch from the center (by center we mean a vertex of minimum eccentricity in the network). So, the stretch of an s-t path P from the center c is defined as the ratio between the length of a shortest s-t path passing through c and the length of P. This gives an indicator on how far a path can be from the center. In order to establish a relation between the running time and the stretch of the center, we plotted (figure 2.6) the running time with respect to the stretch value. Clearly, no flagrant pattern related to the stretch value is found. Then, no concrete relation between the running time of these algorithms and the stretch from the center can be established based on our experiments.



(a) Running time of NC

(b) Running time of PY

(c) Running time of SB*

(d) Running time of PSB

Figure 2.6 – The running time with respect to the stretch from the center of some kSSP algorithms on NY. Each dot corresponds to one pair source/destination ($k = 1,000$).

To conclude, no clear relationship between the running time of an algorithm and the number of hops, neither the stretch from the center is experimentally found. However, it seems that not all queries perform the same. For instance, in Figures 2.1b, 2.2b, 2.5a and 2.6a, it can be observed two distinct clouds of points. It would be interesting to understand whether these two clouds correspond to some specific properties of the queries. This could help us to design improvements of our algorithms.

Conclusion. Our simulation results show that the best algorithm to be chosen for solving the kSSP problem depends on the use case. For instance, on the considered complex networks, the PSB algorithm achieves the best results. Indeed, it is the fastest among the considered algorithms, and, similarly to the other algorithms, it has low memory consumption. Besides, on road networks, if large memory consumption is allowed, the SB* algorithm is the fastest among the considered algorithms on most of the queries. However, the PY algorithm has a running time that is more stable, and it has low working memory. Therefore, the PY algorithm seems to offer a better space-time trade-off than the other considered algorithms on road networks.

An empirical framework for the selection of the most appropriate kSSP algorithm with respect to the use case is suggested in Figure 2.7.

An open problem is how to handle the kSSP problem on networks with arbitrarily arc weights (including negative weights). Another interesting question is how to design a data structure enabling to quickly answer kSSP queries similarly to the data structures used by the hub labelling and contraction hierarchy schemes to answer distance queries [21]. A probably more difficult question would be to address dynamic networks, i.e., where the weights of the arcs evolve along time (e.g., in road networks where the traversal time of an arc may vary). Would it be possible to quickly update the solutions after a modification in the network ?

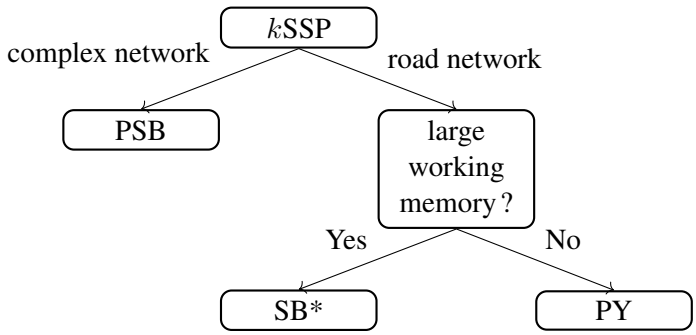


Figure 2.7 – A framework of the appropriate kSSP algorithm with respect to each usecase

2.7 On arbitrarily weighted digraphs with no negative cycles

In this section we study the kSSP problem in arbitrarily weighted digraph with no negative cycles. We first present a straightforward adaptation of Yen's algorithm, called Y-BFM, with worst-case running time in $O(kn^2m)$, and then show how to reduce this time complexity to $O(kn(m + n\log n))$ using an SP in-branching update algorithm. Then, we propose two algorithms, called PY-BFM and PNC-BFM. Both with running time in $O(kn^2m)$ but offering better performances than Y-BFM in practice.

This section is organized as follows. First, a baseline solution is described. Then, we show how to reduce its time complexity to $O(kn(m+n\log n))$ in Section 2.7.1 while the adaptation of other kSSP algorithms are described in Section 2.7.2. Finally, an experimental evaluation is presented in Section 2.7.3.

A baseline solution Recall that Bellman-Ford-Moore's (BFM) algorithm [93] can compute an SP in-branching in a digraph with arbitrary arc weights without negative cycles in time $O(nm)$. So, based on Corollary 2.2.2, one can solve the kSSP problem in digraphs with arbitrary arc weights without negative cycles within Yen's framework by replacing Dijkstra's algorithm by Bellman-Ford-Moore's (BFM) algorithm. This gives a polynomial time algorithm, called Y-BFM, with worst-case time complexity in $O(kn(nm)) = O(kn^2m)$.

2.7.1 Yen-Ball-String (Y-BS) algorithms

Now, let us show that the kSSP problem on arbitrarily weighted digraphs with no negative cycles can also be solved in time $O(kn(m + n\log n))$. For this purpose, we use two building blocks. First, we use the shortest path tree update algorithm described in [90] to show that a second shortest path can be computed in time $O(n(m + n\log n))$. Then, we generalize this result for each $k > 2$ based on the fact that k shortest simple paths in a weighted digraph D can be computed in $O(k)$ calls to a second shortest path algorithm in subdigraphs of D [101]. We call this method Yen-Ball-String algorithm (Y-BS).

Finding a second shortest simple path. Let D be an arbitrarily weighted digraphs without negative cycles, let T be an SP in-branching of D rooted at t, and suppose that an arc e in A is removed from the digraph. If $e \in T$, then T is not a valid SP in-branching anymore. In order to get a valid SP in-branching in $D \setminus \{e\}$, a naive solution is to call the BFM algorithm from scratch. However, a better solution is to use an algorithm for updating an SP in-branching after the addition or deletion of an arc, such as the Ball String algorithm proposed in [95, 90].

Precisely, it has been proved in [90] that the Ball-String algorithm proposed in [95] for updating an SP in-branching after the addition or deletion of any arc is also correct for digraphs with arbitrary arc weights but no negative cycles and that its time complexity remains in $O(m+n\log n)$. Furthermore, we observe that the results of [90] are also valid for vertex deletion. In fact, it suffices to consider the modified digraph in which each vertex u is split into two vertices u^{in} and u^{out} connected by a zero weighted arc $u^{in}u^{out}$, and each arc $uv \in A$ is replaced by an arc $u^{out}v^{in}$ with the same length (i.e., $\ell(u^{out}v^{in}) = \ell(uv)$). Then, the deletion of vertex u is equivalent to the deletion of the arc $u^{in}u^{out}$ in the modified digraph.

It follows that, given an SP in-branching T rooted at t and containing a shortest path P from s to t, one can compute all the shortest simple detours of P following Yen's framework using the Ball-String update algorithm. Note that, at each iteration, at most one vertex and a single arc are removed. So, the Ball-String algorithm is called at most $O(n)$ times. Hence, finding a shortest simple detour at each vertex v in P can be done in $O(n(m + n\log n))$.

Finally, the overall time complexity of finding a second shortest simple path is the sum of the time needed to computes an SP in-branching i.e, $O(nm)$ using BFM, and the time complexity of computing a shortest simple detour at each vertex along the given shortest path $O(n(m+n\log n))$. Therefore, a second shortest path can be found in time $O(n(m + n\log n))$.

Finding $k \geq 2$ shortest simple paths. To generalize this algorithm for each $k \geq 2$, we use the following result from [101].

Theorem 2.7.1 (L. Roddity, 2012 [101])**.** *Let $D = (V, A)$ be a weighted digraph, the problem of computing k shortest simple paths between s and t can be solved by $O(k)$ computations of a second shortest simple path each time in a different sub-digraph of D.*

Thus, our algorithm computes a second shortest path by dynamically updating an SP in-branching as described above. Then, it performs $O(k)$ calls to a second shortest path algorithm in order to extract k shortest simple paths using the reduction described in [101]. We conclude the following.

Theorem 2.7.2. *The kSSP problem can be solved in arbitrarily weighted digraphs without negative cycles in time $O(kn(m + n \log n))$.*

Clearly, the cubic time complexity lower bound of the kSSP on digraphs with non-negative weights provided by Williams and Williams [118] (described in Section 2.1 page 39) is also valid in the case of arbitrarily weighted digraphs without negative cycles. This means that improving this bound, i.e. $O(kn(m + n \log n))$, seems unlikely at the moment.

2.7.2 Adaptation of some kSSP algorithms

As described above, several algorithm were proposed to improve the efficiency of the resolution of the kSSP problem in practice [67, 79, 111, 70, 59, 54, 83, 82, 10, 12]. However, none of them deals with arbitrarily weighted digraphs. Here we consider solving, efficiently in practice, the kSSP on arbitrarily weighted digraphs without negative cycles.

Similarly to what we described in Section 2.5, the main drawback of the Y-BFM algorithm is its large number of calls to an SP algorithm (the number of BFM calls) which is expensive in practice. Here, we adapt the Postponed Yen algorithm (PY) proposed in Section 2.5 and the Postponed Node Classification algorithms (PNC) proposed in Section 2.6 to the case of arbitrarily weighted digraphs without negative cycles in order to end up with new algorithms called the Postponed Yen Bellman-Ford-Moore (PY-BFM) and Postponed Node Classification BFM (PNC-BFM) algorithm. These new algorithms improve the running time of Y-BFM either by avoiding some SP calls, or by reducing the running time consumed by a BFM call using classical SP in-branching updates.

Postponed Yen - Bellman-Ford-Moore (PY-BFM) algorithm The Postponed Yen - Bellman-Ford-Moore algorithm is a straightforward adaptation of the PY algorithm described in Section 2.5. Precisely, it replaces Dijkstra's SP algorithm by the Bellmand-Ford-Moore algorithm. The same pseudocode of PY algorithm (Algorithm 2.2) describe the PY-BFM algorithm, by using BFM algorithm instead of Dijkstra's algorithm to find a shortest path, (lines 4 and 21). In a worst case scenario, a BFM call is needed to compute a shortest simple detour at each vertex of the iterated path, i.e, PY-BFM has the same time complexity as Y-BFM, that is, in $O(kn^2m)$. Besides, our experiments presented in Section 2.7.3 shows that PY-BFM is up to 50 times faster than Y-BFM on road networks, and up to 10 times faster on complex networks.

Postponed Node-Classification - Bellman-Ford-Moore (PNC-BFM) algorithm The Postponed Node Classification - Bellmand-Ford-Moore algorithm (PNC-BFM) is a special variant of the PY-BFM algorithm leading to an important speed up on practice.

In fact, the PY-BFM algorithm computes a shortest simple detour (Algorithm 2.2) using a BFM call (from scratch). Instead, the PNC-BFM computes a shortest path by updating the first SP in-branching computed (T) using standard methods for updating shortest path tree [57]. This reduces the computation to a smaller subdigraph (the invalid part) and leads to a significant speed up.

The PNC-BFM algorithm makes exactly the same number of BFM calls as the PY-BFM algorithm. So, it has the same complexity bound i.e, $O(kn^2m)$. Besides, the computational results presented in Section 2.7.3 show that this update procedure gives an average speed up, in practice, by a factor 1.5 on road network and 1.1 on complex networks. So, the PNC-BFM is between one and two orders of mangitude faster than Y-BFM in practice.

2.7.3 Experimental evaluation

In this section, we describe our experimental evaluation. First, we start by describing our implementation and settings, then we discuss our experimental results on road and complex networks.

Experimental settings Here we specify the details of the implementation and the setting used in our experiments.

We have implemented three algorithms presented in this section (Y-BFM, PY-BFM and PNC-BFM) in C++ and our code is publicly available [14]. We did not implement the improved algorithm described above (Y-BS), as it seems to be noncompetitive despite its appealing complexity bound in $O(kn(m + n \log n))$. First, since the SP in-branching updates procedure is not as performant as a well engineered implementation of BFM, and it is called twice the number of BFM calls in Y-BFM algorithm. Second, because it hides considerable constants in the generalisation procedure (Theorem 2.7.1) of finding k shortest simple path using the described second shortest simple path algorithm.

Our implementation of BFM algorithm follows the shortest path faster algorithm routine (known as SPFA) [93]. Precisely, the vertices of the digraph are partitionned into two sets, a set of relaxed vertices A initialized with $\{s\}$, and a set of the remaining vertices (vertices to be relaxed) $B = V \setminus A$. Then, at each iteration, only the arcs with a tail in A and a head in B are relaxed, finally the set A and B are swapped. This procedure is repeated $n - 1$ times.

Moreover, we have implemented a special copy operation that enables to update the in-branching when a set of nodes is removed from the graph. This corresponds to the operations performed when computing a shortest path by updating the first SP in-branching T computed in PNC-BFM.

Networks setting We have evaluated the performance of our algorithms on some road networks from the 9th DIMACS implementation challenge [44] and on several complex networks.

A road network of a city is the digraph modeling its roads, the characteristics of the chosen networks are reported in Table 2.5.

On the other side, complex networks model different types of networks.

For instance, BTC-Otc is the who-trusts-whom network of people who trade using Bitcoin on a platform called Bitcoin OTC. Since Bitcoin users are anonymous, there is a need to maintain

Network	n	m	Description
Quanzou	1 426	4 619	Road network of Quanzou [78]
ROME	3 353	8 870	Road network of Rome [44]
DC	9 559	29 682	Road network of Washington DC [44]
BitCoin-Otc	4 709	33 461	Bitcoin Otc web of trust network [81]
Slashdot-Zoo	27 222	342 747	Slashdot Zoo signed social network, 16/02/2009 [87]
Soc-sign-epinions	41 441	693 507	Who-trust-whom online social network [87]

Table 2.5 – Characteristics of the graphs used in kSSP experiments : number of nodes (n), number of edges (m) and descriptions.

a record of users' reputation to prevent transactions with fraudulent and risky users. Members of Bitcoin OTC rate other members in a scale of -10 to +10 (total distrust to total trust) in steps of 1 [81]. Similarly, SOC-Epinions is who-trust-whom online social network of a general consumer review site called Epinions [87]. Finally, the Slash-Zoo network contains friend/foe (+1/-1) links between the users of Slashdot (a technology-related news website) [87].

In order to have negative arcs on road networks, the following preprocessing is applied before conducting any experiment. For each of the road networks, a random vertex u is picked, and a shortest path out-branching T rooted at u is computed, then, the length of each arc of T is associated to -1. Fortunately, no negative cycles arise in the three considered road networks after applying this procedure. However, this was not enough for complex networks where a more involved procedure is applied to ensure the absence of negative cycles. Precisely, for each of the considered complex network, a vertex u with maximum degree is picked, a shortest path out-branching T rooted at u is computed. Let d_{max} be the distance from u to the farthest vertex from u in the network, i.e, $d_{max} = Max_{v \in V} d(u, v)$. Moreover, let l_{min} be the length of an arc with minimum length in the network, i.e, for each $a \in A$, $\ell(a) \geq l_{min}$. Finally, the arcs of T are associated to $-1 * l_{min}/(d_{max})$. Clearly, applying this procedure to a positive weighted digraph cannot create any negative cycle as any cycle of the resulting digraph has length bigger than $l_{min} + d_{max} * (-1) * l_{min}/(d_{max})$, which is non-negative. Note that, after applying the described procedures, the number of arcs with negative weights of both road and complex networks is almost n.

In our experiments, we have randomly chosen 1000 queries (source-destination pairs of vertices) for each network, and we have run each algorithm for each of these pairs for $k = 100$.

We have measured the execution time and the number of BFM calls. Note that the number of BFM calls gives an indication of the running time that is independent of the implementation and the architecture of the machine [77].

All reported computations have been performed on computers equipped with 48 12-core 2.20 GHz AMD Opteron(tm) Processor 6174 and 252 GB of RAM.

Experimental results We have measured the average and the median of the algorithms' running time in the considered networks. The data (the running time and the number of BFM calls) in Tables 2.6 and 2.7, and in Figures 2.9 and 2.10, corresponds to the biggest experienced value of k ($k = 100$). While the data in Figure 2.8 corresponds to their evolution with respect to the values of k.

The average and median running times reported in Table 2.6 show that the PY-BFM algorithm is significantly faster than the Y-PFM algorithm for every considered network (a speed up between

		Quanzou	Rome	DC	BTC-Otc	Slash	Soc
Y-BFM	avg	1275	14473	35948	1419	26157	55761
	med	1114	6853	31821	1391	26082	55923
PY-BFM	avg	155	1305	622	142	2554	4964
	med	135	589	319	131	2537	**4021**
PNC-BFM	avg	**109**	**785**	**346**	**126**	**2274**	**4767**
	med	**81**	**361**	**183**	**113**	**2416**	4089

Table 2.6 – Average running time (ms) of the algorithms on road networks, ($k = 100$)

		Quanzou	Rome	DC	BTC-Otc	Slash	Soc
Y-BFM	avg	986	1507	2657	464	718	615
	med	849	1299	2288	456	717	612
PY-BFM	avg	151	193	50	69	110	73
	med	122	178	24	64	113	62

Table 2.7 – number of BFM calls of Y-BFM and PY-BFM on road and complex networks ($k = 100$)

10 and 50 is achieved). Moreover, a refined comparison on Rome and Slash networks (Figures 2.9 and 2.10) show that PY-BFM is faster than Y-BFM for almost all queries. In addition, Figure 2.8 shows that this speed up remains significant even for small values of k (even for $k = 2$) for DC and BTC networks.

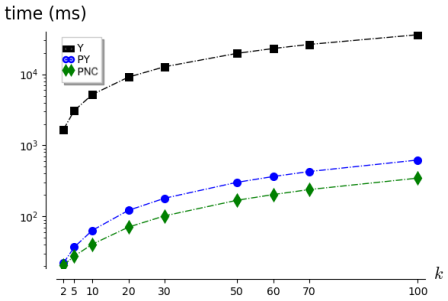
Based on these remarks, we conclude that, in practice, PY-BFM is faster than Y-BFM for almost every scenario (the value of k, the query specifications and the network structure).

Furthermore, Table 2.7 and Figures 2.8d, 2.8b, and 2.9d show that, on all the considered networks, the number of BFM calls is significantly reduced using PY-BFM. This ensures that a similar speed up is guaranteed for any experimental settings [77].

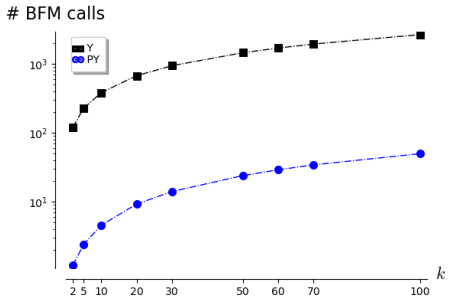
As the obtained results are similar, we only displayed data obtained from experiments on selected networks (Rome and Slash for Figure 2.9 and 2.10, and DC and BTC for Figure 2.8). However, the results/plots corresponding to the remaining networks are very similar.

Now, let us compare PY-BFM and PNC-BFM. Table 2.6 and Figure 2.8 show that PNC-BFM is almost always faster, on average, than PY-BFM (with a speed up up to 1.8 achieved on DC). Moreover, a more refined plot in Figures 2.9c and 2.10c show that PNC-BFM is faster than PY-BFM on the majority of the queries. In other words, PNC-BFM is faster, in practice, than PY-BFM. This is surprising, as the original PNC algorithm (with Dijkstra's algorithm) was outperformed by PY (Section 2.6). This is justified by the fact that a BFM call is more time-consuming than a Dijkstra's call, and the cost (in terms of running time) of a coloring / re-initializing an SP tree is compensated by the speed-up of a BFM call as it is long, while it is not the case with Dijkstra's algorithm as it is less time-consuming.

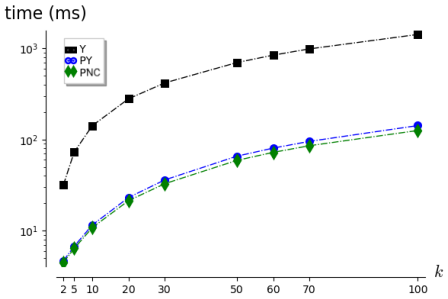
Conclusion To conclude, the PNC-BFM algorithm is the fastest kSSP algorithm in practice, working on arbitrarily weighted digraphs with no negative cycles. Several experiments on road and complex networks show that the PNC-BFM algorithm is between one and two orders of magnitude faster than the straightforward adaptation of the classical solution, i.e, Y-BFM algorithm.



