# A fast algorithm for finding K shortest paths using generalized spur path reuse technique

Bi Yu Chen[a,b,c]*, Xiao-Wei Chen[a,b,c], Hui-Ping Chen[a,b,c] and William H. K. Lam[c]

[a]State Key Laboratory of Information Engineering in Surveying, Mapping and Remote Sensing, Wuhan University, Wuhan 430079, China; [b]Collaborative Innovation Center of Geospatial Technology, Wuhan, China; [c]Department of Civil and Environmental Engineering, The Hong Kong Polytechnic University

## Abstract

The problem of finding the K shortest paths (KSP) between a pair of nodes in a road network is an important network optimization problem with broad applications. Yen's algorithm (Yen, 1971) is a classical algorithm for exactly solving the KSP problem. However, it requires numerous shortest path searches, which can be computationally intensive for real large networks. This study proposes a fast algorithm by introducing a generalized spur path reuse technique. Using this technique, shortest paths calculated during the KSP finding process are stored. Accordingly, many shortest path searches can be avoided by reusing these stored paths. The results of computational experiments on several large-scale road networks show that the introduced generalized spur path reuse technique can avoid more than 98% of shortest path searches in the KSP finding process. The proposed algorithm speeds up Yen's algorithm by up to 98.7 times in experimental networks.

**Keywords:** K shortest path problem; generalized spur reuse technique; network; optimization.

## 1. Introduction

The problem of finding the K shortest paths (KSP) (i.e., the shortest path, the second shortest path and so on until the $k^{th}$ shortest path) between a pair of nodes in a road network is an important network optimization problem with broad applications in various fields (Chen et al., 2020; Liu et al., 2020; Sester, 2020). For example, KSPs are often found to solve complex network optimization problems with multiple constraints and/or objectives. A summary of KSP applications can be found in Eppstein (1998).

The KSP problem can be further classified into two variants. The first variant is to find the K shortest simple paths, in which repeated nodes are not allowed (Yen, 1971; Martins and Pascoal, 2003; Vanhove and Fack, 2012; Chen et al., 2020). The second variant is to find the K shortest non-simple paths, in which repeated nodes may exist (Eppstein, 1998; Martins et al., 1984; Minieka, 1974). Because simple paths are more complicated with an additional loopless constraint and are more applicable in real applications, this study focuses exclusively on the first variant. For convenience, KSPs are hereafter referred to as the K shortest simple paths.

In the literature, considerable effort has been devoted to developing effective algorithms for exactly finding KSPs. Among early KSP algorithms (Hoffman and Pavley, 1959; Clarke, 1963; Yen, 1971; Lawler, 1972), Yen's algorithm (1971) was recognized as having the best worst-case complexity (Vanhove and Fack, 2012; Chen et al., 2020). This algorithm builds on the deviation path concept, i.e., the $j^{th}$ shortest path deviates at a certain node (called the deviation node) from a previously determined $i^{th}$ $(i<j)$ shortest path. The link emanating from the deviation node on the $i^{th}$

---

* To cite this paper: Chen, B.Y., Chen, X.-W., Chen, H.-P. and Lam, W.H.K., 2020, A fast algorithm for finding K shortest paths using generalized spur path reuse technique. *Transactions in GIS*, DOI:10.1111/tgis.12699.

shortest path is called the deviation link. Based on the deviation node, the $j^{th}$ shortest path can be divided into two sub-paths. The first sub-path before the deviation node, called the root path, can be determined easily because it is identical to the corresponding sub-path of the $i^{th}$ shortest path. The second sub-path after the deviation node, called the spur path, requires a further shortest path search in a modified network where all nodes before the deviation node along the root path and the deviation link(s) have been removed from the original network. Using these deviation path concepts, Yen's algorithm essentially performs a series of shortest path searches in modified networks to calculate spur paths. Therefore, spur path calculation performance dominates the computational efficiency of Yen's algorithm.