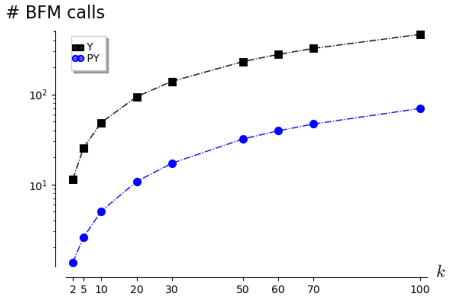
(a) Average running time on DC



(b) Average number of BFM calls on DC



(c) Average running time on BTC



(d) Average number of BFM calls on BTC

Figure 2.8 – The running time of the $kSSP$ algorithms on DC and BTC network with respect to the values of k

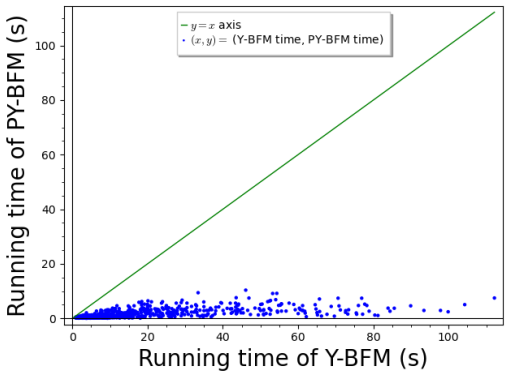
2.8 Arbitrarily weighted digraphs

Finding a shortest path in arbitrarily weighted digraphs, also known as the shortest elementary path (ESP) is known to be NP-Hard, as it can be reduced from the Hamiltonian path problem. So, the kSP and the kSSP on arbitrarily weighted digraphs are NP-Hard too, as both require solving the ESP for $k = 1$. However, as we are interested in practical solutions, we present in this section several Mixed Integer linear Programming algorithms allowing to solve the kSSP on arbitrarily weighted digraphs. Most of these models are inspired from models proposed to solve the ESP and the Traveling Salesman problem (TSP).

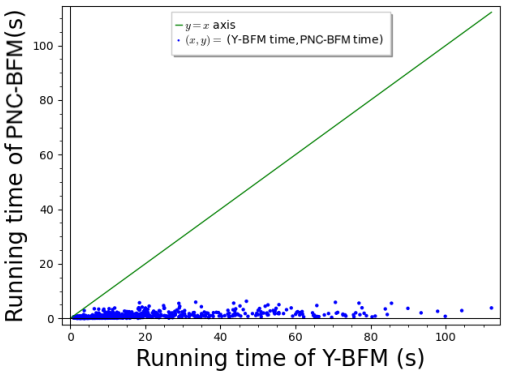
Mixed Integer Linear Programming (MIP) formulations In this section, we present several MIP formulations for solving the kSSP problem in arbitrarily weighted digraphs (with or without negative cycles). More precisely, we present in Section 2.8.2 a model enabling to compute the k paths at once. Then, in Section 2.8.3 we present a model to compute a shortest path that is different from a set of given paths. This model can then be used to compute the paths one after the other. We start presenting in Section 2.8.1 a MIP formulation for computing a shortest s-t simple path in directed graphs with arbitrary arc weights.

2.8.1 Finding a shortest simple path

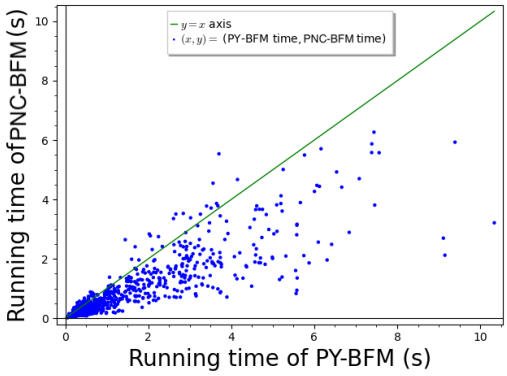
In this section, we present two MIP formulations for solving the shortest simple path problem, also known as the shortest elementary path (ESP) problem, in directed graphs with arbitrary arc



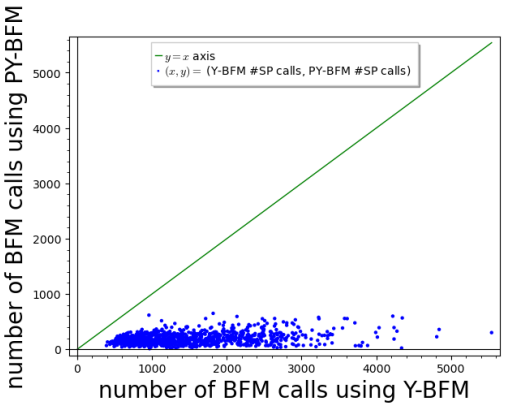
(a) Running time of Y-BFM and PY-BFM



(b) Running time of Y-BFM and PNC-BFM

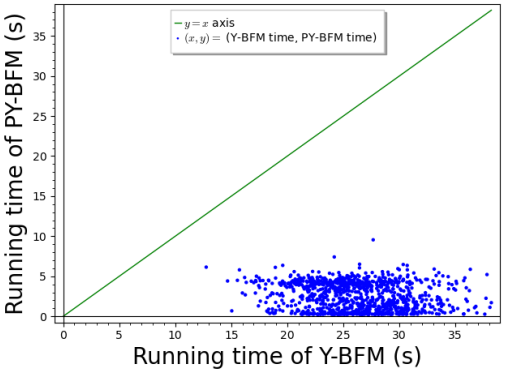


(c) Running time of PY-BFM and PNC-BFM

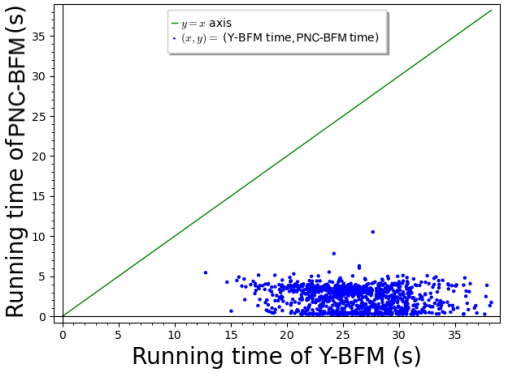


(d) Number of BFM calls of Y-BFM and PY-BFM

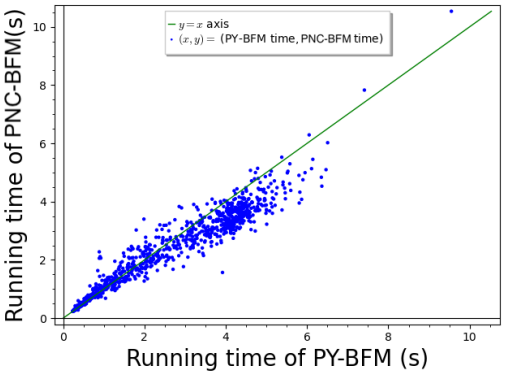
Figure 2.9 – Comparison of the running time and the number of BFM calls on Rome network. Each dot corresponds to one pair source/destination ($k = 100$).



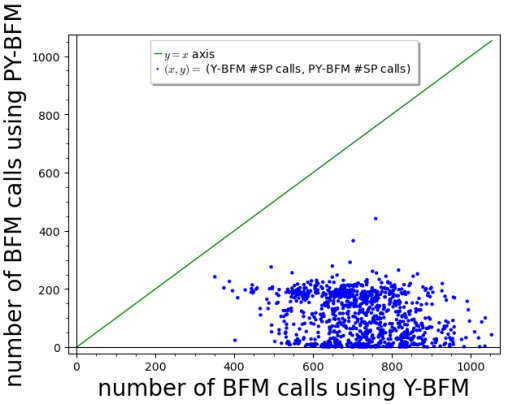
(a) Running time of Y-BFM and PY-BFM



(b) Running time of Y-BFM and PNC-BFM



(c) Running time of PY-BFM and PNC-BFM



(d) number of BFM calls of Y-BFM and PY-BFM

Figure 2.10 – Comparison of the running time and the number of BFM calls on Slash network. Each dot corresponds to one pair source/destination ($k = 100$).

lengths and possibly negative length cycles. We choose to use first a formulation based on the so-called Miller-Tucker-Zemlin (MTZ) subtour elimination constraints [92] that have been proposed for solving the asymetric traveling salesman problem (ATSP) and that are valid for the ESP problem. Other subtour elimination constraints have been proposed for ATSP and ESP, and are based on cutset inequalities [40, 115, 49, 55], reformulations of the MTZ constraints [42], partial orderings [73, 113], layered graphs [115, 41], and single [61] or multicommodity auxiliary flows [123, 38]. Although formulations based on the MTZ constraints for the ESP problem are known to have a weak linear programming relaxation compared to other formulations [84, 65], experiments shows that this approach is effective on small instances [42]. Hence, many improvements have been proposed [45, 23, 103, 115, 42, 27] for solving the ATSP and ESP problems.

Formulation 1. *Minimize Objective* (2.1) *subject to Constraints* (2.2)-(2.8).

We use the following variables :
— x_{uv} is a binary variable set to 1 if arc $uv \in A$ is selected and 0 otherwise.
— $p_u \in \mathbb{R}$ is a variable encoding the relative position of vertex $u \in V$.

Objective :

$$\text{minimize} \qquad \sum_{uv \in A} \ell(uv) x_{uv} \tag{2.1}$$

Flow conservation constraints :

$$\sum_{v \in N^-(u)} x_{vu} - \sum_{v \in N^+(u)} x_{uv} = \begin{cases} -1 & \text{if } u = s \\ 1 & \text{if } u = t \\ 0 & \text{else} \end{cases} \qquad \forall u \in V \tag{2.2}$$

$$\sum_{v \in N^-(u)} x_{vu} \leq 1 \qquad \forall u \in V \tag{2.3}$$

Subtour elimination constraints :

$$p_u + 1 + n(x_{uv} - 1) \leq p_v \qquad \forall uv \in A \tag{2.4}$$

$$p_u - p_v + (n-1)x_{uv} + (n-3)x_{vu} \leq n - 2 \qquad \forall uv \in A \text{ such that } vu \in A \tag{2.5}$$

$$p_s + \text{dist}_{\text{BFS}}(s, u) \leq p_u \qquad \forall u \in V \tag{2.6}$$

Domains of the variables :

$$x_{uv} \in \{0, 1\} \qquad \forall uv \in A \tag{2.7}$$

$$p_u \in \mathbb{R} \qquad \forall u \in V \tag{2.8}$$

Constraints (2.2) are the flow conservation constraints ensuring the routing of one unit of flow from s to t. Constraints (2.3) ensure that a vertex can be the head of at most one selected arc. Constraints (2.4) are the Miller-Tucker-Zemlin (MTZ) subtour elimination constraints [92]. Roughly, suppose that arcs ab, bc and ca are selected. Then, by Constraints (2.4), we have $p_a + 1 \leq$

p_b, $p_b+1 \leq p_c$ and $p_c+1 \leq p_a$, that is $p_a+3 \leq p_a$, a contradiction. Hence, the set of selected arcs is acyclic. Constraints (2.5) are the so-called lifted MTZ constraints proposed in [45] to strengthen Constraints (2.4). Constraints (2.6) further strengthen Constraints (2.4) by imposing a minimum gap between the relative position of s and of any vertex $u \in V$ based on the minimum number $\text{dist}_{\text{BFS}}(s,u)$ of arcs along any path from s to u (i.e., the unweighted distance from s to u in D). Constraints (2.7) and (2.8) define the domains of the variables. Overall, this MIP has m binary variables, n fractional variables, and $O(n+m)$ constraints.

Another formulation for the ESP problem, that is also considered efficient in practice, is based on the generalized cutset inequalities (GCS) that have been proposed in [55] to eliminate subtours. The GCS constraints ensure that if a vertex v is the tail of an arc of the solution (i.e., if there is $a \in \delta^+(v)$ such that $x_a = 1$, where $\delta^+(v)$ denote the set of arcs going out from v), then any cut induced by a subset $S \subset V$ containing v is traversed by at least one outgoing arc. This is formalized with Constraints (2.9) that ensure that the number of selected outgoing arcs of each subset S is at least the number of selected outgoing arcs of each vertex in S. In Constraints (2.9), $\delta^+(S)$ denotes the set of arcs going out of the set S of vertices. Constraints (2.9) have been shown to be a strengthen formulation of the Dantzig-Fulkerson-Johnson (DFJ) cutset inequalities [40] for the ESP problem [115].

Formulation 2. *Minimize Objective* (2.1) *subject to Constraints* (2.2)-(2.3)*,* (2.7)*, and* (2.9)*.*

$$\sum_{a \in \delta^+(S)} x_a \geq \sum_{a \in \delta^+(v)} x_a \qquad \forall S \subset V \setminus \{s,t\},\ |S| \geq 2,\ \forall v \in S \tag{2.9}$$

Formulation 2 has m binary variables and $O(n2^n)$ constraints. However, in practice, these constraints can be added incrementally, within a branch-and-cut framework, and only a few of them will be added. More precisely, we start from an initial solution of formulation 2 without Constraints (2.9). Then, for each subset $S \subset V \setminus \{s,t\}$ for which the inequality is violated, we add the corresponding constraint and solve this improved MIP. The separation of violated inequalities can be done using the minimum cuts or the strongly connected components of the digraph build from D by keeping only the arcs for which $x_a = 1$ (see for instance [49, 115] for more details). This process ends when it is no longer possible to add constraints, that is, when no subset of vertices violates the inequality. Formulation 2 is then optimally solved.

2.8.2 Compact MIP formulation for kSSP

In this section, we first present a compact MIP formulation for the kSSP problem. It is based on a multi-commodity flow problem, each commodity corresponding to the routing of one unit of flow from s to t along a unique path, with additional constraints ensuring the use of different paths for each unit of flow.

Let $I = \{1, 2, \ldots, k\}$ and consider the following variables :

— x^i_{uv} is a binary variable set to 1 if arc $uv \in A$ is selected for path P_i and 0 otherwise.

— $y^{i,j}_{uv}$ is a binary variable set to 1 if arc $uv \in A$ is used for both paths P_i and P_j, and 0 otherwise.

— $p^i_u \in \mathbb{R}$ is a variable used in the subtour elimination constraints related to the calculation of path P_i. It encodes the relative position of vertex $u \in V$.

The MIP formulation is as follows.

Formulation 3. *Minimize Objective* (2.10) *subject to Constraints* (2.11)-(2.23)*.*

Objective :

$$\text{minimize} \qquad \sum_{uv \in A} \ell(uv) \sum_{i \in I} x^i_{uv} \tag{2.10}$$

Flow conservation constraints :

$$\sum_{v \in N^-(u)} x^i_{vu} - \sum_{v \in N^+(u)} x^i_{uv} = \begin{cases} -1 & \text{if } u = s \\ 1 & \text{if } u = t \\ 0 & \text{else} \end{cases} \qquad \forall i \in I,\ \forall u \in V \tag{2.11}$$

$$\sum_{v \in N^-(u)} x^i_{vu} \leq 1 \qquad \forall i \in I,\ \forall u \in V \tag{2.12}$$

Subtour elimination constraints :

$$p^i_u + 1 + n(x^i_{uv} - 1) \leq p^i_v \qquad \forall i \in I,\ \forall uv \in A \tag{2.13}$$

$$p^i_u - p^i_v + (n-1)x^i_{uv} + (n-3)x^i_{vu} \leq n-2 \qquad \forall i \in I,\ \forall uv \in A \text{ such that } vu \in A \tag{2.14}$$

$$p^i_s + \text{dist}_{\text{BFS}}(s,u) \leq p^i_u \qquad \forall i \in I,\ \forall u \in V \tag{2.15}$$

Constraints ensuring that selected paths are different :

$$y^{i,j}_{uv} \leq x^i_{uv} \qquad \forall i,j \in I,\ \forall uv \in A \tag{2.16}$$

$$y^{i,j}_{uv} \leq x^j_{uv} \qquad \forall i,j \in I,\ \forall uv \in A \tag{2.17}$$

$$x^i_{uv} + x^i_{uv} \leq 1 + y^{i,j}_{uv} \qquad \forall i,j \in I,\ \forall uv \in A \tag{2.18}$$

$$1 + \sum_{uv \in A} y^{i,j}_{uv} \leq \sum_{uv \in A} x^i_{uv} \qquad \forall i,j \in I \tag{2.19}$$

$$1 + \sum_{uv \in A} y^{i,j}_{uv} \leq \sum_{uv \in A} x^j_{uv} \qquad \forall i,j \in I \tag{2.20}$$

Domains of the variables :

$$x^i_{uv} \in \{0,1\} \qquad \forall i \in I,\ \forall uv \in A \tag{2.21}$$

$$y^{i,j}_{uv} \in \{0,1\} \qquad \forall i,j \in I,\ \forall uv \in A \tag{2.22}$$

$$p^i_u \in \mathbb{R} \qquad \forall i \in I,\ \forall u \in V \tag{2.23}$$

Constraints (2.11)-(2.15) ensure the computation of k s-t simple paths. Constraints (2.16)-(2.18) correspond to the linearization of the expression $y^{i,j}_{uv} = x^i_{uv} x^j_{uv}$ and ensure that variable $y^{i,j}_{uv}$ is set to 1 if and only if $x^i_{uv} = 1$ and $x^j_{uv} = 1$. Constraints (2.19) and (2.20) ensure that paths P_i and P_j differ by at least one arc. Observe that variables $y^{i,j}_{uv}$ and Constraints (2.16)-(2.20) can be restricted to $i < j$. Finally, Constraints (2.21)-(2.23) define the domains of the variables.

Overall, this MIP formulation uses $O(k^2 m)$ binary variables, $O(kn)$ fractional variables, and $O(k^2 m)$ constraints.

As for the ESP problem, one can also use the GCS constraints to eliminate subtours. This yields the following MIP formulation.

Formulation 4. *Minimize Objective* (2.10) *subject to Constraints* (2.11), (2.12), (2.16)-(2.22) *and* (2.24).

$$\sum_{a \in \delta^+(S)} x_a^i \geq \sum_{a \in \delta^+(v)} x_a^i \qquad \forall i \in I,\ \forall S \subset V \setminus \{s,t\},\ |S| \geq 2,\ \forall v \in S \tag{2.24}$$

Observe that Formulations 3 and 4 do not ensure that $\ell(P_i) \leq \ell(P_{i+1})$ for $1 \leq i < k$, but that the k selected paths are distinct and are the k shortest. In order to get the paths in the right order, one can either simply relabel the paths with the right ordering, or add constraints enforcing the length of path P_i to be at most the length of path P_{i+1}, that is

$$\sum_{uv \in A} \ell(uv) x_{uv}^i \leq \sum_{uv \in A} \ell(uv) x_{uv}^{i+1} \qquad \forall\, 1 \leq i < k \tag{2.25}$$

2.8.3 MIP formulation for kSSP with constraints generation

A drawback of Formulation 3 is the large number of binary variables and constraints. Also, we now present an incremental method for solving the kSSP problem using constraints generation. Roughly, the idea is to first solve Formulation 1 to get a shortest s-t simple path P_1, then add a constraint preventing to find this path again and solve this extended MIP to get the shortest s-t simple path P_2 that is different from P_1. We repeat this process until k paths are found (or no new path can be found).

More precisely, let $\mathcal{P}$ be the set of paths that have already been computed (initially, $\mathcal{P} = \emptyset$). We use the following MIP formulation to find the next path P, and then add it to $\mathcal{P}$.

Formulation 5. *Minimize Objective* (2.1) *subject to Constraints* (2.2)-(2.8) *and* (2.26).

$$1 + \sum_{uv \in A(P)} x_{uv} \leq |A(P)| \qquad \forall P \in \mathcal{P} \tag{2.26}$$

In Formulation 5, Constraints (2.26) prevent from selecting the same sets of arcs than a previously found path. Overall, this MIP formulation uses m binary variables, n fractional variables, and $O(k+n+m)$ constraints. To get k paths, one has to solve this formulation k times, each time with one constraint more than the previous time. So one has to solve k times a problem that is roughly $O(k^4)$ smaller than Formulation 3.

The same approach can be used using other subtour elimination constraints. In particular, when using the GCS constraints, we get

Formulation 6. *Minimize Objective* (2.1) *subject to Constraints* (2.2), (2.3), (2.7), (2.9) *and* (2.26).