The worst-case complexity of Yen's algorithm has been unrivalled for the past four decades. Rather than improving the algorithm's worst-case complexity, much research has been carried out to enhance its practical implementations, particularly to improve its spur path calculation performance. In the original implementation of Yen's algorithm (Yen, 1971), the one-to-all Dijkstra's algorithm (Dijkstra, 1959) was used to compute every spur path by constructing the entire one-to-all shortest path tree. However, this one-to-all shortest path tree construction is computationally intensive in large networks. To address this issue, an improved Yen's algorithm was suggested by researcher by using a one-to-one Dijkstra's algorithm (Hershberger et al., 2007). Martins and Pascoal (2003) proposed a path re-optimization technique. This technique speeds up spur path calculations by reusing the results of the one-to-all shortest path tree generated at the previous iteration. However, it is still necessary to update the entire one-to-all shortest path tree, leading to high computational overhead in large networks. It was reported that such algorithm (Martins and Pascoal, 2003) ran even slower than the improved Yen's algorithm in most cases (Vanhove and Fack, 2012). Along the line of re-optimization techniques, Chen et al. (2020) further proposed an efficient KSP algorithm, called KSP-LPA*, by explicitly formulating the spur path calculation process as subsequent iterations of one-to-one shortest path searches. They incorporated the lifelong planning A* technique to improve spur path calculation performance by reusing the results of one-to-one shortest path searches in the previous iteration. The proposed KSP-LPA* algorithm can speed up the improved Yen's algorithm by up to 2.5 times in test networks.

Unlike the above approaches based on improving spur path calculation performance, Vanhove and Fack (2012) introduced a new spur path reuse technique to reduce the number of spur path calculations involved in Yen's algorithm. For convenience, their proposed algorithm is hereafter called the V–F algorithm. The V–F algorithm first executed a backward one-to-all Dijkstra's algorithm to pre-calculate the shortest paths from all nodes to the destination. During the KSP finding process, the V–F algorithm tried to determine the spur path by reusing the pre-calculated shortest path from the deviation node to the destination if the pre-calculated path concatenated with the root path did not form a cycle. When the reuse failed (i.e., forming cycles), the V–F algorithm simply used the one-to-one Dijkstra's algorithm to calculate the spur path. Empirical studies have shown that the V–F algorithm can reuse 10–50% of spur paths and speed up the improved Yen's algorithm by about 10%–50% (Vanhove and Fack, 2012; Chen et al., 2020).

Along the line of previous work (Yen, 1971, Vanhove and Fack, 2012), this study proposes an efficient solution algorithm, called KSP-SPR, for exactly solving the KSP problem in road networks. This study contributes to the previous literature in the following aspects:

(1) A generalized spur path reuse technique is introduced to address the reuse failure issue in the V–F algorithm. The introduced technique calculates a set of candidate paths by incrementally removing nodes forming cycles. These calculated candidate paths are stored in a tree structure and efficiently retrieved for reuse in later iterations. Therefore, using this introduced technique can significantly enhance the percentage of reused spur paths and reduce computational time for spur path calculations.

(2) The A* technique (Fu et al., 2006) is incorporated into the candidate path calculations by using the results of pre-calculated shortest paths in the V–F algorithm as an effective heuristic function. Accordingly, candidate path calculations can be improved by giving higher priorities to nodes closer to the destination. A technique for avoiding label initialization (Chen et al., 2014) is also used to improve candidate path calculation performance.
(3) A comprehensive case study is carried out using five real road networks. For comparison, three state-of-the-art KSP algorithms are also implemamented, including the improved Yen's algorithm (Yen, 1971; Hershberger et al., 2007), the V–F algorithm (Vanhove and Fack, 2012), and the KSP-LPA* algorithm (Chen et al., 2020). The results of the case study show that the proposed algorithm performs consistently faster than all existing algorithms, e.g., it can speed up the improved Yen's algorithm by up to 98.7 times in test networks.

The remainder of this paper is organized as follows. Section 2 briefly describes the improved Yen's algorithm (Yen, 1971; Hershberger et al., 2007) and the V–F algorithm (Vanhove and Fack, 2012) to provide necessary background. Section 3 introduces the proposed KSP-SPR algorithm. Section 4 reports computational experiments using several real road networks. Finally, Section 5 presents conclusions together with future research recommendations.

## 2. Traditional K shortest simple path algorithms

### 2.1. Problem definition

Let $\mathrm{G(N, A)}$ be a directed graph with $|N|$ nodes and $|A|$ links. Each link $a(n_i, n_j) \in \mathrm{A}$, from the tail node $n_i$ to the head node $n_j$, has a nonnegative link cost, $t(n_i, n_j)$, typically representing link travel time or link length.

A simple path $p^u$ from the origin node $o \in N$ to the destination node $d \in N$ consists of $l$ subsequent nodes, $o = n_1^u, \ldots, n_l^u = d$, where $a(n_i^u, n_{i+1}^u) \in \mathrm{A}, n_i^u \neq n_{i+1}^u, i = 1, \ldots, l-1$. The path cost, denoted by $t(p^u)$, can be calculated as the summation of corresponding link costs along the path:

$$t(p^u) = \sum_{i=1}^{l-1} t(n_i^u, n_{i+1}^u) \tag{1}$$

Let $\mathrm{P}$ be the set of all simple paths between the same origin and destination (O-D) pair. The KSP problem can be defined as below.

**Definition 1**: (K shortest simple path problem) Given an integer $k \geq 1$, the KSP problem is to find the set of K shortest simple paths, denoted as $\mathrm{P}^k = \{p^1, \ldots, p^i, \ldots, p^k\}$, satisfying:
(1) $t(p^i) \leq t(p^{i+1})$ for $\forall p^i \in \mathrm{P^k}$;
(2) $t(p^k) \leq t(p^u), \forall p^u \in \mathrm{P} - \mathrm{P^k}$.

### 2.2. Improved Yen's algorithm

Yen's algorithm (1971) is one of the best-known algorithms for solving the KSP problem. It builds on the *deviation path* concept. Fig. 1 illustrates this concept using a simple example. Given the first shortest path $p^1 = \{n_1^1, \ldots, n_l^1\}$ consisting of $l-1$ links, there are $l-1$ deviation paths, $\{\bar{p}_1^1, \ldots, \bar{p}_i^1, \ldots, \bar{p}_{l-1}^1\}$, forming the deviation path set, denoted by $D^1$. The $i^{th}$ deviation path, $\bar{p}_i^1 \in D^1$ (e.g., $\bar{p}_2^1$ in Fig. 1) is defined as the shortest path between the O-D pair in the road network by excluding the $i^{th}$ link, $a(n_i^1, n_{i+1}^1)$, and is called the *deviation link* (e.g., $a(2,3)$). The tail node $n_i^1$ of the deviation link is called the *deviation node* (e.g. Node 2), where $\bar{p}_i^1$ deviates from $p^1$. Based on the deviation node, the deviation path can be divided into two sub-paths: the *root path* $\bar{r}_i^1$ and the *spur path* $\bar{s}_i^1$. The root path is the sub-path from the origin to the deviation node (e.g., $\bar{r}_2^1 = \{1,2\}$), and the spur path extends from the deviation node to the destination (e.g., $\bar{s}_2^1 = \{2,5,3\}$). The root path can be easily identified as the corresponding sub-path of $p^1$ according to the Bellman principle of optimization (Bellman, 1958), which states that a sub-path between any pair of nodes on the shortest path is the shortest path itself. The spur path, $\bar{s}_i^1$, however, should be further calculated due to the introduction of two types of constraints:

(1) $\bar{s}_i^1$ should not pass through the deviation link to guarantee that $\bar{p}_i^1$ is not identical to $p^1$;

(2) $\bar{s}_i^1$ should not pass through any node in $\bar{r}_i^j - \{n_i^j\}$ to ensure that $\bar{p}_i^1$ has no cycles.

The deviation path, $\bar{p}_i^1$, can be determined as $\bar{r}_i^1 \oplus \bar{s}_i^1$, where $\oplus$ is the path concatenation operator. After all deviation paths in $D^1$ have been calculated (e.g., $D^1 = \{\bar{p}_1^1, \bar{p}_2^1\}$), the second shortest path, $p^2$, is the deviation path with the minimum cost in $D^1$ (e.g., $\bar{p}_1^1$).