2.9 Conclusion

In this chapter, we studied the kSSP on different class of weighted digraphs. On non-negatively weighted digraphs, we showed how to improve the state-of-the-art algorithms in terms of running time / working memory in practice. Then, we adapted some of these methods to arbitrarily weighted digraphs with no negative cycles. The state-of-the-art algorithms and engineered ones proposed in this chapter were implemented and evaluated experimentally on various road and complex networks. Finally, we proposed some MIP models to solve the kSSP on arbitrarily weighted digraphs.

The main questions asked after this chapter concern the design of a kSSP algorithm with a preprocessing routing, enabling to drastically decrease the running time in practice. Another interesting question concerns the practical performance of the MIP models for arbitrarily weighted digraphs, and the design of hybrid methods allowing to use the fast kSSP algorithm of arbitrarily weighted digraphs with no negative cycles as a subroutine for arbitrarily weighted digraphs with small number of negative cycles.

On finding k (shortest) dissimilar paths in a graph

The similarity between two paths can be measured according to the proportion of arcs they share. We study the complexity of several variants of the problem of computing "dissimilar" paths (whose measure of similarity does not exceed a certain threshold) between two given vertices of a weighted directed graph. For four of the most studied measures in the literature, we give a unified and simple proof of the fact that finding k shortest dissimilar paths is NP-COMPLETE.
We then consider the problem of finding an alternative to one or more given paths. We show that finding a path that is dissimilar to another given path can be done in polynomial time for one of the four considered measures while it is NP-COMPLETE *for the three remaining measures. In addition, we show that if $k = 2$ paths are given, finding a new path that is dissimilar to the given ones is* NP-COMPLETE *even on DAGs for the four considered measures. Moreover, for the four considered measures, we show that if a path P is given, finding a shortest path among those that are dissimilar to P is* NP-COMPLETE *in DAGs.*
Finally, we present an alternative pseudo polynomial time algorithm allowing to find k shortest dissimilar paths for one of the considered similarity measures in arbitrarily weighted digraphs with no negative cycles.

3.1 Introduction

As described in Chapter 2, the k shortest simple paths problem aims at finding a shortest path, a second shortest path, *etc.*, a k^{th} shortest simple paths between a pair of source and destination node 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 many optimization problems. Let $D = (V, A)$ be a digraph, an s-t path is a sequence $(s = v_0, v_1, \cdots, v_l = t)$ of vertices starting with s and ending with t, such that $(v_i, v_{i+1}) \in A$ for all $0 \leq i < l$. A path P is called *simple* if all of its vertices are distinct, i.e, for every $i \neq j$, $v_i \neq v_j$.

Let $\ell : A \to \mathbb{R}^+$ be a length function over the arcs. For any path P, its *length* $\ell(P) = \sum_{e \in A(P)} \ell(e)$ is the sum of the lengths of its arcs. The top-k shortest paths is therefore a set containing a shortest s-t path, a second shortest s-t paths, *etc.* until a k^{th} shortest s-t path.

However, the k shortest simple paths are often quite "similar". Roughly, they often share a "large" proportion of their arcs. This is undesirable in many applications. For instance, in transportation networks, users may expect to have several options offering more diversity, a user prefers a shortest path, another user wants to avoid a traffic jam, a third one prefers to travel along the coast *etc.*

To deal with this issue, the problem of computing "dissimilar" (shortest) paths has been investigated. Several definitions of the similarity between two paths (including the Jaccard and the Max measures defined below [34]) were first proposed by Erkut and Verter [53], motivated by the transportation of hazardous materials where it is recommended to avoid residential areas and crowded routes.

Akgün *et al.* [3] proposed and analyzed a first basic solution, consisting in computing a huge set of shortest paths and then choosing a subset of these paths that are mutually dissimilar. In their experiments, this method scaled only on small transportation networks (about 300 vertices). The first scalable solutions were proposed by Abraham *et al.* [1] where a shortest path P is fixed, and "locally shortest" paths with limited intersection with P are requested (this corresponds to the Asymmetric measure defined below). However, except for the initial path P, this definition does not guaranty any mutual dissimilarity between the computed paths. A noticeable heuristic proposed in [1] is the penalty based approach. This heuristic adds a penalty on the arcs of the already chosen paths in order to limit the chances of falling back on the same paths.

Chondrogiannis *et al.* [35] offer both theoretical and empirical study of the problem. They formally proved that finding k shortest dissimilar paths is weakly NP-COMPLETE for both the Asymmetric measure and a new dissimilarity measure that they define (referred to as Min measure below). For these two measures, they proposed an exact pseudo-polynomial time algorithm, with several pruning techniques, that allows to find 4 dissimilar paths in a road network with 3,000 vertices in less than one second. They also proposed advanced heuristics enabling to scale on a road network with one million vertices while computing paths that are close to shortest ones in practice.

In this chapter, we further study the computational complexity of computing (shortest) dissimilar paths for four of the main measures studied so far. More formally, let P, P' be two s-t simple paths in D and let $X = \sum_{e \in A(P) \cap A(P')} \ell(e)$, i.e., the total length of the intersection of P and P'. The four considered measures are defined as follows.

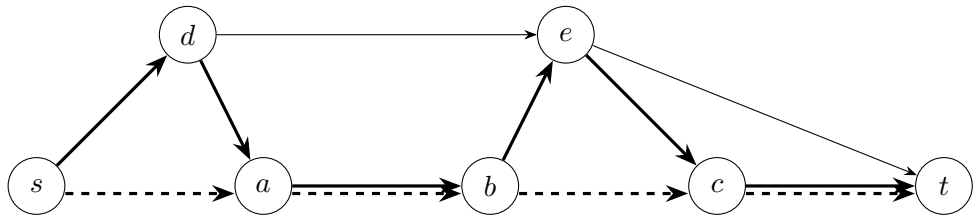


Figure 3.1 – Two s-t paths, $Q = (s, a, b, c, t)$ (dotted) and $Q' = (s, d, a, b, e, c, t)$ (thick). Every arc have length 1.

Name (Z)	Jaccard [53]	Asymmetric [1]	Min [34]	Max [53]
$S_Z(P, P') =$	$\frac{X}{\ell(P \cup P')}$	$\frac{X}{\ell(P)}$	$\frac{X}{Min\{\ell(P),\ell(P')\}}$	$\frac{X}{Max\{\ell(P),\ell(P')\}}$
[Fig. 3.1] $S_Z(Q, Q')$=	0.25	0.5	0.5	0.33

Let $\mathcal{S} = \{Asy, Jaccard, Min, Max\}$. Given one of the measures $Z \in \mathcal{S}$ and a threshold value $0 \leq \theta \leq 1$, two paths P and P' are said *θ-dissimilar* (or P' is said *θ-dissimilar to P* in the case of Asymmetric similarity) for the measure Z if $S_Z(P, P') \leq \theta$.

Our contributions. In Section 3.2, we study the problem of finding k shortest pairwise dissimilar paths. We give a unified and simple proof of the NP-completeness of this problem for each of the four similarity measures defined above. Then, in section 3.3, we study the problem of finding a (shortest) path that is dissimilar to a given set of paths. In particular, we show that if only one path P is initially given, computing a second path that is dissimilar to P for the Asymmetric measure can be done in polynomial time while it is NP-COMPLETE for the remaining measures (Min, Max and Jaccard). Then, we prove that finding a path dissimilar (for each of the considered four measures) to a given set of $k \geq 2$ paths is NP-COMPLETE on DAGs. In addition, for each of these four measures, we show that computing a shortest path among those dissimilar to a given path is NP-COMPLETE on DAGs. Finally, we propose an alternative pseudo polynomial time algorithm allowing to find k shortest dissimilar paths for the Asy measure in arbitrarily weighted digraphs with no negative cycles.

3.2 Finding k shortest dissimilar paths

In this section, we show that the problem of finding k shortest dissimilar paths is NP-COMPLETE for all the considered similarity measures.

More formally, given a digraph $D = (V, A)$ with length function $\ell : A \to \mathbb{R}^+$, a pair of vertices $(s, t) \in V \times V$, an integer $k \geq 2$, a threshold value $0 \leq \theta \leq 1$, k constants $L_1, L_2, \cdots, L_k$ and a similarity measure $Z \in \mathcal{S}$, the problem k-SHORTESTDISS(Z) of finding k shortest dissimilar paths asks to decide whether there exists k paths from s to t that are mutually θ-dissimilar with respect to Z, i.e, $S_Z(P_i, P_j) \leq \theta$ for $1 \leq i \leq j \leq k$ and such that $\ell(P_i) \leq L_i$ for $1 \leq i \leq k$.

Note that, for the extreme case where $\theta = 0$, the problem of finding k dissimilar paths (not necessarily the shortest) is the problem of finding k arc-disjoint paths, and it can be solved in polynomial time using any min cost flow algorithm.

Finding k shortest dissimilar paths has already been proved NP-COMPLETE for the Asymetric and Min measures [35]. Here we propose a simple and unified proof (for all considered measures).

Theorem 3.2.1. *For every $k \geq 2$ and $Z \in \mathcal{S}$, the* k-SHORTESTDISS(Z) *problem is* NP-COMPLETE *in the class of DAGs with a single source and a single sink.*

Proof. Let us first consider the case $k = 2$.

For every $Z \in \mathcal{S}$, the problem is clearly in NP. We prove the NP-hardness by a reduction from the MIN-MINDP problem. Given a digraph $D = (V, A)$ with length function $\ell : A \to \mathbb{R}^+$, two terminals $s, t \in V$ and a real number $\delta \in \mathbb{R}^+$ as inputs, the MIN-MINDP problem asks whether there exists two edge disjoint paths P and P' with $\ell(P) \leq \delta$. This problem is NP-COMPLETE [66].

Let $I = (D = (V, A), \ell, s, t, \delta)$ be an instance of the MIN-MINDP problem and let $I' = (D, s, t, k = 2, \theta = 0, \ell, L_1 = \delta, L_2 = n \cdot \max_{e \in A} \ell(e))$ be an instance of the k-SHORTESTDISS(Z) problem.

— If I is a positive MIN-MINDP instance, it means that there are two arc disjoint s-t paths P and P' such that $\ell(P) \leq \delta$. Let $P_1 = P$ and $P_2 = P'$. For every similarity measure $Z \in \mathcal{S}$, we have $S_Z(P_1, P_2) = 0$ since $A(P_1) \cap A(P_2) = \emptyset$, and so $\sum_{e \in A(P_1) \cap A(P_2)} \ell(e) = 0$. In addition, $\ell(P_1) \leq \delta = L_1$ and $\ell(P_2) \leq L_2$. So, I' is a positive k-SHORTESTDISS(Z) instance.

— If I' is a positive k-SHORTESTDISS(Z) instance, it means that there are two s-t paths P_1 and P_2 such that $\ell(P_1) \leq L_1$ and $S_Z(P_1, P_2) = 0$ for every similarity measure $Z \in \mathcal{S}$, In other words, P_1 and P_2 are arc-disjoint. Let $P = P_1$ and $P' = P_2$. P and P' are two arc-disjoint s-t paths. In addition $\ell(P) \leq L_1 = \delta$, so I is a positive MIN-MINDP instance.

We conclude that the 2-SHORTESTDISS(Z) problem is NP-HARD.

To extend the result to any $k \geq 2$, it is sufficient to add, to the digraph D in I', $k-2$ arc-disjoint s-t paths $P_3, \cdots, P_k$, each with length L_2 and to set $L_2 = L_i$ for all $2 \leq i \leq k$. □

3.3 Finding a path dissimilar to several given paths

In this section, we present our main results. First, we show that the problem of finding a path dissimilar to another given path can be solved in polynomial time for the Asymmetric measure. Then, we prove that the problem of finding a path dissimilar to two given paths is NP-COMPLETE. Finally, we show that finding a shortest path dissimilar to one given path is also NP-COMPLETE.

3.3.1 Finding a path dissimilar to another given path

First, let us start with the easiest variant of the problem that is the problem of finding a path dissimilar to another for the Asymmetrical measure. Given a digraph $D = (V, A)$ with $\ell : A \to \mathbb{R}^+$, two vertices $s, t \in V$, a threshold value $0 \leq \theta \leq 1$, an s-t simple path P and a similarity measure $Z \in \mathcal{S}$, DISS$(1, Z)$ is the problem of finding an s-t path Q that is θ-dissimilar to P using the measure Z.

Proposition 3.3.1. DISS$(1, Asy)$ *can be solved in the same time as any shortest-path algorithm.*

Proof. Let $\ell' : A \to \mathbb{R}^+$ be defined such that, for every $e \in A$, $\ell'(e) = \ell(e)$ if $e \in A(P)$, and $\ell'(e) = 0$ otherwise. Hence, a shortest s-t simple path Q is a solution of the DISS$(1, Asy)$ problem if and only if $\ell'(Q) = \sum_{e \in A(P) \cap A(Q)} \ell(e) \leq \theta \cdot \ell(P)$. □

Theorem 3.3.2. DISS$(1, Z)$ *is* NP-COMPLETE *for all* $Z \in \{Jaccard, Min, Max\}$

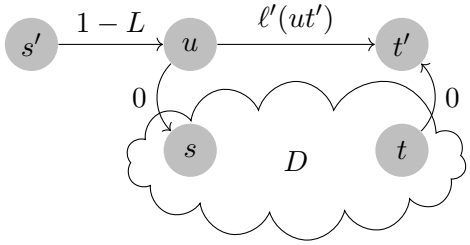


Figure 3.2 – Digraph D' defined from D (Theorem 3.3.2) with $\ell'(ut') = 1$ for the Min measure and $\ell'(ut') = 0$ for the Max and Jaccard measures.

Proof. For every $Z \in \{Jaccard, Min, Max\}$, the problem is clearly in NP, so we only prove the NP-hardness by a reduction from the LONG-PATH problem. Given a digraph $D = (V, A)$ with a length function $\ell : A \to \mathbb{R}^+$, two terminals $s, t \in V$ and a real number $L \in \mathbb{R}^+$ as inputs, the LONG-PATH problem asks whether there exists an s-t simple path Q with $\ell(Q) \geq L$. This problem is NP-COMPLETE [105]. Moreover, it remains NP-COMPLETE when $L \leq 1$ (by dividing the length of each arc by $M = \sum_{a \in A} \ell(a)$).

Let $I = (D = (V, A), \ell, s, t, L)$ be a LONG-PATH instance with $0 < L \leq 1$. Let $D' = (V \cup \{s', u, t'\}, A \cup \{s'u, ut', us, tt'\})$ with lengths $\ell'(a) = \ell(a)$ for every $a \in A$, $\ell'(s'u) = 1-L$, the value of $\ell'(u, t')$ depends on the considered measure and will be specified later, $\ell'(us) = 0$ and $\ell'(tt') = 0$ (see Figure 3.2). Let also $I' = (D', \ell', P, \theta = 1 - L)$, with $P = \{s', u, t'\}$, be an instance of the DISS$(1, Z)$ problem.

First, let us consider the Min measure and let $\ell'(ut') = L$.

— If I is a positive LONG-PATH instance, then there is an s-t path $R = (s, \cdots, t)$ of length at least L in D. Let $Q = (s', u, s, \cdots, t, t')$ be the concatenation of s', u, R and t'. Note that $\ell'(P) = 1$ and $\ell'(Q) \geq 1$. We have $S_{Min}(P, Q) = \frac{\ell'(s'u)}{\min\{\ell'(P), \ell'(Q)\}} = \frac{1-L}{1} = \theta$, and so I' is a positive DISS$(1, Min)$ instance.

— If I' is a positive DISS$(1, Min)$ instance, then there is an s-t path Q s.t. $S_{Min}(P, Q) = \frac{\ell'(P \cap Q)}{\min\{\ell'(P), \ell'(Q)\}} \leq \theta$. Since $\ell'(P \cap Q) = \ell'(s'u) = 1 - L = \theta$, we have $\min\{\ell'(P), \ell'(Q)\} \geq 1$ and so $\ell'(Q) \geq 1$ (since $\ell'(P) = 1$). Let R be the subpath of Q starting from s and ending at t, i.e., $R = (s, \cdots, t)$ and $\ell(R) = \ell'(R) = \ell'(Q) - (1-L) \geq L$ (since R is a simple path and $\ell'(Q) \geq 1$). Therefore, I is a positive LONG-PATH instance.

We conclude that DISS$(1, Min)$ is NP-Hard.

Using a similar construction, the NP-hardness of DISS$(1, Max)$ and DISS$(1, Jaccard)$ can be proved.

Precisely, for both the $Jaccard$ and Max measure, it is sufficient to keep the same reduction as before but setting $\ell'(ut') = 0$.

— If I is a positive LONG-PATH instance, then there is an s-t path $R = (s, \cdots, t)$ of length at least L in D. Let $Q = (s', u, s, \cdots, t, t')$ be the concatenation of s', u, R and t'. Note that $\ell'(P) = 1 - L$ and $\ell'(Q) \geq 1$.

In the case of the Max measure, we get that $S_{Max}(P, Q) = \frac{\ell'(P \cap Q)}{\max\{\ell'(P), \ell'(Q)\}} \leq \frac{1-L}{1} = \theta$, and so I' is a positive DISS$(1, Max)$ instance.

In the case of the Jaccard measure, we get that $S_{Jaccard}(P, Q) = \frac{\ell(P \cap Q)}{\ell(P \cup Q)} = \frac{1-L}{1-L+\ell(R)} \leq 1 - L = \theta$ (since $\ell(R) \geq L$), and so I' is a positive DISS$(1, Jaccard)$ instance.

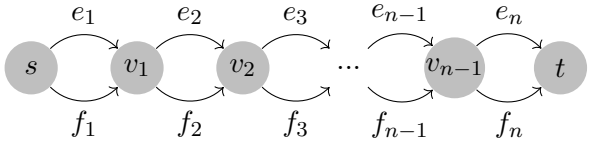


Figure 3.3 – Digraph $D_S = (V, A)$ defined from $\mathcal{S} = \{x_1, \cdots, x_n\}$. For all $1 \leq i \leq n$, $\ell(e_i) = x_i$. For all $1 \leq i \leq n$, we have $\ell(f_i) = x_i$ in the proof of Theorem 3.3.3 and $\ell(f_i) = M \cdot x_i$ with $M > 1$ in the proof of Theorem 3.3.4.

— If I' is a positive DISS$(1, Max)$ instance, then there is an s-t path Q s.t. $S_{Max}(P, Q) = \frac{\ell'(P \cap Q)}{\max\{\ell'(P), \ell'(Q)\}} \leq \theta$. Since $\ell'(ut') = 0$, we have that $\ell'(P) = \ell'(su) = \ell'(P \cap Q) = 1 - L = \theta$ and $\ell'(Q) \geq \ell'(P)$. Since $S_{Max}(P, Q) \leq \theta$, $\ell'(Q) \geq 1$. Let R be the subpath of Q starting from s and ending at t, i.e., $R = (s, \cdots, t)$ and $\ell(R) = \ell'(R) = \ell'(Q) - (1 - L) \geq L$ (since Q is a simple path). Therefore, there is a path from s to t in D of length at least L and I is a positive LONG-PATH instance.

— If I' is a positive DISS$(1, Jaccard)$ instance, then there is an s-t path Q s.t. $S_{Jaccard}(P, Q) = \frac{\ell'(P \cap Q)}{\ell'(P \cup Q)} \leq \theta$. By construction, $\ell'(P \cup Q) = \ell'(Q)$. Since, moreover $\ell'(P \cap Q) = \ell'(s'u) = 1 - L = \theta$, then $\ell'(Q) \geq 1$. Let R be the subpath of Q starting from s and ending at t, i.e., $R = (s, \cdots, t)$ and $\ell(R) = \ell'(R) = \ell'(Q) - \theta \geq 1 - (1 - L) = L$ (since Q is a simple path). Therefore, there is a path from s to t in D of length at least L and I is a positive LONG-PATH instance.

We conclude that DISS$(1, Max)$ and DISS$(1, Jaccard)$ are NP-Hard. □

3.3.2 Finding a path dissimilar to several given paths

Given a digraph $D = (V, A)$ with $\ell : A \to \mathbb{R}^+$, two vertices $s, t \in V$, a threshold value $0 \leq \theta \leq 1$, k s-t simple paths $P_1, \cdots, P_k$ and a similarity measure $Z \in \mathcal{S}$, DISS(k, Z) is the problem of finding an s-t path Q that is θ-dissimilar to P_i for all $i \leq k$ using the measure Z.

Theorem 3.3.3. *For every $k \geq 2$ and $Z \in \mathcal{S}$, the* DISS(k, Z) *problem is* NP-COMPLETE *even if D is a Directed Acyclic Graph (DAG) with a single source and a single sink.*

Proof. Let $Z \in \mathcal{S}$. Let us first consider the case $k = 2$. We use a reduction from the PARTITION problem. Recall that the PARTITION problem takes as input a multiset $S = \{x_1, ..., x_n\}$ of positive integers and asks whether there exists a partition (X, Y) of S such that $\sum_{x \in X} x = \sum_{x \in Y} x = h$ where $2h = \sum_{x \in S} x$ (so $\sum_{x \in S} x$ is even). The PARTITION problem is weakly NP-COMPLETE [60].

Let $D_S = (V, A)$ be the DAG defined such that $V = \{s = v_0, v_1, \cdots, v_{n-1}, v_n = t\}$ and, for every $1 \leq i \leq n$, let us add arcs $e_i = v_{i-1}v_i$ and $f_i = v_{i-1}v_i$ with length $\ell(e_i) = \ell(f_i) = x_i$ (see Figure 3.3). Let P_1 be induced by $\{e_i \mid 1 \leq i \leq n\}$, P_2 be induced by $\{f_i \mid 1 \leq i \leq n\}$ (note that $\ell(P_1) = \ell(P_2) = 2h$) and let $\theta = 1/2$.

Note that there is a one-to-one mapping between the s-t simple paths and the bipartitions of $\{1, \cdots, n\}$. Indeed, let P be any such path. Then, for every $1 \leq i \leq n$, path P goes through exactly one of e_i or f_i. Let $X_P = \{1 \leq i < n \mid e_i \in A(P)\}$ and $Y_P = \{1 \leq i < n \mid f_i \in A(P)\}$. Clearly, (X_P, Y_P) is a partition of $\{1, \cdots, n\}$. Reciprocally, let (X, Y) be any partition

of $\{1, \cdots, n\}$. Then, let P_{XY} be the path induced by $\{e_i \mid i \in X\} \cup \{f_i \mid i \in Y\}$. Clearly, P_{XY} is an s-t simple path.

First, we consider only the three similarity measures Asy, Min and Max. Note that every s-t simple path has length $2h$ and therefore, for every s-t simple paths P and R, $S_{Asy}(P, R) = S_{Asy}(R, P) = S_{Min}(P, R) = S_{Max}(P, R)$. Hence, all similarity measures in $\{Asy, Min, Max\}$ are equivalent.

By construction, for every bipartition (X, Y) of $\{1, \cdots, n\}$ (equivalently, for every s-t simple path P_{XY}), $\ell(P_1 \cap P_{XY}) = \sum_{i \in X} x_i$ and $\ell(P_2 \cap P_{XY}) = \sum_{i \in Y} x_i$. Since $\ell(P_1 \cap P_{XY}) = S_Z(P_1, P) \cdot 2h$ and $\ell(P_2 \cap P_{XY}) = S_Z(P_2, P) \cdot 2h$, it follows that $(D_S, \ell, s, t, \frac{1}{2}, P_1, P_2)$ admits a s-t simple path P with $S_Z(P_1, P) \leq \frac{1}{2}$ and $S_Z(P_2, P) \leq \frac{1}{2}$ if and only if S admits a balanced partition. So the DISS$(2, Z)$ problem is NP-Hard for all $Z \in \{Asy, Min, Max\}$.

Concerning the $Jaccard$ measure, i.e, the DISS$(2, Jaccard)$ problem, using the same construction proposed above but with $\theta = \frac{1}{3}$ one can prove that the described reduction is valid.

Finally, to extend the result to any $k \geq 2$, it is sufficient to add, to D_S, $k - 2$ arc-disjoint s-t paths $P_3, \cdots, P_k$ with length = $2h$. □

3.3.3 Shortest path dissimilar to one given path

We now study the problem of finding a path of bounded length that is dissimilar to a set of k given paths. By Theorem 3.3.3, this problem is NP-COMPLETE (without bounding the length) whenever $k \geq 2$. So, let us study the case for $k = 1$. By Theorem 3.3.2, the problem is NP-COMPLETE for $Z \in \{Min, Max, Jaccard\}$. So, the only remaining case is the Asy measure. In contrast with Proposition 3.3.1, we prove that SDISS$(1, Asy)$ is NP-COMPLETE. Moreover, this result holds on DAGs and for every $Z \in \mathcal{S}$.

Precisely, the SDISS$(1, Z)$ problem takes as input a tuple $(D, \ell, s, t, \theta, L, P)$ where $D = (V, A)$ is a directed graph with $\ell : A \to \mathbb{R}^+$, $s, t \in V$, $0 \leq \theta \leq 1$, $L \in \mathbb{R}^+$, and P is an s-t simple path. It aims at deciding whether there exists an s-t simple path Q that is θ-dissimilar to P such that $\ell(Q) \leq L$.

Theorem 3.3.4. *Let $Z \in \mathcal{S}$. The* SDISS$(1, Z)$ *problem is* NP-COMPLETE *in the class of DAGs with a single source and a single sink.*

Proof. The problem is clearly in NP, so we prove its NP-hardness by a reduction from the PARTITION problem.

Let $S = \{x_1, ..., x_n\}$ be an instance of the PARTITION problem and $2h = \sum_{i=1}^{n} x_i$. Let $M > 1$. Let $D_S = (V, A)$ be the DAG defined such that $V = \{s = v_0, v_1, \cdots, v_{n-1}, v_n = t\}$ and, for every $1 \leq i \leq n$, let us add arcs $e_i = v_{i-1}v_i$ and $f_i = v_{i-1}v_i$ with length $\ell(e_i) = x_i$ and $\ell(f_i) = M \cdot x_i$ respectively (see Figure 3.3). Note that, if we want to avoid parallel arcs, we can simply subdivide each arc $a \in A$ into two arcs with length $\ell(a)/2$ each.

Let P be the s-t simple path that consists of arcs $e_1, \cdots, e_n$ and so $\ell(P) = 2h$. Note that, since $M > 1$, $\ell(P) \leq \ell(P')$ for every s-t simple path P'. Finally, let $L = h(M + 1)$ and let $\theta = 1/2$ for Asy and Min measures, $\theta = \frac{1}{M+1}$ for the Max measure and $\theta = \frac{1}{M+2}$ for the $Jaccard$ measure.

As in the proof of Theorem 3.3.3, it can be shown that there is an s-t simple path Q with $\ell(Q) \leq L$ and Q is θ-dissimilar from P if and only if S is a positive instance of the PARTITION problem. □

3.4 Algorithms for finding k (shortest) dissimilar paths

As described in the introduction of this chapter, several exact algorithms and heuristics were proposed to find k shortest dissimilar paths in practice. The most recent algorithms were proposed by Chond. et al. [35] where they presented two novel algorithms called MULTIPASS and ESX. MULTIPASS is the first pseudo polynomial time algorithm allowing to find k shortest dissimilar paths for the Min measure. MULTIPASS algorithm is similar to Dijkstra's algorithm, i.e, it examines paths starting from s in increasing order of their length and expands every path from s to a vertex v in V that satisfy some conditions (being dissimilar to each of the paths that have already been added to the output and not "dominated" (see definition below) by other s-v path already computed). The complexity of MULTIPASS algorithm is in $O(kn^2(\theta M)^{2k})$ where M is the sum of the length of the arcs of the input digraph, $M = \sum_{a \in A} \ell(a)$. Is is also shown experimentally in [35] that MULTIPASS is the fastest exact algorithm to find k shortest dissimilar paths in practice. Moreover, the ESX heuristic is similar to the penalty based approach of Abraham et al. [1]. Precisely, it starts by computing a shortest s-t path P_0. Then, it iteratively removes an arc a of P_0 with minimum length, and computes a new shortest path P_1 in the digraph after removing the arc a. This process is repeated until a new shortest path P_1 became dissimilar to P_2. Then, the same routine is applied on P_2 until finding P_3 that is dissimilar to P_1 and P_2, *etc*. until finding a k^{th} path P_k.

Our contribution We propose an alternative pseudo polynomial time algorithm allowing to find k shortest dissimilar paths for the Asy and Min measures [*]. The advantage of our algorithm is that it remains valid for arbitrarily weighted digraphs with no negative cycles.

3.4.1 Pseudo Polynomial algorithm

In this section, we describe our Pseudo Polynomial time algorithm finding k shortest dissimilar paths for the Asy and Min measures. Recall that our algorithm takes as input a weighted digraph with unitary length function $\ell : A \longrightarrow \mathbb{Z}$ with no negative cycles, two vertices s and t and finds k shortest θ dissimilar s-t paths. For the sake of simplicity, we show how the algorithms works for the Asy measure, and we describe later why the same algorithm works for the Min measure. Also, we first consider the problem of finding a shortest path dissimilar to a given set of δ paths (SDISS(δ, Z)), i.e, it takes as input a weighted digraph with unitary length function $\ell : A \longrightarrow \mathbb{Z}$ with no negative cycles, two vertices s and t, a constant L and a set $\mathcal{P} = \{P_1, P_2, ..., P_\delta\}$ of s-t paths, and asks whether there is an s-t path P of length $\ell(P) \leq L$ such that P is dissimilar for the measure Z (in what follows, we mainly consider $Z = Asy$) to each path in $\mathcal{P}$.

The algorithm Let $\mathcal{P} = \{P_1, P_2, ..., P_\delta\}$ be a set of s-t paths and let P and P' be two s-t paths, we say that P dominates P' ($P \preccurlyeq P'$) if and only if $\ell(P) \leq \ell(P')$ and $S_{Asy}(P_i, P) \leq S_{Asy}(P_i, P')$ for every path $P_i \in \mathcal{P}$. Note that, the following properties are valid :

- — Reflexivity : $P \preccurlyeq P$
- — Stability under concatenation : if $P \preccurlyeq P'$ then $P.(u, v) \preccurlyeq P'.(u, v)$ for each $v \in V \setminus P$
- — Transitivity : $P \preccurlyeq P'$ and $P' \preccurlyeq P''$ then $P \preccurlyeq P''$

[*]. In fact, we designed our pseudo polynomial algorithm before noticing that MULTIPASS had already been published.

Moreover, for each vertex v in V, an s-v path P is called *good* if and only if $\ell(P) \leq L$ and $S_{Asy}(P_i, P) \leq \theta$ for every $P_i \in \mathcal{P}$.

Our pseudo-polynomial algorithm, whose pseudocode presented in Algorithm 3.1, is similar to Bellmand-Ford-Moore's shortest path algorithm. It relaxes n times all the arcs of the digraph, except that instead of keeping the distance from s to v for every vertex v, it keeps a table $T(v)$ containing a set of *good* s-v paths. The relaxation procedure consists of replacing a path P by another P' only if P is dominated by P' ($P' \preccurlyeq P$).

Input A digraph $D = (V, A)$, $s, t \in V$, $\mathcal{P} = \{P_1, P_2, ..., P_\delta\}$, a similarity threshold $0 \leq \theta \leq 1$ and a constant L
Output an s-t good path if any
for every $v \in V$ **do** ▷ Initialization
    $T(v) = []$
$T(s) = [((s), 0, S_{Asy}(P_1, (s)) = 0, \cdots, S_{Asy}(P_\delta, (s)) = 0)]$. ▷ Each element of $T(u)$ consists of a path P from s to u, its length and its similarity with respect to each path in $\mathcal{P}$
for $i = 1$ to n **do**
    for every arc $(u, v) \in A$ **do**
        Dominated $= False$
        for every element $e = (pref, \ell(pref), S_1, \cdots, S_\delta)$ in $T(u)$ **do**
            $pref^* = pref.(u, v)$
            if $pref^*$ is *good* **then**
                for every $e' = (pref', \ell(pref'), S'_1, \cdots, S'_\delta)$ in $T(v)$ **do**
                    if $pref' \preccurlyeq pref^*$ **then**
                        Dominated $= True$
                        break
                    if $pref^* \preccurlyeq pref'$ **then**
                        delete e' from $T(v)$
                if not Dominated **then**
                    add $(pref^*, \ell(pref^*), S_{Asy}(P_1, pref^*), \cdots, S_{Asy}(P_\delta, pref^*))$ to $T(v)$
if $T(t)$ is not empty **then**
    return any element of $T(t)$

Algorithm 3.1 – PseudoPoly, a pseudo polynomial time algorithm to find a shortest path dissimilar to a set of given paths

Correcteness of the algorithm

Claim 3.4.1. *If $SDiss(k, Asy)$ has a solution then it has a simple solution.*

Claim 3.4.1 can be proved by observing that removing the cycles from a path won't increase its length, neither its similarity with the given paths. So, if a non-simple path is good, removing its cycles keeps it good.

Theorem 3.4.2. *Pseudo-Poly returns a good path if any.*

Proof. The proof of correctness of our pseudo polynomial algorithm is similar to the proof of correctness of Bellmand-Ford-Moore's algorithm for finding a shortest path.

Let $v \in V$ and let $T_i(v)$ be the table $T(v)$ after the i^{th} iteration. By induction on i, we will prove that there is an s-v path P that is *good* and uses at most i arcs if and only if $T_i(v)$ contains an s-v path P' s.t. $P' \preccurlyeq P$.

Base case : for $i = 0$, there is only one *good* path using 0 arcs, that is $P = \{s\}$ and it is in $T_0(s)$ so the result holds. And for every vertex $v \neq s$ there is no *good* paths using 0 edges, and $T_0(v)$ is empty for every $v \neq s$. So the claim is satisfied for $i = 0$.

Inductive step : Suppose that for every vertex $u \in V$, there is an s-u path Q that is *good* and uses at most $i-1$ arcs if and only if there is an s-u path $Q' \in T_{i-1}(u)$ s.t. $Q' \preccurlyeq Q$.

We distinguish two cases :

— Case 1 : There is a *good* s-v path P that uses at most i arcs.
  Let u be the vertex just before v on P, and let Q be the subpath of P going from s to u, i.e., $P = Q.v$. Then Q uses at most $i-1$ arcs and Q is *good* (as P is *good*). By the induction hypothesis, $T_{i-1}(u)$ contains an s-u path Q' s.t. $Q' \preccurlyeq Q$. By the stability under concatenation of $\preccurlyeq$, $P' = Q'.(u,v) \preccurlyeq Q.(u,v) = P$.
  At iteration i, the algorithm either will add P' to $T_i(v)$ (line 16) and the claim is satisfied as $P' \preccurlyeq P$ or P' won't be added because $T_{i-1}(v)$ has a path P'' s.t. $P'' \preccurlyeq P'$ (line 12-13). In this case, $T(v)$ has a path P'' s.t. $P'' \preccurlyeq P$ (by transitivity of $\preccurlyeq$).

— Case 2 : There is no *good* s-v path that uses at most i arcs.
  In this case, no good s-v path using at most $i-1$ arcs exists (otherwise we are in case 1). So, by the induction hypothesis $T_{i-1}(v)$ does not contain any *good* s-v path. And since the algorithm only adds *good* paths to the tables (line 10). We deduce that $T_{i-1}(v)$ is empty.
  Let $N^-(v)$ be the set of in-neighborhoods of v and let $u \in N^-(v)$. At iteration i the algorithm iterates over every arc (u,v).
  — If there is no *good* s-u path using at most $i-1$ arcs, then $T_{i-1}(u)$ is empty. So, no paths are added to $T_i(v)$ after considering the arc (u,v).
  — There is a *good* s-u path using $\leq i-1$ arcs, so $T_{i-1}(u)$ contains a *good* s-u path P. but after iterating the arc (u,v) the path $P.(u,v)$ is not *good* anymore (otherwise we are in case 1), so it won't be added to $T_i(v)$. Therefore $T_i(v)$ remains empty.
  So, if no *good* s-v path using $\leq i$ arcs exists, $T_i(v)$ is empty.

Therefore the claim is satisfied after the i^{th} iteration. Then the claim holds for each $i \in \mathbb{N}$.

Claim 3.4.1 ensures that if there is no simple *good* path, then no *good* path exists. The longest path we can find has at most $n-1$ arcs. Therefore we only need to iterate $n-1$ times to ensure the correctness of the algorithm.

Note that, no restriction of the length of the arcs are required in this proof (except of being integers and the absence of negative cycles). In other words, the pseudo polynomial algorithm is also valid on arbitrarily weighted digraphs with no negative cycles. □

The complexity of the algorithm

Claim 3.4.3. *If the length function ℓ has integer values bounded by a constant C, and $\mathcal{P} = \{P_1, \cdots, P_\delta\}$ then, for each table $T_i(v)$ of Algorithm 3.1, $|T_i(v)| \leq (n.C)^{\delta+1}$ for each $v \in V$ and $1 \leq i \leq n$*

Proof. Let $v \in V$, and let P be an s-v path in $T_i(v)$ of length $0 \leq \ell(P) \leq n.C$ where C is the length of the longest arc in A, i.e, $\ell(a) \leq C$ for $a \in A$.

Following the fact that PseudoPoly agorithm (Algorithm 3.1) does not keep two paths in $T_i(v)$ if one of them dominates the second (lines 11-16). Then, each two entry e and e' in $T_i(v)$ have either different length, of different similarity of one of the paths in $\mathcal{P} = \{P_1, \cdots, P_\delta\}$. So, for each length value $0 \leq l \leq n.C$, there is at most $(n.C)^\delta$ corresponding elements in $T_i(v)$ for $0 \leq i \leq n$. Therefore, $|T_i(v)| \leq n.C(nC)^\delta = (nC)^{\delta+1}$. □

Theorem 3.4.4. *If the length function ℓ has integer values bounded by a constant C. The Pseudo-Poly algorithm solves $SDiss(\delta, Asy)$ in $O(n^{2\delta+5} \cdot C^{2\delta+2})$*

Proof. The algorithm has four main loops, the first has n iteration multiplied by the number of arcs that is m. Both are multiplied by the square of the number of entries of the biggest table among $T_i(v)$ for $1 \leq n$ and $v \in V$, that is, by Claim 3.4.3, bounded by $(nC)^{\delta+1}$. Therefore the complexity of Pseudo-Poly-Diss algorithm is bounded by $O(nm((nC)^{\delta+1})^2) = O(n^{2\delta+5} \cdot C^{2\delta+2})$. □

3.5 Conclusion

In this chapter, we studied several versions of the problem of finding (shortest) dissimilar paths in a digraph considering four similarity measures. We also reviewed the state of the art results and algorithms and proposed an alternative pseudo polynomial algorithm allowing to find k shortest dissimilar paths in an arbitrarily weighted digraphs. The results of this chapter are summarized in Table 3.1.

Problem	Objective	Complexity
k-SHORTESTDISS(Z)	Finding k shortest dissimilar paths	NP-COMPLETE for $k \geq 2$ for all $Z \in \mathcal{S}$
DISS(1,Z)	Finding a path dissimilar to a given path	Polynomial for $Z = Asy$ and NP-COMPLETE for $Z \in \{jaccard, Min, Max\}$
DISS(k, Z)	Finding a path dissimilar to k given path	NP-COMPLETE for $k \geq 2$ on DAGs for all $Z \in \mathcal{S}$
SDISS(k, Z)	Finding a shortest path dissimilar to k given path	NP-COMPLETE for $k \geq 1$ on DAGs for all $Z \in \mathcal{S}$

Table 3.1 – The complexity of different variants of the problem of finding dissimilar paths, all the paths of this table are s-t paths

An interesting question is whether there is a similarity measure for which the problem of finding k dissimilar paths can be solved in polynomial time. Another interesting question regards the accuracy of these similarity measures for real life applications.