Analogously, the $(j+1)^{th}$ $(1 < j < K)$ shortest path, $p^{j+1}$, can be determined as a path among all deviation paths of the previously determined $j$ shortest paths, denoted by $\{D^1 \cup \ldots \cup D^j\} - \{p^2, \ldots, p^j\}$, where $D^j$ is the set of deviation paths of the $j^{th}$ shortest path. Unlike the scenario of $p^1$, given $p^j$ consisting of $l-1$ links, it is not necessary to calculate $l-1$ deviation paths. If $p^j$ coincides with a previous determined path $p^u$ $(u < j)$ at its first $m-1$ links, then the calculation of the first $m-1$ deviation paths, i.e., $\{\bar{p}_1^j, \ldots, \bar{p}_{m-1}^j\}$, can be omitted because they have been calculated and stored in $D^u$. Accordingly, the first deviation node of $p^j$ is the last node $n_m^j$ of the longest sub-path that completely coincides with a previously determined shortest path, and only $l-m$ deviation paths, $\{\bar{p}_m^j, \ldots, \bar{p}_{l-1}^j\}$, require further calculations. When calculating $\bar{p}_m^j$, the $m^{th}$ link of $p^u$ is also excluded as an additional deviation link to ensure that $\bar{p}_m^j$ is different from the previously determined deviation path, $\bar{p}_m^u \in D^u$. Note that multiple deviation links can be identified if the longest sub-path coincides with more than one previously determined shortest path. For example, $p^3$ in Fig. 1 coincides with $p^1$ at the longest sub-path, {2, 3}, and then the first deviation node is 2, and only two deviation paths $\{\bar{p}_2^3, \ldots, \bar{p}_3^3\}$ should be calculated. When calculating $\bar{p}_2^3$, both link $a(2,5)$ from $p^3$ and link $a(2,3)$ from $p^1$ are excluded as deviation links.