On finding k earliest arrival journeys in public transit networks

Journey planning in (schedule-based) public transit networks has attracted interest from researchers in the last decade. In particular, many algorithms aiming at efficiently answering queries of journey planning have been proposed. However, most of the proposed methods give the user a single or a limited number of journeys in practice, which is undesirable in a transportation context.
In this chapter, we consider the problem of finding k earliest arrival time journeys in public transit networks from a given origin to a given destination, i.e, an earliest arrival journey from the origin to the destination, a second earliest arrival journey, etc. until the k^{th} earliest arrival journey.
For this purpose, we propose an algorithm, denoted by Yen - Public Transit (Y-PT), that extends to public transit networks the algorithm proposed by Yen to find the top-k shortest simple paths in a graph. Moreover, we propose a more refined algorithm, called Postponed Yen - Public Transit (PY-PT), enabling an important speed up in practice.
Our experiments on several public transit networks show that, in practice, PY-PT is faster than Y-PT by an order of magnitude.
Keywords : *Journey planning, shortest path, routing, timetables.*

4.1 Introduction

In the context of multimodal transportation, journeys planning in (schedule-based) public transit networks and accelerating queries for efficient journey planning is a long-standing problem [21]. In the last decade, many algorithms have been developed not only to answer efficiently basic queries like a quickest or an earliest arrival journey, but also to optimize additional criteria like the number of transfers, the cost of the trip, *etc.* or even to offer Pareto optimal solutions combining several criteria [21, 43, 46].

A *transit network* is a set of stops (such as bus stops or trains stations), a set of routes (such as bus, tramway, ferries, metro or train lines), and a set of trips. Trips correspond to individual vehicles that visit the stops along a certain route at a specific time of the day. Trips can be further subdivided into sequences of elementary connections, each given as a pair of (origin/destination) stops and (departure/arrival) times between which the vehicle travels without stopping. In addition, footpaths model walking transfers between nearby stops. A *journey* is a sequence of trips one can take within a transit network (also referred to as a transportation network or a *timetable*).

As we have already seen, a road network can be modelled using a weighted directed graph where crossroads are represented by vertices and routes by arcs with length corresponding to the distances or the travel time between crossroads. So, finding k "best" (shortest, fastest or cheapest) paths from a given origin to a given destination in a road network is straightforward using any kSSP algorithm. Unfortunately, this problem becomes harder in public transit networks. First, because public transit networks are time dependent, i.e., certain segments of the network can only be traversed at specific times. Second, several additional optimization criteria are considered in public transit network such as the arrival time, the departure time, the number of transfers, *etc.*

Journey planning queries in public transit networks As described in the Introduction of this thesis (Section 1.6 page 29), various algorithms were designed to efficiently answer optimal journeys queries in public transit networks. For instance, the *Connection Scan Algorithm (CSA)* [46] is the fastest algorithm, without any preprocessing routine, enabling to find an earliest arrival journey from o to d departing after t_0 in a public transit network. With the help of a heavy preprocessing routine, the Transfer Patterns algorithm [20] can achieve a tremendous speed up with respect to the CSA. Besides, *Round Based Public Transit Routing (RAPTOR)* [43] is the fastest algorithm (also without any preprocessing routine) enabling to compute a Pareto optimal set of journeys optimizing the arrival time and the number of transfers of a journey. A survey on journey planning in public transit networks is published recently by Bast et al. [21].

Related work In [119], Vo et al. proposed a time dependent graph modeling of a bus network and they adapted Yen's algorithm to find alternative journeys in this network model. Precisely, they select a set of alternative journeys (journeys sharing only a limited part of their common edges) among those given by Yen's adaptation.

As described before, Yen's algorithm uses Dijkstra's algorithm as a basic brick to compute shortest detours of a given path. Analogously, Vo et al. [119] used a standard time-dependent shortest path (TDSP) algorithm [107] to compute earliest detours of a journey in a bus network. They evaluated their method on a single network of around $4\,000$ stops and $8\,000$ connections, and showed that their algorithm needs, on average, around 1 second to find 5 journeys.

On the other hand, Scano et al. [104] proposed a labelled directed graph modeling of a transportation network where a label of an arc is an object composed of the transportation mode (foot,

car, bus, *etc*.) and a travel time. This model merges a road and a public transport network together. Then, it is shown how the k shortest path algorithms can be adapted for this model. Specifically, they adapted Yen's and Eppstein's algorithm to work on their model. In both algorithms, a Dijkstra-like algorithm called Dijkstra Regular Language Constraint (DRegLC) [19] is used to answer earliest arrival journeys queries. Moreover, an Iterative Enumeration Algorithm (IEA) is proposed to extract only simple journeys using Eppstein's algorithm. i.e, using Eppstein's k shortest paths algorithm as an iterator and then selecting the simple corresponding journeys (a journey is *simple* if it does not visit a stop more than once).

Experimentally, Scano et al. showed that their IEA is faster than Yen's straightforward adaptation on the transportation network of Toulouse ($75\,000$ nodes, $500\,000$ road edges and $43\,000$ public transport edges). On this network, the average running time of Yen's adaptation to find 100 journeys is 250 seconds while it is 0.6 seconds using their refined IEA. However, IEA is not a polynomial-time algorithm, and its memory consumption is too high [104]. In addition, using the labelled directed graph model described in [104] may cause a duplication of the public transit part in practice, i.e, a large number of journeys given by the algorithms proposed in [104] may only differ on the foot-path part while sharing the exact same public transit part. This is usually undesirable as in most of the applications, diverse public transit journeys are requested.

Our contribution In this chapter, we aim at answering the k earliest arrival journeys queries from a given origin to a given destination in a public transit network. For this purpose, we use the timetable model of public transit networks, i.e, the well-known common model used in [21, 46, 43]. First, we propose a performant adaptation of Yen's k shortest simple paths algorithm to public transit networks (Yen - Public Transit, Y-PT algorithm). In contrast with [104, 119], we use the Connection Scan Algorithm (CSA) to answer earliest arrival journey queries in our algorithm.

Our main contribution is a novel algorithm, called Postponed Yen's algorithm for Public Transit networks (PY-PT). With the help of a lower bound on the arrival time of a detour journey (a journey that may be one of the k earliest arrival journeys), PY-PT postpones the effective computation of such detour (and so the corresponding earliest arrival journey queries using CSA) with the aim of skipping it.

Our experimental results on several train and public transit networks show that the running time of our adaptation of Yen's algorithm is acceptable in practice. Moreover, on the same dataset, the PY-PT algorithm performs 10 to 30 times faster than the Y-PT algorithm on average.

Note that the output journeys are not guaranteed to be dissimilar. However, this can be achieved by filtering the output journeys and selecting a subset of ($k' \leq k$) journeys that are sufficiently dissimilar. This work is motivated by adapting the k shortest simple path problem to a public transit context. So, an interesting superset of journeys will be proposed and one may filter it to obtain satisfying solutions regarding the considered constraints (price, walking distance, accessibility, dissimilarity, *etc*.)

4.2 Preliminaries

In this chapter, we use the same graph notations and definitions described in the Introduction (see Section 16 page 16). In what follows, we formalize the inputs and describe algorithms related to journeys planning in public transit networks.

4.2.1 Timetable - definitions and notations

In this section, we describe the data structures used by the CSA, with the same formalization as in [46]. Then we will describe briefly the Connection Scan Algorithm and one of its variant called the profile CSA (PCSA).

Timetable A timetable represents for one specific day the vehicles that exist (train, bus, tram, ferry, ...), when they travel, where they travel, and how a passenger can go from one vehicle to another. Formally, a timetable is a quadruple $\mathcal{T} = (S, T, C, F)$ of stops S, trips T, connections C and footpaths F :

— A *stop* is a position outside a vehicle where a passenger can wait. At a stop (and only at a stop) a vehicle can halt and passengers can leave or get on.

— A *trip* is defined by a vehicle going through stops at fixed times. Precisely, a trip is a scheduled vehicle, i.e, a journey done by a unique vehicle from a starting stop to a last stop at a fixed time.

— A *connection* is a vehicle going from one stop to another with no intermediate stops. Formally, a connection c is a quintuple $(c_{dep\_stop}, c_{arr\_stop}, c_{dep\_time}, c_{arr\_time}, c_{trip})$ whose attributes are the departure stop, the arrival stop, the departure time, the arrival time and the trip of c, respectively. A connection must respect two conditions : (1) it cannot be a self loop, i.e, $c_{dep\_stop} \neq c_{arr\_stop}$ and (2) it has a non-zero travel time, i.e, $c_{dep\_time} < c_{arr\_time}$.

— A *footpath* is used to model a transfer, i.e, how to get from one vehicle to another. Formally, a footpath f is a triple $(f_{dep\_stop}, f_{arr\_stop}, f_{dur})$ whose attributes are the departure stop, the arrival stop and the duration of the footpath, respectively. Note that, footpaths are neither trips, nor connections.

Note that all the connections of a trip form a sequence $c^1, c^2 \dots\ c^\phi$, such that $c^i_{arr\_stop} = c^{i+1}_{dep\_stop}$ and $c^i_{arr\_time} < c^{i+1}_{dep\_time}$ for all $0 \leq i \leq \phi$.

Going from a connection c to a connection c' with $c_{trip} \neq c'_{trip}$ is possible if and only if there is a footpath f^t from c_{arr_stop} to c'_{dep_stop} such that c' is reachable via f^t, i.e, $f^t_{dur} \leq c'_{dep\_time} - c_{arr\_time}$. A loop is introduced on each stop to allow a passenger to get off at a stop and take another trip going through this stop.

Journeys A journey describes how a passenger can travel through a public transit network. It is made of *legs* that are sequences of connections of the same trip. Formally, a journey is a sequence of alternating footpaths and legs $J = (f^0, l^0, f^1, l^1 \dots f^{r-1}, l^r, f^r)$, where $l^i = (c^i_0, \cdots, c^i_{\delta_i})$. That is, a passenger takes the footpath f^0 from $f^0_{dep\_stop}$ to $f^0_{arr\_stop}$, then takes the connection c^1_0, c^1_1, $\cdots$, $c^1_{\delta_1}$, proceeds to take the footpath f^1 from $f^1_{dep\_stop}$ to $f^1_{arr\_stop}$ *etc*. until he reaches $f^r_{arr\_stop}$. A journey must start and end with a footpath, which can be a self loop. In this chapter, we sometimes denote a journey as a sequence of footpaths and connection, i.e, $J = (f^0, c^0, c^1, \cdots, c^\alpha, f^1, c^{\alpha+1}, \cdots, f^{r-1}, c^{\gamma+1}, \cdots c^\phi, f^r)$ where $c^0 = c^0_0, c^1 = c^0_1, \cdots, c^\phi = c^r_{\delta_r}$.

Given two stops o and d in $\mathcal{S}$, an o-d journey J is a journey $(f^0, c^0, \cdots, c^\phi, f^r)$ such that f^0 starts from o and f^r ends at d. We define the departure time of a journey $dep_t(J)$ as the departure time of its first footpath, formally, $dep_t(J) = c^0_{dep\_time} - f^0_{dur}$. Similarly, the arrival time of a journey $arr_t(J)$ is the arrival time of its last footpath, i.e, $arr_t(J) = c^\phi_{arr\_time} + f^r_{dur}$.

A journey is called simple if it does not visit twice the same stop (except for self loop footpaths). Formally, let $J = (f^0, l^0 = (c^0_0, \cdots, c^0_{\delta_0}), \cdots, f^i, l^i = (c^i_0, \cdots, c^i_{\delta_i}), \cdots, f^j, l^j = (c^j_0, \cdots, c^j_{\delta_j}), \cdots, l^r = (c^r_0, \cdots, c^r_{\delta_r}), f^r)$ be a journey. For all $0 \leq i < j \leq r$, let c_{dep_stop} be the departure stop of c^i_α for $0 \leq \alpha \leq \delta_i$. Similarly, for $0 \leq \beta \leq \delta_j$, let c'_{dep_stop} be the departure stop of c^j_β and c'_{arr_stop} be the arrival stop of c^j_β. We have $c_{dep\_stop} \neq c'_{dep\_stop}$ and $c_{dep\_stop} \neq c'_{arr\_stop}$.*

The concatenation of two journeys $J = (f^0, l^0, \cdots, l^r, f^r)$ and $J' = (f'^0 = f^r, l'^0, \cdots, l'^\ell, f'^\ell)$ such that $f^r = f'^0$ and $arr_t(J) \leq dep_t(J')$ is the journey starting by f^0, follows J until f^r, then it follows J' until f'^ℓ. Formally, $J'' = (f^0, l^0, \cdots, f^r = f'^0, \cdots, = l'^\ell, f'^\ell)$ (we denote $J'' = J.J'$).

Given a journey $J = (f^0, c^0, \cdots c^i, \cdots, c^\phi, f^r)$, a journey $Q = (f'^0, c'^0, \cdots, c'^i, \cdots, c'^w, f'^\ell)$ is called a *detour* of J at i if $f'^0 = f^0, c'^0 = c^0, \cdots, c'^{i-1} = c^{i-1}$ but $c'^i \neq c^i$ and $f'^\ell_{arr\_stop} = f^r_{arr\_stop}$. Moreover, if Q is simple, it is called a *simple detour* of J at i. Similarly, Q is called an earliest arrival (simple) detour of J at i, if $arr_t(Q) \leq arr_t(Q')$ for each (simple) detour Q' of J at i.

Two journeys are equal if and only if all of their attributes are the same.

We denote by $\mathcal{J}^{t_0,t_{max}}_{o,d}$ the set of o-d simple journeys starting from o after t_0 and reaching d before t_{max}, i.e, $\mathcal{J}^{t_0,t_{max}}_{o,d} = \{J$ s.t. J is a simple o-d journey with $dep_t(J) \geq t_0$ and $arr_t(J) \leq t_{max}\}$.

4.2.2 Connection Scan Algorithm

As already said in the Introduction of this thesis (Section 1.6.3 page 31), the Connection Scan Algorithm (CSA) answers earliest arrival time journey queries from a given origin o to a given destination d. That is, departing after a given time t_0, how to get from o to d as soon as possible.

Similarly to Dijkstra's algorithm, the CSA will store an earliest arrival time for each stop in an array. A connection is considered *reachable* if a passenger can sit in the public transit vehicle of the connection. However, the main difference between Dijkstra's algorithm and the CSA is the fact that the CSA does not use a priority queue. Instead, the CSA iterates over all the connections sorted by their departure time (the same ordering is used for all queries). The CSA checks whether a connection is reachable or not. If so, it improves the arrival time at the arrival stop of the connection. Once all the connections have been scanned, the earliest arrival time to a stop is the current arrival time stored for the stop. The main advantage of avoiding the use of a priority queue is that, while more connections are scanned, the amount of work per connections is significantly reduced. Therefore, the CSA is significantly faster than Dijkstra's algorithm in this context [46].

4.2.3 Profile Connection Scan Algorithm

The result of the Profile Connection Scan Algorithm (PCSA) is a mapping between a departure time from a departure stop onto the earliest arrival time at the arrival stop. In other words, the profile problem solves simultaneously the earliest arrival problem for all departure times.

Compared with the CSA, the PCSA iterates on the connections sorted decreasingly by departure time, which leads to the fact that it solves the all-to-one problem. The PCSA constructs journeys from late to early and exploits the fact that an early journey can only have later journeys

∗. We suppose that a leg cannot have a loop, as a user may get off and wait outside the corresponding vehicle.

as subjourneys. It has been reported in [46] that the PCSA is one order of magnitude slower than the CSA, which is acceptable considering the fact that it solves the all-to-one problem.

Note that, the PCSA offers, from each stop s to the arrival stop d, a single earliest arrival s-d journey departing after t_0 and reaching d before t_{max}.

Let M be the output of the PCSA, we denote by $M_{o,d}^{t_0,t_{max}}$ an earliest arrival journey starting from o and reaching d, departing after t_0 and arriving before t_{max}.

4.3 Problem definition

In this section, we formalize the k Earliest Arrival Time problem definition.

k Earliest Arrival Time ($kEAT$) problem In this chapter, we aim at finding k earliest arrival time ($kEAT$) simple journeys from a given origin to a given destination. Formally, the problem takes as input a timetable $\mathcal{T} = (S, T, C, F)$, origin and destination stops o, d in S, a departure time t_0, a maximum arrival time t_{max} (often $t_{max} = t_0 + 24h$ or $t_{max} = t_0 + 48h$) and an integer k. It asks to find a set $\mathcal{J}^* = \{J_1, J_2, \cdots, J_k\}$ of top-k earliest arrival o-d simple journeys i.e, $J_i \neq J_j$ for $0 \leq i < j \leq k$, and for every J in $\mathcal{J}^*$, $J' \in \mathcal{J}_{o,d}^{t_0,t_{max}} \setminus \mathcal{J}^*$, $arr_t(J) \leq arr_t(J')$.

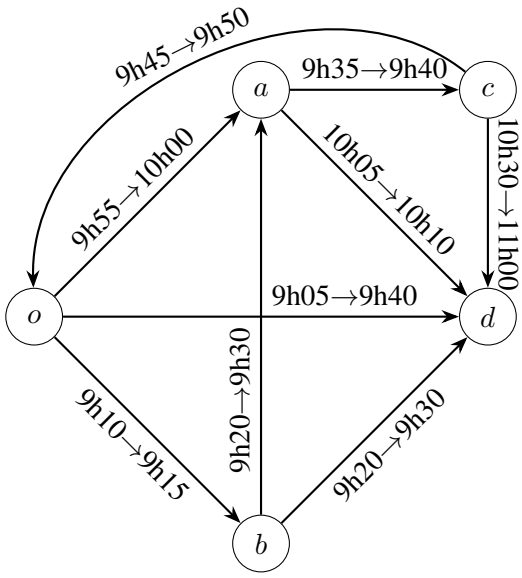


Figure 4.1 – Toy network for k earliest arrival time journeys

Example In the example of Figure 4.1, we look for the four earliest arrival time journeys from o to d departing after 9h00 :

The earliest arrival journey arrives at d at 9h30, starts with o and reaches d via b, the passenger arrives at b at 9h15 and waits 5 minutes before boarding the connection going from b to d, $J_0 = (o, d, b)$. The second journey arrives at 9h40 and goes directly from o to d, $J_1 = (o, d)$. The third journey arrives at 10h10 and goes from o to b then a then d, the passenger arrives at b at 9h15, waits 10 minutes then boards the connection going from b to a, arrives at 9h30 and waits 35 minutes

before boarding the connection going from a to d, $J_2 = (o, b, a, d)$. The fourth journey arrives at 10h10 and goes from o to a then d, $J_3 = (o, a, d)$.

Note that the journey $J_{ns} = (o, b, a, c, o, a, d)$ arriving at 10h10 is not a part of the solution as it is not simple (it visits the station o twice).

Indeed, there are other o-d journeys in this example, however they all have an arrival time greater than 10h10, that is $\{J_0, J_1, J_2, J_3\}$ are the four earliest arrival simple o-d journeys.

Each edge in the graph belongs to a specific trip, meaning that between each step in the examples there is a self loop footpath.

4.4 Public Transit Yen's algorithm (Y-PT)

In this section, we describe our adaption of Yen's algorithm on public transit networks, called Y-PT algorithm. As described before, Y-PT algorithm solves the $kEAT$ problem. So, it takes as input a timetable $\mathcal{T} = (S, T, C, F)$, origin and destination stops o, d in S, a departure time t_0, a maximum arrival time t_{max} ($= t_0 + 48h$) and an integer k, and returns a set *Output* $= \{J_1, J_2, \cdots, J_k\}$ of top-k earliest arrival o-d simple journeys in $\mathcal{T}$.

Roughly, Y-PT algorithm starts by computing a first earliest arrival journey (using the CSA algorithm), iterates over its connections in order to compute its earliest arrival simple detours and adds their minimum (the detour with minimum arrival time) to the output. Then, Y-PT algorithm repeats this process until k journeys are added to the output.

Now, let us give a precise and formal description of Y-PT algorithm whose pseudocode is presented in Algorithm 4.1. Analogously to Yen's algorithm, Y-PT starts by computing an earliest arrival journey J_0 and adding it (with 0 as deviation index) to a set of candidate journeys called $Candidates$. The journeys of the set $Candidates$ are non-decreasingly sorted by their arrival time. Also, the algorithm initializes the output set $Output$ as an empty set. After this initialization phase, the algorithm extracts a minimum element from the set $Candidates$, i.e, a journey J with minimum arrival time among those in $Candidates$ and adds it to $Output$. Let $C_J = (c^0, c^1, \cdots, c^\phi)$ be the sequence of connections of J,i.e, J has the form $J = (f^0, c^0, \cdots, c^\phi, f^r)$. The algorithm iterates over the connections in C_J starting from the deviation index of J. Precisely, let j be the deviation index of J, for each connection $c^i = (c^i_{dep\_stop}, c^i_{arr\_stop}, c^i_{dep\_time}, c^i_{arr\_time}, c^i_{trip})$ for $j \leq i \leq \phi$, the algorithm removes the prefix stations, i.e, each station visited by one of the connections $c^0, \cdots, c^{i-1}$, (equivalent to the prefix path of Yen's) from $\mathcal{T}$. This is done to ensure that the candidate journey is simple.

Moreover, in order to avoid duplications of journeys, for each journey J in $Output$ starting with the connections $c^0, c^1, \cdots, c^{i-1}, c'$, the connection c' is removed from $\mathcal{T}$. Then, using the CSA, the Y-PT algorithm computes an earliest arrival journey $Q = (f^0_Q, c^0_Q, \cdots, c^\omega_Q, f^\ell_Q)$ from $c^{i-1}_{arr\_stop}$ to d with $c^{i-1}_{arr\_time}$ as departure time[†]. Let J_{new} be the concatenation of the prefix of J and Q, i.e, $J_{new} = (f^0, c^0, \cdots, c^{i-1}, f^0_Q, c^0_Q, \cdots, c^\omega_Q, f^\ell_Q)$. The journey J_{new} is added to $Candidates$ with i as deviation index.

Y-PT algorithm repeats this process until k journeys are added to $Output$.

†. If the element right before c^i is a footpath, i.e, $J = (f^0, \cdots, f^\lambda, c^i, \cdots, f^r)$. It is possible to have journeys with two consecutive footpaths. In order to avoid such scenario, the CSA call is forced to compute a journey starting with a self loop footpath. This won't hurt the correctness of the algorithm, as footpaths are transitively closed

Input A timetable $\mathcal{T} = (S, T, C, F)$, an origin and a destination stops (o and d), departure and maximum arrival time t_{dep}, t_{max} and an integer k
Output a set of top-k earliest arrival journeys from o to d departing after t_{dep}
$J_0 \leftarrow CSA(\mathcal{T}, o, d, t_{dep}, t_{max})$
$Candidates \leftarrow \{(J_0, 0)\}$
$Output \leftarrow \emptyset$
while $\mid Output \mid < k$ and $Candidate \neq \emptyset$ **do**
    $\varepsilon = (J, j) \leftarrow extractmin(Candidates)$
    let $C_J = (c^0, \cdots, c^\phi)$ be the sequence of connections of J
    add J to $Output$
    for each connection c^i with $j \leq i \leq \phi$ in C_J **do**
        $c_{arr_stop} \leftarrow$the arrival stop of c^{i-1}
        $c_{arr_time} \leftarrow$the arrival time of c^{i-1}
        $\pi = (f^0, c^0, \cdots, c^{i-1})$
        $S_\pi \leftarrow$ the set of stations visited by one of the connections $(c^0, \cdots, c^{i-1})$
        $C_{dev} \leftarrow \{c'$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c')\}$
        $\mathcal{T}' = (S \setminus S_\pi, T, C \setminus C_{dev}, F)$
        $Q \leftarrow CSA(\mathcal{T}', c_{arr\_stop}, d, c_{arr\_time}, t_{max})$
        $J_{new} \leftarrow \pi.Q$
        add (J_{new}, i) to $Candidates$
Return $Output$

Algorithm 4.1 – Public Transit - Yen's algorithm (PT-Y)

4.5 Public Transit Postponed Yen's algorithm (PY-PT)

Here we explain Postponed Yen algorithm for public transit (PY-PT algorithm) whose pseudo-code is presented in Algorithm 4.2. This algorithm is inspired from the Postponed Yen algorithm (PY) for the $kSSP$ described in Chapter 2 (Section 2.5 page 49).

PY-PT algorithm has the same input as Y-PT algorithm, and it also returns a set of top-k earliest arrival simple journeys from the origin to the destination in a timetable. However, the journeys given by Y-PT are not necessarily the same as those given by PY-PT, i.e, the order of extraction of journeys is not necessarily the same. This may occur in scenarios where several journeys from the origin to the destination have the same arrival time.

The main drawback of Y-PT algorithm is its excessive number of calls of the CSA. Here, with the help of lower bounds on the arrival time of simple detours, we propose to postpone these calls in order to avoid some of them. We show that this can be done while preserving the correctness of the algorithm. In contrast with Y-PT algorithm where all journeys in the set $Candidates$ are simple, the PY-PT algorithm may add non-simple journeys to the set $Candidates$. As shown below, this corresponds to detours whose effective computation (and so their corresponding CSA calls) are postponed.

Let us now describe PY-PT algorithm in details.

For a query from the origin o to the destination d starting at time t_0, the PY-PT algorithm first uses the Profile CSA (PCSA). Let M be the mapping output by PCSA. The mapping M associates to each station $s \in S$ and each departure time $t \geq t_0$ the earliest arrival s-d journey, providing

it is possible to reach d from s before t_{max} when starting at t (we let $t_{max} = t_0 + 48h$ in our experiments).

Similarly to Y-PT algorithm, PY-PT algorithm starts by adding an earliest arrival time journey J_0 to a set of candidate journeys called $Candidates$. An element ε in $Candidates$ has three attributes, the journey J, its deviation index i and a boolean flag ζ indicating whether J is simple or not. So, the element $\varepsilon_0 = (J_0, 0, 1)$ is added to $Candidates$. In contrast with Y-PT algorithm where a CSA call is consumed to compute J_0, PY-PT algorithm extract J_0 from the already computed mapping M. Precisely, $J_0 = M_{o,d}^{t_0,t_{max}}$. Then, also like Y-PT algorithm, the $Output$ set is initialized with an empty set. After these initializations steps, the algorithms starts by extracting an earliest arrival journey (J, j, ζ) among those in $Candidates$. Suppose $J = (f^0, c^0, \cdots, c^\alpha, f^1, c^{\alpha+1}, \cdots, c^\beta, f^2, \cdots, c^{\gamma+1}, \cdots, c^\phi, f^r)$. Two cases are distinguished :

— **if** $\zeta = 1$ **(**J **is simple)** : J is added to the $Output$, then all the earliest arrival detours of J are added to $Candidates$. This is done as follows, let $C_J = (c^0, c^1, \cdots, c^\phi)$ be the sequence of connections of J, at each connection c^i (for $j \leq i < \phi$) in C_J, an earliest arrival detour J_{new} of J at i is extracted. This is done with the help of M as described below.

The journey J_{new} may not be simple (also described below). However, J_{new} will be added to the set $Candidate$ with i as deviation index and $\zeta = 1$ if Q is simple (and $\zeta = 0$ otherwise).

— **if** $\zeta = 0$ **(**J **is not simple)** : Then J is "repaired", i.e., it is replaced (if possible) by its corresponding earliest arrival simple journey. For this purpose, the algorithm applies almost the same routine as Y-PT algorithm. Precisely, let $c^j = (c^j_{dep\_stop}, c^j_{arr\_stop}, c^j_{dep\_time}, c^j_{arr\_time}, c^j_{trip})$ be the connection at the deviation index, the algorithm removes the prefix stations, i.e, each station visited by one of the connections $c^0, \cdots, c^{j-1}$, from $\mathcal{T}$. Also, for each journey J' in $Output$ starting with the connections $c^0, c^1, \cdots, c^{j-1}, c'$, the connection c' is removed from $\mathcal{T}$. Then, using the CSA, PY-PT algorithm computes an earliest arrival journey $Q = (f^0_Q, c^0_Q, \cdots, f^{\ell-1}_Q, l^\phi_Q, f^\ell_Q)$ from $c^{j-1}_{arr\_stop}$ to d with $c^{j-1}_{arr\_time}$ as departure time. Let J_{new} be the concatenation of the prefix of J and Q, i.e, $J_{new} = (f^0, c^0, \cdots, c^{j-1}, f^0_Q, c^0_Q, \cdots, f^\ell_Q)$. The journey J_{new} is added to the $Candidates$ with j as deviation index and with $\zeta = 1$ (as J_{new} is simple).

The PY-PT algorithm repeats this process until k journeys are added to $Output$.

Now, let us explain how the journey J_{new} is computed (in the case where $\zeta = 1$). The pseudocode of this procedure is described in Algorithm 4.3. Let $c^i = (c^i_{dep\_stop}, c^i_{arr\_stop}, c^i_{dep\_time}, c^i_{arr\_time}, c^i_{trip})$ be the i^{th} connection of C_J (for $j \leq i < \phi$), the following procedure is applied :

— First, the algorithm scans the connections starting with $c^i_{arr\_stop}$ after $c^i_{arr\_time}$ leading to new journeys, i.e, different from those in $Output$. Precisely, let $C_{dev} = \{c_{old} \in C$ s.t. there is a journey in $Output$ starting with the connections $c^0, \cdots, c^{i-1}, c_{old}\}$, let $C^N = \{c \in C$ s.t. $c_{dep\_stop} = c^i_{dep\_stop}$, $c_{dep\_time} \geq c^i_{dep\_time}$ and $c \notin C_{dev}\}$ be the set of new deviating connections. The algorithm scans the connections of C^N. Let c^{LB} be a connection of C^N leading to an earliest arrival journey from $c^i_{dep\_stop}$ to d using M. Formally, for each c in

Input A timetable $\mathcal{T}$, an origin and a destination stops (o and d), departure and maximum arrival time (t_{dep} and t_{max}), and an integer k
Output a set of top-k earliest arrival simple journeys from o to d departing after t_{dep}
$M \leftarrow PCSA(\mathcal{T}, o, d, t_{dep}, t_{max})$
$J_0 \leftarrow M_{o,d}^{t_{dep},t_{max}}$
$Candidate \leftarrow \{(J_0, 0, \zeta = 1)\}$
$Output \leftarrow \emptyset$
while $Candidate \neq \emptyset$ and $|Output| < k$ **do**
    $\varepsilon = (J, j, \zeta) \leftarrow extractmin(Candidates)$
    Let $J = (f^0, c^0, \cdots, c^\phi, f^r)$
    let $C_J = (c^0, \cdots, c^\phi)$ be the sequence of connections of J
    if $\zeta = 1$ (J is simple) **then**
        add J to $Output$
        for each connection c^i in C_J **do**
            $J_{new} \leftarrow$ EarliestArrivalDetour(J, i, M)
            $\zeta' \leftarrow 0$
            if J_{new} is simple **then**
                $\zeta' \leftarrow 1$
            add (J_{new}, i, ζ') to $Candidate$
    else
        $S_\pi \leftarrow$ the set of stations visited by one of the connections $(c^0, \cdots, c^{j-1})$
        $C_{dev} \leftarrow \{c$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c)\}$
        $\mathcal{T}' = (S \setminus S_\pi, T, C \setminus C_{dev}, F)$
        $Q \leftarrow CSA(\mathcal{T}', c_{arr\_stop}, d, c_{arr\_time}, t_{max})$
        if Q exists **then**
            $J_{new} \leftarrow (f^0, c^0, \cdots, c^j, Q)$
            add $(J_{new}, j, \zeta = 1)$ to $Candidates$
return $Output$

Algorithm 4.2 – Public Transit - Postponed Yen's algorithm (PY-PT)

C^N, let J_c be the journey via c following M, i.e, let $J_c = c.M_{c_{arr\_stop},d}^{c_{arr\_time},t_{max}}$, then c^{LB} is a connection in C^N s.t. $arr_t(J_{c^{LB}}) \leq arr_t(J_c)$ for each c in C^N ‡.

— Second, the algorithm scans the footpaths starting with $c^i_{dep\_stop}$ leading to new journeys, i.e, different from those in $Output$. Again, let $F_{dev} = \{f_{old}$ s.t. there is a journey in $Output$ starting with the connections $c^0, \cdots, c^{i-1}$ followed by $f_{old}\}$, let $F^N = \{f \in F$ s.t. $f_{dep\_stop} = c^i_{dep\_stop}$ and $f \notin F_{dev}\}$ be the set of the new deviating footpaths and let f^{LB} be a footpath of F^N leading to an earliest arrival journey from $c^i_{dep\_stop}$ to d using M. Precisely, for each f in F^N, let J_f be the journey via f following M, i.e, $J_f = f.M_{farr\_stop,d}^{c^i_{dep\_time}+f_{dur},t_{max}}$, then f^{LB} is a footpath in F^N s.t. $arr_t(J_{f^{LB}}) \leq arr_t(J_f)$ for each f in F^N.

‡. Similarily to Y-PT, if the element right before c^i is a footpath, i.e, $J = (f^0, \cdots, f^\lambda, c^i, \cdots, f^r)$. It is possible to have journeys with two consecutive footpaths. In order to avoid such scenario, the footpaths starting with $c^i_{arr\_stop}$ will not be scanned.

Now let Q_{min} be the journey with minimum arrival time among $J_{c^{LB}}$ and J_{fLB} and let J_{min} be the journey formed by the concatenation of the prefix journey of J and Q_{min}, i.e, $J_{min} = (f^0, c^0, \cdots, c^{i-1},$
$Q_{min})$. Note that, J_{min} may not be simple as the sub-journey extracted from M may revisit a station of one of the prefix connections. For instance, a station that is visited by c^0 or $c^1, \cdots$, or c^{i-1} may be visited again by $J_{c^{LB}}$ (or by J_f^{LB}) §.

$c^i \leftarrow$ the i^{th} connection of J
$c_{arr\_stop} \leftarrow$ the arrival stop of c^{i-1}
$c_{arr\_time} \leftarrow$ the arrival time of c^{i-1}
$C_{dev} \leftarrow \{c'$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, c')\}$
$C^N = \{c' \in C$ s.t. $c'_{dep\_stop} = c_{arr\_stop}$, $c'_{dep\_time} \geq c_{arr\_time}$ and $c' \notin C_{dev}\}$
$c^{LB} \leftarrow$ a connection in C^N leading to a minimum arrival time from c_{arr_stop} to d after c_{arr_time} following M
$F_{dev} \leftarrow \{f$ s.t. there is a journey J' in $Output$ starting with $(c^0, \cdots, c^{i-1}, f)\}$
$F^N = \{f \in F$ s.t. $f_{dep\_stop} = c_{arr\_stop}$ and $f \notin F_{dev}\}$
$f^{LB} \leftarrow$ a footpath in F^N leading to a minimum arrival time from c_{arr_stop} to d following M
$J_{c^{LB}} \leftarrow c^{LB}.M^{c^{LB}_{arr\_time}, t_{max}}_{c^{LB}_{arr\_stop}, d}$
$J_{f^{LB}} \leftarrow f^{LB}.M^{c_{arr\_time}+f^{LB}_{dur}, t_{max}}_{f^{LB}_{arr\_stop}, d}$
$J_{min} \leftarrow$ the earliest arrival journey among $J_{c^{LB}}$ and $J_{f^{LB}}$
$\pi = (f^0, c^0, \cdots, c^{i-1})$
$J_{new} \leftarrow \pi.J_{min}$
return J_{new}

Algorithm 4.3 – EarliestArrivalDetour(J, i, M)

To conclude, in contrast with Y-PT algorithm where an earliest arrival simple detour is computed at each index of an extracted journey using the CSA, PY-PT algorithm consider an earliest arrival detour (not necessarily simple) given by the already computed PCSA at each index, and two cases are distinguished : If the earliest arrival detour is simple, then a CSA call is saved and a shortest simple detour is added to $Candidates$. If not, i.e, the earliest arrival detour is not simple, PY-PT algorithm inserts this non-simple detour to the set of $Candidates$ with a flag indicating that it is not simple. Recall that journeys in $Candidates$ are non-decreasingly stored by their arrival time. So, only when this non-simple detour is extracted from $Candidates$, its simple version will be computed using the CSA. In other words, the actual computation of such simple detour is "postponed". Such postponement may end up saving some CSA calls, typically when k earliest arrival journey are added to $Output$ while none-simple journeys, whose actual computation is postponed, are still in $Candidates$, i.e, their whole "repair" procedure is skipped.

§. When scanning the connections starting with $c^i_{arr\_stop}$ after $c^i_{arr\_time}$, the journey $M^{c^i_{arr\_time}, t_{max}}_{c^i_{arr\_stop}, d}$ can start either with a self loop footpath or a footpath. On the other hand, when scanning footpaths starting with $c^i_{dep\_stop}$ the journey $M^{c^i_{dep\_time}+f_{dur}, t_{max}}_{f_{arr\_stop}, d}$ cannot start with anything other than a self loop footpath, to do so the PCSA stores journeys in two separate data structures, one for journeys starting with a self loop footpath and one for the other journeys.

Note that, despite these postponements, the order of extraction of simple journeys from $Candidates$ remains valid. This is because a journey J in $Candidate$ is either inserted with its real arrival time (the case where J is simple) or with a lower bound on its arrival time (the case where J is non-simple, by Claim 4.5.1).

Claim 4.5.1. *Let $J = (f^0, c^0, f^1 \cdots, c^\phi, f^r)$ be an o-d journey with J_{ns} an earliest arrival detour of J at i and J_s with an earliest simple arrival detour of J at i (where $0 \leq i \leq \phi$). Then, $arr_t(J_{ns}) \leq arr_t(J_s)$*

Proof. The proof follows from the fact that an earliest arrival detour of J at i arrives earlier than any detour of J at i. In particular, it arrives earlier than any earliest arrival simple detour of J at i. □

4.5.1 Experimental settings

Here we describe the details of the implementation and the setting used in our experiments.

We have implemented Y-PT and PY-PT algorithms in Java and our code is publicly available at [16].

Note that in our implementations the parameter k is not part of the input, this enables the use of these methods as iterators, able to return a next earliest arrival itinerary as long as one exists. Despite the fact that some additional optimizations could be added to the implementation if k is a part of the input.

Networks setting We have evaluated the performances of our algorithms on two train networks (Germany and Switzerland) and three public transit networks (Paris, Berlin and Stockholm). The characteristics of these networks are presented in Table 4.1. This dataset is publicly available via a GTFS feed (`https://transitfeeds.com/`), we downloaded this dataset in October 2019.

The public transit networks are more dense than the train networks, i.e. the connections to stops ratio is smaller on train networks than public transit networks. This can be easily explained because the train networks can only use trains whereas the public transit networks can use buses, trains, ferries and many other means of transportation. Therefore, we will show the performances of our algorithms on those two types of networks.

In our experiments, we have randomly chosen 1000 queries (source-destination pairs of stops) for each public transit network, and we have run each algorithm for each of these pairs for k going from 2 to 100.

We have considered the execution time and the number of CSA calls. Note that the number of CSA calls is an indication of the running time which is independent of the implementation and the architecture of the machine.

All reported computations have been performed on computers equipped with an Intel(R) Core(TM) i7-1185G7 at 3.00GHz and 32 GB of RAM.

4.5.2 Experimental results

In this section, we describe and analyse our experimental results on public transit networks.

We have measured the average and the median of the algorithms' running time in the considered networks. The data (the running time and the number of CSA calls) in Tables 4.2 and 4.3

Network	Stops	Connections	Lines	Trips	Footpaths
Germany	74 398	3 601 420	3 599	168 024	599 284
Switzerland	29 844	2 599 675	5 645	248 826	27 202
Paris	44 534	3 209 401	1 864	150 963	502 291
Berlin	28 651	1 379 755	1 296	63 569	62 456
Stockholm	14 258	703 326	664	34 799	22 138

Table 4.1 – Characteristics of the PT networks : number of stops, connections, lines, trips and footpaths.

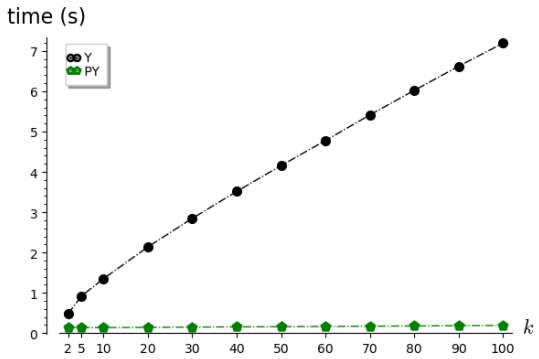
		Germany	Switzerland	Paris	Berlin	Stockholm
Y-PT	avg	94.6	42.0	66	22.7	7.2
	med	47.3	30.6	25.1	14	3.5
PY-PT	avg	3.6	1.9	5.4	0.8	0.2
	med	1.7	1.4	3.8	0.5	0.1

Table 4.2 – Running time (s) of the algorithms on PT networks, ($k = 100$)

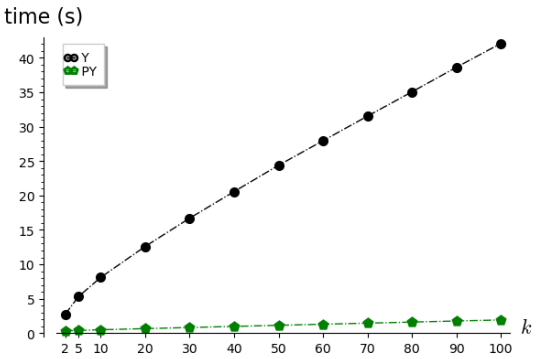
		Germany	Switzerland	Paris	Berlin	Stockholm
Y-PT	avg	2132	2158	1355	1788	2072
	med	1729	1749	1262	1604	1510
PY-PT	avg	32	77	39	7.6	8.3
	med	12	56	26	7	2

Table 4.3 – Number of CSA calls using each of the algorithms on PT networks, ($k = 100$)

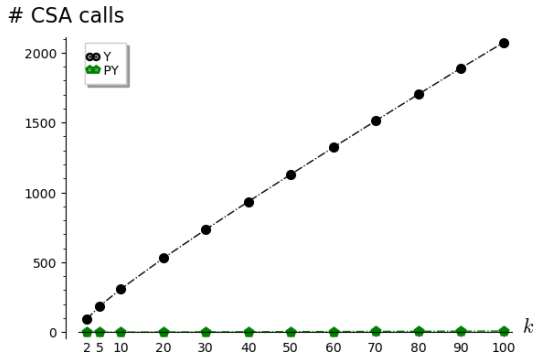
and Figure 4.3 corresponds to the biggest experienced value of k ($k = 100$). While the data in Figure 4.2 corresponds to their evolution with respect to the values of k.



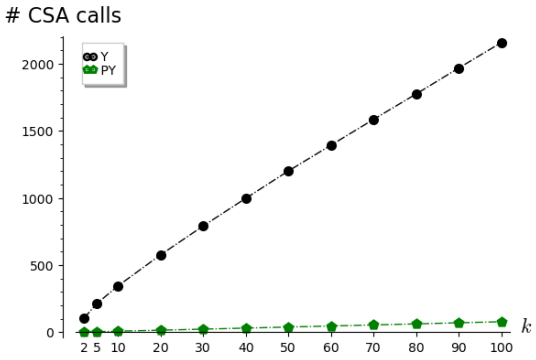
(a) Average running time on Stockholm



(b) Average running time on Switzerland



(c) Average number of CSA calls on Stockholm



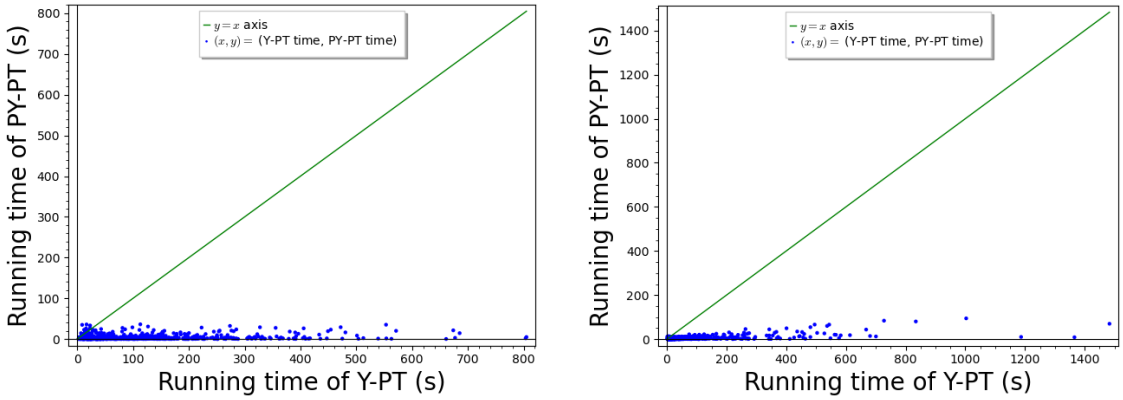
(d) Average number of CSA calls on Switzerland

Figure 4.2 – The running time of the $kEAT$ algorithms on Switzerland train network and Stockholm public transit network with respect to the values of k

The average and median running times reported in Table 4.2 show that the PY-PT algorithm is significantly faster than the Y-PT algorithm for every considered network (the average speed up of the running time is bigger than 10 for $k = 100$). Moreover, a refined comparison on Germany and Paris networks (Figure 4.3) show that PY-PT is faster than Y-PT for almost all queries. In addition, Figures 4.2a and 4.2b shows that this speed up remains important even for small values of k (even for $k = 2$) for Stockholm and Switzerland networks. This means that the time consumed for the PCSA computation routine is compensated by the extraction of simple detours, even for $k = 2$. In addition, very similar results were obtained on the remaining networks. Based on these remarks, we conclude that, in practice, PY-PT is faster than Y-PT for almost every scenario (the value of k, the query specifications and the network structure).

Furthermore, on Stockholm and Switzerland networks, Table 4.3 and Figures 4.2c and 4.2d show that the number of CSA calls is significantly reduced using PY-PT. This ensures that a similar speed up is guaranteed for any experimental settings [77].

As the obtained results are similar, we only displayed data obtained from experiments on selected networks (Stockolm and Switzerland for Figure 4.2, Paris and Germany for Figure 4.3). However, the results/plots corresponding to the remaining networks are very similar.



(a) Running time of Y-PT and PY-PT on Germany

(b) Running time of Y-PT and PY-PT on Paris

Figure 4.3 – Comparison of the running time of Y-PT and PY-PT on a train network and a public transit network

To conclude, on average, PY-PT algorithm is more than 10 times faster than Y-PT algorithm, it is also faster than Y-PT for almost every scenario.

4.6 Conclusion

In this chapter, we have shed lights on a new style of journey planning in public transit networks, offering a vast set of interesting solutions. This is done by adapting the k shortest simple paths problem to the public transit network context. We proposed a straightforward adaptation of Yen's algorithm and a more refined version answering the proposed problem in a reasonable running time.

Interesting questions are asked about designing algorithms answering k earliest arrival journeys query faster. Whether by improving / proposing faster methods than PY-PT algorithm, or even with the help of a preprocessing routine. For instance, a more specific question is whether one can use journey planning algorithms like Transfer Patterns algorithm [20] to answer k earliest arrival journeys queries ?

In addition, the approach proposed in this chapter does not guarantee any dissimilarity of the proposed journeys, i.e, a large part of output journeys may overlap in some scenarios. So, an interesting question is the study of finding journeys that are "dissimilar" in public transit networks, as studied for shortest dissimilar paths finding in a graph [33, 15].

Conclusions and Perspectives

In this chapter, we summarize the results obtained in this thesis. Then, we describe the research perspectives of this thesis.

5.1 Conclusions

This thesis tackles different variants and aspects of the k shortest paths problem. We first studied the original problem and showed how to improve / outperform the state-of-the-art algorithm either by improving the running time in practice or by proposing interesting space-time trade-offs and we studied the behavior of these algorithms on complex networks. Moreover, we initialized the study of this problem on arbitrarily weighted digraphs. In addition, we added dissimilarity constraints on the paths in order to increase their mutual diversity, where we proved that these constraints make the problem NP-COMPLETE for four of the most studied similarity measures. Also, we reviewed the state-of-the art algorithms allowing to find k shortest dissimilar paths and we reviewed an alternative pseudo-polynomial time algorithm allowing to solve the problem on arbitrarily weighted digraph with no negative cycles. Finally, we showed how to adapt the k shortest simple paths problem to public transit networks, where we proposed algorithms allowing to find a set of k earliest arrival time journeys from a departure stop to an arrival stop, departing after a given departure time.

Framework 5.1 summarizes and shows the link between the algorithms presented in this thesis. As described in Chapter 4, finding k earliest arrival time journeys from a departure to an arrival stop while departing after a given departure time can be seen as a straightforward adaptation of the k shortest simple paths to public transit networks. It is also shown in Chapter 4 that Postponed Yen - Public Transit (PY-PT) algorithm is the most efficient algorithm in practice, among the two considered algorithms, allowing to solve this version of the problem within an acceptable running time. Moreover, the k shortest problem on directed weighted digraph has different versions depending on the context or the use case. For instance, in a use case where dissimilarity between paths is desired, then MULTIPASS algorithm proposed by Chondrogiannis et al. [35] and PSEUDOPOLY algorithm proposed in Chapter 3 are recommended, depending on the sign of the arcs of the input digraph. However, to solve the original k shortest path problem, as defined in the literature on no negative weighted digraphs, we interrogate the structure of the digraph and we propose the framework established, experimentally, in Section 2.6.3 of Chapter 2. Precisely, If the digraph models a complex network, we recommend the PSB algorithm. In the case of road network, we propose either the postponed Yen (PY) or the Sidetrack Based (SB*) algorithm of Kurz and Mutzel with our improvement depending on the availability of the working memory. Finally, if the digraph has negative weighted arcs, then we recommend the Postponed Node Classification algorithm with Bellman Ford Moore (PNC-BFM) algorithm (proposed in Section 2.7 of Chapter 2) if the input digraph has no negative cycles. Otherwise, (for arbitrarily weighted digraphs), we recommend to

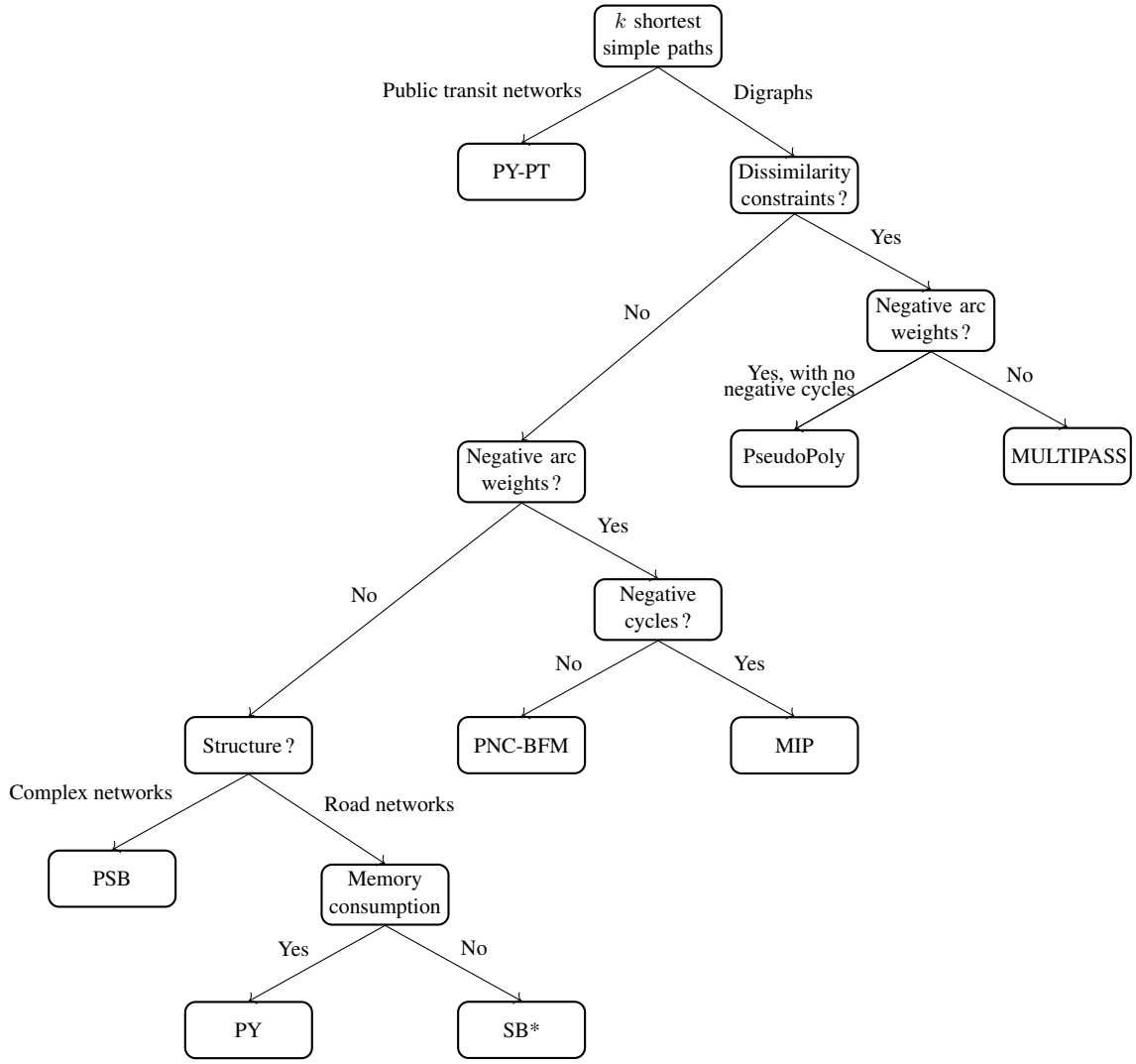


Figure 5.1 – A framework of the appropriate kSSP algorithm with respect to the use case

simply use one of the Mixed Integer linear Programming models proposed in Section 2.8 of Chapter 2. Note that all of these MIP models can be adapted to find dissimilar constraints with a single additive constraint to limit the intersection between the output paths.

5.2 Perspectives and questions

k shortest simple paths. A straightforward question, is whether one can improve one of the three algorithms (PSB, PY and SB*) in order to achieve better space-time tradeoffs in practice. This is probably achievable on complex networks, as studying their structures and properties may lead to kSSP algorithms tailored for complex networks and outperforming the PSB algorithm [*].

*. In fact, the PSB algorithm was designed in order to establish space-time tradeoffs on road networks. However, we discovered, experimentally, that it outperforms all the considered kSSP algorithms on complex networks.

Another interesting question is about the design of an algorithm with a preprocessing routine, enabling to quickly answer a kSSP query with the help of an already computed and stored data structure. In other words, achieving good trade-offs of preprocessing / query running time and working memory would be desired. Such trade-offs may be designed by adapting one of the classical one-to-one shortest path queries algorithm like CH and HL. Note that, such trade-offs are already achieved on k shortest paths by Akiba et al. [4] as described in the introduction of this thesis (see Section 1.4.2 page 21). Moreover, in all the considered algorithms, we did not suppose that the value of k is a part of the input. However, knowing in advance the number of requested paths may lead to various practical improvement. For instance, one may design an algorithm with different behavior with respect to the number of paths already added to the output and the number of the remaining paths to extract.

Even though no major theoretical improvements are expected regarding the complexity bound of the kSSP, there is room for possible improvements. Precisely, the algorithm with the best complexity bound, i.e, $O(n(m + n \log\log n))$ proposed in [64] (for $k = 2$) is larger than the lower bound of Williams and Williams [118], that is, $O(n \cdot m)$ on sparse digraphs. So, a natural question is about the possibility of getting rid of the $O(n \log n)$ and the design of a kSSP algorithm with $O(knm)$ time complexity.

Finally, after testing and evaluating all the kSSP algorithms, we noticed that most of the time is consumed in the process of finding a shortest path. Therefore, we try to ask a reversed question, that is, suppose we have access to a limited (k') number of shortest path queries (one-to-one or one-to-all), what is the number of shortest simple paths one can extract. A more direct question is the following : Can we find a second shortest simple path, using a single shortest path in-branching ?

Other similar problems to the kSSP were discussed. Most of these variants are related to the replacement path problem. The replacement path problem takes as input a digraph with an s-t path P, and asks to find a shortest s-t path in the digraph after removing, separately, each of the arcs of P. However, we are interested, in practical scenarios, by paths with short longest replacement path. i.e, finding a "short" s-t path P_L, such that, removing any arc of P_L won't severely increase the distance from s to t. Another question is a natural combination of the kSSP and the replacement path problem, let us call it the k replacement path problem. That is, with the same input as the replacement path, we ask to find, for each arc a in P, k shortest s-t paths avoiding a.

k shortest dissimilar paths. As most of the results presented in Chapter 2 were negative, i.e, most of the proposed variants were NP-COMPLETE. We are wondering if an interesting variant of the problem, or another meaningful similarity measure, can lead to a polynomial time algorithm for finding k shortest dissimilar paths. Another interesting theoretical question is about the problem of finding a non shortest path, i.e, an s-t path with length strictly higher than the distance from s to t. Surprisingly, it has been proven that this problem is NP-COMPLETE on non-negative weighted digraph. We are asking whether this problem is polynomial time solvable in positively weighted digraph (with no zero weighted arcs).

Also, practical interesting questions interrogate the evaluation of the similarity in practice, and whether better similarity measures could be designed with the help of geographical information, area's intersection or the type of the roads taken by the journey (highways, tunnels, *etc.*).

k earliest arrival time journeys. The main issue of the proposed algorithms of finding k earliest arrival time journeys, is that they do not ensure any dissimilarity between the journeys of the

output. Indeed, one can use these algorithms to find a super set of journeys and filter them in order to extract the journeys that are somehow dissimilar. However, this may be impractical in some scenarios. Therefore, a natural question is about the design of k earliest arrival time journeys with dissimilarity constraints with respect to a meaningful similarity measure in order to propose diverse journeys.

Another interesting question concerns the update of the objective function. Precisely, the adaptation of the k earliest arrival time journeys to answer k latest departure time journeys, k cheapest journeys, *etc.*.

References

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative routes in road networks. *Journal of Experimental Algorithmics (JEA)*, 18 :1–3, 2013.

[2] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 782–793. SIAM, 2010.

[3] Vedat Akgün, Erhan Erkut, and Rajan Batta. On finding dissimilar paths. *European Journal of Operational Research*, 121(2) :232–246, 2000.

[4] Takuya Akiba, Takanori Hayashi, Nozomi Nori, Yoichi Iwata, and Yuichi Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 2–8. AAAI Press, 2015.

[5] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360, 2013.

[6] Ali Al Zoobi, David Coudert, and Arthur Finkelstein. On finding k earliest arrival time journeys in public transit networks. 2021.

[7] Ali Al Zoobi, David Coudert, and Nicolas Nisse. On the k shortest simple paths : A faster algorithm with low memory consumption. *Journées Graphes et Algorithmes I3S et INRIA, Spohia Antipolis (en distanciel) 16–18 novembre 2020*, page 25.

[8] Ali Al Zoobi, David Coudert, and Nicolas Nisse. Compromis espace-temps pour le problème de k plus courts chemins simples. In *ALGOTEL 2020 – 22èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, page 4, Lyon, France, September 2020.

[9] Ali Al Zoobi, David Coudert, and Nicolas Nisse. On the top-k shortest paths with dissimilarity constraints. Technical report, 2020.

[10] Ali Al Zoobi, David Coudert, and Nicolas Nisse. Space and Time Trade-Off for the k Shortest Simple Paths Problem. In *18th International Symposium on Experimental Algorithms (SEA)*, volume 160 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 18 :1–18 :13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[11] Ali Al Zoobi, David Coudert, and Nicolas Nisse. De la difficulté de trouver des chemins dissimilaires. In *ALGOTEL 2021 - 23èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*, La Rochelle, France, September 2021.

[12] Ali Al Zoobi, David Coudert, and Nicolas Nisse. Finding the k Shortest Simple Paths : Time and Space trade-offs. Research report, Inria ; I3S, Université Côte d'Azur, April 2021.

[13] Ali Al Zoobi, David Coudert, and Nicolas Nisse. *k shortest simple paths (Version 2.0)*, 2021. `https://gitlab.inria.fr/dcoudert/k-shortest-simple-paths`.

[14] Ali Al Zoobi, David Coudert, and Nicolas Nisse. *k shortest simple paths with arbitrary arc weights*, 2021. `https://gitlab.inria.fr/aalzoobi/kssp-negative-weights`.

[15] Ali Al Zoobi, David Coudert, and Nicolas Nisse. On the complexity of finding k shortest dissimilar paths in a graph. Research report, Inria ; CNRS ; I3S ; UCA, 2021.

[16] Ali Al Zoobi and Arthur Finkelstein. *PT-KSSP Github repository*, 2021. `https://github.com/fink-arthur/PT-KSSP`.

[17] Masanori Arita. Metabolic reconstruction using shortest paths. *Simulation Practice and Theory*, 8(1-2) :109–125, 2000.

[18] Krishna Bala, Thomas E. Stern, and Kavita Bala. Algorithms for routing in a linear lightwave network. In *IEEE INFOCOM'91-Communications Societies Proceedings*, pages 1–9. IEEE Computer Society, 1991.

[19] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav Marathe, and Dorothea Wagner. Engineering label-constrained shortest-path algorithms. In *International conference on algorithmic applications in management*, pages 27–37. Springer, 2008.

[20] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Algorithms - ESA 2010, 18th Annual European Symposium. Proceedings, Part I*, pages 290–301, 2010.

[21] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route planning in transportation networks. In *Algorithm engineering*, pages 19–80. Springer, 2016.

[22] Hannah Bast and Sabine Storandt. Frequency-based search for public transit. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 13–22, 2014.

[23] Tolga Bektaş and Luis Gouveia. Requiem for the Miller-Tucker-Zemlin subtour elimination constraints ? *European Journal of Operational Research*, 236(3) :820 – 832, 2014. Vehicle Routing and Distribution Logistics.

[24] Martin Betz and Hermann Hild. Language models for a spelled letter recognizer. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 856–859. IEEE, 1995.

[25] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and Dong-uk Hwang. Complex networks : Structure and dynamics. *Physics reports*, 424(4-5) :175–308, 2006.

[26] Thomas H. Byers and Michael S. Waterman. Determining all optimal and near-optimal solutions when solving shortest path problems by dynamic programming. *Operations Research*, 32(6) :1381–1384, 1984.

[27] Giovanni Campuzano, Carlos Obreque, and Maichel M. Aguayo. Accelerating the Miller-Tucker-Zemlin model for the asymmetric traveling salesman problem. *Expert Systems with Applications*, 148 :1–9, 2020.

[28] 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.

[29] Horng-Jinh Chang and Uei-Tseng Lai. Empirical comparison between two k-shortest path methods for the generalized assignment problem. *Journal of Information and Optimization Sciences*, 19(2) :151–171, 1998.

[30] Eugene Charniak and Mark Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 173–180, 2005.

[31] Jung-Kuei Chen and Frank K. Soong. An n-best candidates-based discriminative training for speech recognition applications. *IEEE Transactions on Speech and Audio Processing*, 2(1) :206–216, 1994.

[32] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Alternative routing : k-shortest paths with limited overlap. In *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, (SIGSPATIAL)*, pages 68 :1–68 :4, Bellevue, WA, USA, November 2015. ACM.

[33] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, and Ulf Leser. Exact and approximate algorithms for finding k-shortest paths with limited overlap. In *Proceedings of the 20th International Conference on Extending Database Technology, (EDBT)*, pages 414–425, Venice, Italy, March 2017. OpenProceedings.org.

[34] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. Finding k-dissimilar paths with minimum collective length. In *Proceedings of the 26th ACM SIGSPATIAL*, pages 404–407, 2018.

[35] Theodoros. Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. Finding k-shortest paths with limited overlap. *The VLDB Journal*, 29(5) :1023–1047, 2020.

[36] Wu Chou, Tatsuo Matsuoka, Biing-Hwang Juang, and Chin-Hui Lee. An algorithm of high resolution and efficient multiple string hypothesization for continuous speech recognition using inter-word models. In *Proceedings of ICASSP'94. IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages II–153. IEEE, 1994.

[37] S. Clarke, A. Krikorian, and J. Rausen. Computing the n best loopless paths in a network. *Journal of the Society for Industrial and Applied Mathematics*, 11(4) :1096–1102, 1963.

[38] A. Claus. A new formulation for the travelling salesman problem. *SIAM Journal on Algebraic Discrete Methods*, 5(1) :21–25, 1984.

[39] John R. Current, Charles S. Revelle, and Jared L. Cohon. The median shortest path problem : A multiobjective approach to analyze cost vs. accessibility in the design of transportation networks. *Transportation Science*, 21(3) :188–197, 1987.

[40] George Bernard Dantzig, Delbert Ray Fulkerson, and Selmer Martin Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4) :393–410, 1954.

[41] Rafael Castro de Andrade. New formulations for the elementary shortest-path problem visiting a given set of nodes. *European Journal of Operational Research*, 254(3) :755–768, 2016.

[42] Rafael Castro de Andrade and Rommel Dias Saraiva. MTZ-primal-dual model, cutting-plane, and combinatorial branch-and-bound for shortest paths avoiding negative cycles. *Annals of Operations Research volume*, 286 :147–172, 2020.

[43] Daniel Delling, Thomas Pajor, and Renato F Werneck. Round-based public transit routing. *Transportation Science*, 49(3) :591–604, 2015.

[44] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. 9th DIMACS implementation challenge - shortest paths, 2006.

[45] Martin Desrochers and Gilbert Laporte. Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints. *Operations Research Letters*, 10(1) :27 – 36, 1991.

[46] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. Connection scan algorithm. *Journal of Experimental Algorithmics (JEA)*, 23 :1–56, 2018.

[47] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. *Journal of Experimental Algorithmics (JEA)*, 21 :1–49, 2016.

[48] Bajis Dodin. Determining the k most critical paths in pert networks. *Operations Research*, 32(4) :859–877, 1984.

[49] Michael Drexl. A note on the separation of subtour elimination constraints in elementary shortest path problems. *European Journal of Operational Research*, 229(3) :595 – 598, 2013.

[50] David Eppstein. Finding the k shortest paths. *SIAM Journal on computing*, 28(2) :652–673, 1998.

[51] David Eppstein. *Encyclopedia of Algorithms*, chapter k-Best Enumeration, pages 1003–1006. Springer New York, 2016.

[52] David Eppstein and Denis Kurz. K-best solutions of MSO problems on tree-decomposable graphs. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, volume 89, pages 16 :1–16 :13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[53] Erhan Erkut and Vedat Verter. Modeling of transport risk for hazardous materials. *Operations Research*, 46(5) :625–642, 1998.

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

[55] Matteo Fischetti, Juan José Salazar González, and Paolo Toth. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10(2) :133–148, 1998.

[56] Michael L. Fredman, Robert Sedgewick, Daniel D. Sleator, and Robert E. Tarjan. The pairing heap : A new form of self-adjusting heap. *Algorithmica*, 1(1) :111–129, 1986.

[57] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic algorithms for maintaining shortest paths trees. *Journal of Algorithms*, 34(2) :251–281, 2000.

[58] Liping Fu and Larry R. Rilett. Expected shortest paths in dynamic and stochastic traffic networks. *Transportation Research Part B : Methodological*, 32(7) :499–516, 1998.

[59] Jun Gao, Huida Qiu, Xiao Jiang, Tengjiao Wang, and Dongqing Yang. Fast top-k simple shortest paths discovery in graphs. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 509–518, 2010.

[60] Michael Randolph Garey and David Stifler Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., USA, 1990.

[61] Bezalel Gavish and Stephen C Graves. The travelling salesman problem and related problems. Technical Report OR-078-78, Massachusetts Institute of Technology, Operations Research Center, 1978.

[62] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. *Journal of Algorithms*, 53(1) :85–112, 2004.

[63] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path : A search meets graph theory. In *SODA*, volume 5, pages 156–165. Citeseer, 2005.

[64] Zvi Gotthilf and Moshe Lewenstein. Improved algorithms for the k simple shortest paths and the replacement paths problems. *Information Processing Letters*, 109(7) :352–355, 2009.

[65] Luis Gouveia and Jose Manuel Pires. The asymmetric travelling salesman problem and a reformulation of the Miller-Tucker-Zemlin constraints. *European Journal of Operational Research*, 112(1) :134–146, 1999.

[66] Longkun Guo and Hong Shen. On finding min-min disjoint paths. *Algorithmica*, 66(3) :641–653, 2013.

[67] Eleni Hadjiconstantinou and Nicos Christofides. An efficient implementation of an algorithm for finding k shortest simple paths. *Networks*, 34(2) :88–101, 1999.

[68] Yijie Han and Tadao Takaoka. An $O(n^3 \log\log n / \log^2 n)$ time algorithm for all pairs shortest paths. *Journal of Discrete Algorithms*, 38 :9–19, 2016.

[69] Nicolas Hanusse, David Ilcinkas, and Antonin Lentz. Framing algorithms for approximate multicriteria shortest paths. In *20th Symposium on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2020)*, 2020.

[70] John Hershberger, Matthew Maxel, and Subhash Suri. Finding the k shortest simple paths : A new algorithm and its implementation. *ACM Transactions on Algorithms*, 3(4) :45, 2007.

[71] G. J. Horne. Finding the k least cost paths in an acyclic activity network. *Journal of the Operational Research Society*, 31(5) :443–448, 1980.

[72] Stefan Irnich and Guy Desaulniers. Shortest path problems with resource constraints. In *Column generation*, pages 33–65. Springer, 2005.

[73] Adalat Jabrayilov and Petra Mutzel. *A new Integer Linear Program for the Steiner Tree Problem with Revenues, Budget and Hop Constraints*, pages 107–116. SIAM, 2019.

[74] Zhanfeng Jia and Pravin Varaiya. Heuristic methods for delay constrained least cost routing using/spl kappa/-shortest-paths. *IEEE Transactions on Automatic Control*, 51(4) :707–712, 2006.

[75] Víctor M. Jiménez and Andrés Marzal. A lazy version of eppstein's k shortest paths algorithm. In *International Workshop on Experimental and Efficient Algorithms*, pages 179–191. Springer, 2003.

[76] Wen Jin, Shuiping Chen, and Hai Jiang. Finding the k shortest paths in a time-schedule network with constraints on arcs. *Computers & operations research*, 40(12) :2975–2982, 2013.

[77] David S. Johnson. A theoretician's guide to the experimental analysis of algorithms. *Data structures, near neighbor searches, and methodology : fifth and sixth DIMACS implementation challenges*, 59 :215–250, 2002.

[78] Alireza Karduni, Amirhassan Kermanshah, and Sybil Derrible. A protocol to convert spatial polyline data to network formats and applications to world urban road networks. *Scientific data*, 3(1) :1–7, 2016.

[79] Naoki Katoh, Toshihide Ibaraki, and Hisashi Mine. An efficient algorithm for k shortest simple paths. *Networks*, 12(4) :411–427, 1982.

[80] Adrian Kosowski and Laurent Viennot. Beyond highway dimension : small distance labels using tree skeletons. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1462–1478. SIAM, 2017.

[81] Srijan Kumar, Bryan Hooi, Disha Makhija, Mohit Kumar, Christos Faloutsos, and VS Subrahmanian. Rev2 : Fraudulent user prediction in rating platforms. In *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, pages 333–341. ACM, 2018.

[82] Denis Kurz. *k-best enumeration - theory and application*. Theses, Technischen Universität Dortmund, March 2018.

[83] Denis Kurz and Petra Mutzel. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 64 of *LIPIcs*, pages 49 :1–49 :13. Schloss Dagstuhl, 2016.

[84] André Langevin, François Soumis, and Jacques Desrosiers. Classification of travelling salesman problem formulations. *Operations Research Letters*, 9(2) :127–132, March 1990.

[85] Mohsen Lashgari, Ata Allah Taleizadeh, and Abbas Ahmadi. Partial up-stream advanced payment and partial down-stream delayed payment in a three-level supply chain. *Annals of Operations Research*, 238(1-2) :329–354, 2016.

[86] Eugene L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7) :401–405, 1972.

[87] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1361–1370, 2010.

[88] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution : Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1) :2–42, 2007.

[89] Jure Leskovec and Andrej Krevl. SNAP Datasets : Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[90] Sanjiang Li and Yongming Li. Semi-dynamic shortest-path tree algorithms for directed graphs with arbitrary weights. *arXiv preprint arXiv :1903.01756*, 2019.

[91] Amgad Madkour, Walid G Aref, Faizan Ur Rehman, Mohamed Abdur Rahman, and Saleh Basalamah. A survey of shortest-path algorithms. *arXiv preprint arXiv :1705.02044*, 2017.

[92] Clair E. Miller, Albert William Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the Association for Computing Machinery*, 7(4) :326–329, October 1960.

[93] Edward F. Moore. The shortest path through a maze. In *International Symposium on the Theory of Switching*, page 285–292. Harvard University Press, 1959.

[94] Dalit Naor and Douglas L. Brutlag. On near-optimal alignments of biological sequences. *Journal of Computational Biology*, 1(4) :349–366, 1994.

[95] Paolo Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. New dynamic spt algorithm based on a ball-and-string model. *IEEE/ACM transactions on networking*, 9(6) :706–718, 2001.

[96] Mari Ostendorf, Ashvin Kannan, Steve Austin, Owen Kimball, Richard Schwartz, and J. Robin Rohlicek. Integration of diverse recognition methodologies through reevaluation of n-best sentence hypotheses. In *Speech and Natural Language : Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991*, 1991.

[97] Rose Oughtred, Chris Stark, Bobby-Joe Breitkreutz, Jennifer Rust, Lorrie Boucher, Christie Chang, Nadine Kolas, Lara O'Donnell, Genie Leung, Rochelle McAdam, et al. The biogrid interaction database : 2019 update. *Nucleic acids research*, 47(D1) :D529–D541, 2019.

[98] Thomas Pajor. Multi-modal route planning. Master's thesis, Universität Karlsruhe, 2009.

[99] Seth Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312(1) :47–74, 2004.

[100] Duc-Minh Phan and Laurent Viennot. Fast public transit routing with unrestricted walking through hub labeling. In *International Symposium on Experimental Algorithms*, pages 237–247. Springer, 2019.

[101] Liam Roditty and Uri Zwick. Replacement paths and k simple shortest paths in unweighted directed graphs. *ACM Transactions on Algorithms (TALG)*, 8(4) :1–11, 2012.

[102] Lukasz Salwinski, Christopher S. Miller, Adam J. Smith, Frank K. Pettit, James U. Bowie, and David Eisenberg. The database of interacting proteins : 2004 update. *Nucleic acids research*, 32(suppl_1) :D449–D451, 2004.

[103] Tadeusz Sawik. A note on the Miller-Tucker-Zemlin model for the asymmetric traveling salesman problem. *Bulletin of the Polish Academy of Sciences*, 2016.

[104] Grégoire Scano, Marie-José Huguet, and Sandra Ulrich Ngueveu. Adaptations of k-shortest path algorithms for transportation networks. In *2015 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 663–669. IEEE, 2015.

[105] Alexander Schrijver. *Combinatorial optimization : polyhedra and efficiency*. Springer, Berlin, 2003.

[106] Alexander Schrijver. On the history of the shortest path problem. *Documenta Mathematica*, 17(1) :155–167, 2012.

[107] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line : An empirical case study from public railroad transport. *ACM Journal of Experimental Algorithmics (JEA)*, 5 :12–es, 2000.

[108] Richard Schwartz and Steve Austin. A comparison of several approximate algorithms for finding multiple (n-best) sentence hypotheses. In *Acoustics, Speech, and Signal Processing, IEEE International Conference on*, pages 701–704. IEEE Computer Society, 1991.

[109] Robert Sedgewick. *Algorithms in Java, Part 5 : Graph Algorithms, 3rd Edition*. Addison-Wesley Professional, 2013.

[110] Tetsuo Shibuya and Hiroshi Imai. New flexible approaches for multiple sequence alignment. *Journal of Computational Biology*, 4(3) :385–413, 1997.

[111] Douglas R. Shier. On algorithms for finding the k shortest paths in a network. *Networks*, 9(3) :195–214, 1979.

[112] Yu-Keng Shih and Srinivasan Parthasarathy. A single source k-shortest paths algorithm to infer regulatory pathways in a gene network. *Bioinformatics*, 28(12) :i49–i58, 2012.

[113] Markus Sinnl and Ivana Ljubić. A node-based layered graph approach for the steiner tree problem with revenues, budget and hop-constraints. *Mathematical Programming Computation*, 8 :461–490, 2016.

[114] Frank K Soong and Eng-Fong Huang. A tree. trellis based fast search for finding the n best sentence hypotheses in continuous speech recognition. In *Speech and Natural Language : Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27, 1990*, 1990.

[115] Leonardo Taccari. Integer programming formulations for the elementary shortest path problem. *European Journal of Operational Research*, 252(1) :122–130, 2016.

[116] The Cooperative Association for Internet Data Analysis (CAIDA). The CAIDA AS relationships dataset. `http://www.caida.org/data/active/as-relationships/`, 2013.

[117] Donald M. Topkis. A k shortest path algorithm for adaptive routing in communications networks. *IEEE transactions on communications*, 36(7) :855–859, 1988.

[118] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 645–654. IEEE, 2010.

[119] Khoa D. Vo, Tran Vu Pham, Huynh Tuong Nguyen, Nghia Nguyen, and Tran Van Hoai. Finding alternative paths in city bus networks. In *2015 International Conference on Computer, Control, Informatics and its Applications (IC3INA)*, pages 34–39. IEEE, 2015.

[120] Dorothea Wagner and Tobias Zündorf. Public transit routing with unrestricted walking. In *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[121] Michael S. Waterman. Sequence alignments in the neighborhood of the optimum with general application to dynamic programming. *Proceedings of the National Academy of Sciences*, 80(10) :3123–3124, 1983.

[122] Mathias Weller. Optimal hub labeling is np-complete. *CoRR, abs/1407.8373*, 2014.

[123] Richard T. Wong. Integer programming formulations of the traveling salesman problem. In *IEEE International Conference on Circuits and Computers for Large Scale Systems*, pages 149–152, 1980.

[124] Feng Xie and David Levinson. Measuring the structure of road networks. *Geographical analysis*, 39(3) :336–356, 2007.

[125] Wangtu Xu, Shiwei He, Rui Song, and S. Choudhryb Chaudhry. Finding the k shortest paths in a schedule-based transit network. *Computers & Operations Research*, 39(8) :1812–1826, 2012.

[126] Horacio Hideki Yanasse, Nei Yoshihiro Soma, and Nelson Maculan. An algorithm for determining the k-best solutions of the one-dimensional knapsack problem. *Pesquisa Operacional*, 20 :117–134, 2000.

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

Calcul pratique de chemins simples avec des contraintes de longueur et de diversité dans des réseaux complexes et multimodaux

Ali AL ZOOBI

Résumé

Le problème du plus court chemin est l'un des problèmes les plus étudiés en théorie des graphes et en recherche opérationnelle. Une généralisation classique de ce problème est le problème de trouver k plus courts chemins simples (kSSP). C'est-à-dire, le problème de trouver le plus court, le deuxième plus court, etc. jusqu'au k-ième plus court chemin simple d'une source à une destination dans un graphe orienté pondéré. Yen (1971) a proposé l'algorithme avec la meilleure complexité théorique connue pour résoudre le kSSP dans un graphe orienté pondéré à n sommets et m arcs, avec une complexité en $O(kn(m + n \log n))$. Depuis, le problème a été largement étudié du point de vue de l'ingénierie algorithmique.

Dans cette thèse, nous étudions également le problème kSSP sous cet angle, c'est-à-dire que nous proposons des algorithmes exacts offrant de meilleures performances en pratique que l'état de l'art, en termes de temps d'exécution, de consommation mémoire ou offrant de meilleurs compromis espace-temps. Nous montrons aussi comment étendre nos algorithmes au cas des graphes avec des poids arbitraires sans cycles négatifs.

De plus, nous étudions le problème de trouver k plus courts chemins simples qui sont mutuellement dissimilaires. Plus précisément, nous étudions la complexité du problème en fonction de quatre mesures de similarité différentes, et nous montrons dans quels cas le problème est NP-Complet ou peut être résolu en temps polynomial.

Enfin, nous montrons comment adapter le problème kSSP à un modèle de réseau de transport public multimodal. Nous adaptons certains de nos algorithmes pour le kSSP au problème de trouver, dans un réseau de transport public multimodal, les k itinéraires d'une station source à une station destination arrivant au plus tôt.

Mots-clés : Théorie des graphes, plus court chemin, Ingénierie algorithmique.