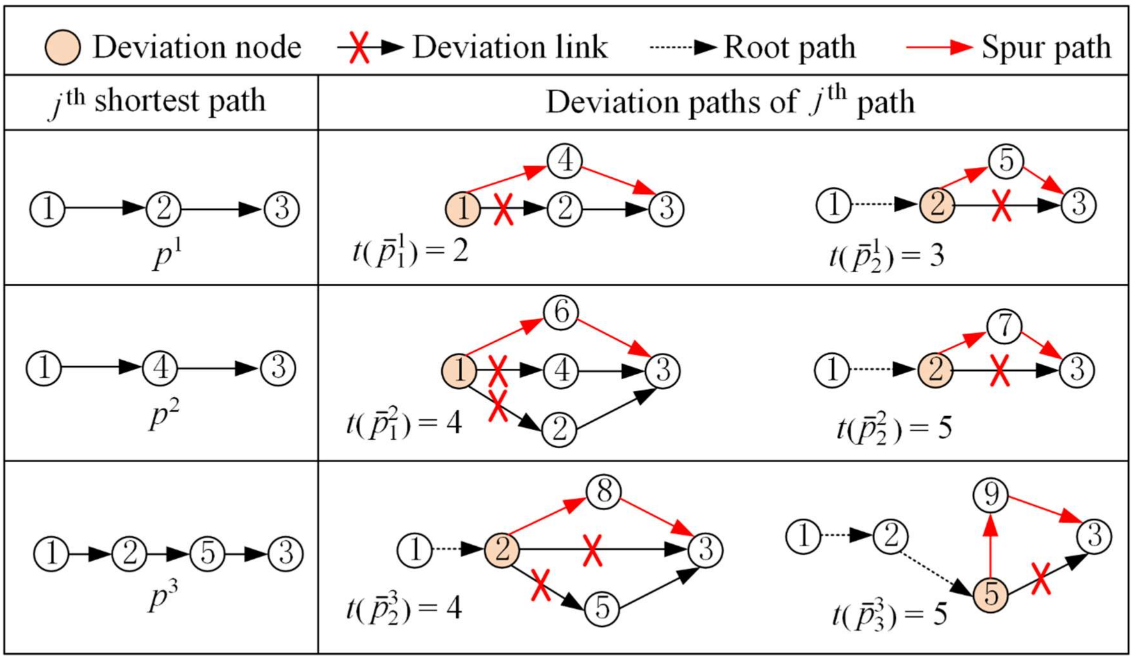


**Fig. 1.** Illustrative example of deviation path concept.

Using the above deviation path concepts, the steps of Yen's algorithm can be described as below. As shown in Table 1, Yen's algorithm maintains two sorted lists: the determined shortest path list, L, and the deviation path list, C. Initially, the first shortest path $p^1$ is calculated using the classical *forward one-to-one Dijkstra's algorithm* and added to L. In each iteration, the deviation path set, $D^j$, is calculated to determine the $(j+1)^{th}$ shortest path, $p^{j+1}$. First, the first deviation node $n_m^j$ and the deviation link set $\tilde{A}_m^j$ at $n_m^j$ are determined using the *FindFirstDevNode* procedure, which can be efficiently implemented by representing L as a deviation tree structure (Roddity and Zwick, 2005; Vanhove and Fack, 2012). Then all deviation paths $\{\bar{p}_m^j, \dots, \bar{p}_{l-1}^j\}$ in $D^j$ are calculated using the *CalculateDeviationPaths* procedure. To determine every deviation path $\bar{p}_v^j$, the spur path $\bar{s}_v^j$ is calculated using the *FindSpurPath-Yen* procedure. As shown in Table 2, the procedure simply uses the classical *forward one-to-one Dijkstra's algorithm* in a modified road network, in which all deviation links (denoted by $\tilde{A}_m^j$) and all nodes in $\bar{r}_i^j - \{n_i^j\}$ (denoted by $\tilde{N}_i^j$) are removed. Finally, all calculated deviation paths are added to C, and the $(j+1)^{th}$ shortest path, $p^{j+1}$, is determined as the path with the minimum path cost at the top of C. This step can be efficiently implemented using a priority queue structure such as a Fibonacci heap (Fredman and Tarjan, 1987). The algorithm terminates when all KSPs have been found or C is empty.

**Table 1.** Generic procedure of the Yen's algorithm

**Input:** O-D pair, *K*
**Return:** Determined path collection L
**01:** Call *forward one-to-one Dijkstra's algorithm* to calculate $p^1$. (Algorithms different here)
02: If $p^1 = \emptyset$, Then stop and return $\emptyset$.
03: Set determined path collection $\mathrm{L} := \{p^1\}$, and set candidate deviation path collection $\mathrm{C} := \emptyset$.
04: For $j$ :=1 to *K*-1
05: Call *FindFristDevNode*($p^j$, L) to determine the first deviation node $n_m^j$ and the deviation link set $\tilde{A}_m^j$ at $n_m^j$.
06: Call *CalculateDeviationPaths*($p^j$, $n_m^j$, $\tilde{A}_m^j$) to calculate deviation path set $D^j$.
07: Set $\mathrm{C} := \mathrm{C} \cup \mathrm{D}^j$.
08: If $\mathrm{C} = \emptyset$, Then stop and return L.
09: Set $p^{j+1}$ as the path with minimum path cost at the top of C; and remove $p^{j+1}$ from C.
10: Add $p^{j+1}$ into L.
11: End for
12: Return L.

**Procedure:** ***CalculateDeviationPaths***
**Input:** The $j^{th}$ shortest simple path $p^j$, the first deviation node $n_m^j$, the deviation links set $\tilde{A}_m^j$.
**Return:** Deviation path set $D^j$.
01: Set $\tilde{A}_i^j := \tilde{A}_m^j$.
02: For $i = m$ to $l-1$ ($l$ is the number of links of $p^j$)
03: if $i \neq m$, then Set $\tilde{A}_i^j := \{a(n_i^j, n_{i+1}^j)\}$.
04: Set root path $\bar{r}_i^j := (n_1^j, \dots, n_i^j)$, and Set removed node set $\tilde{N}_i^j := \bar{r}_i^j - \{n_i^j\}$.
**05:** Call *FindSpurPath-Yen*($\tilde{A}_i^j$, $\tilde{N}_i^j$) to calculate spur path $\bar{s}_i^j$. (Algorithms different here)
06: Determine deviation path $\bar{p}_i^j := \bar{r}_i^j \oplus \bar{s}_i^j$.
07: Add $\bar{p}_i^j$ into $D^j$.
08: End for

09: Return $D^j$.

**Table 2.** The procedure for calculating spur paths in the Yen's algorithm

**Procedure: *FindSpurPath-Yen***
**Input:** Deviation links set $\tilde{A}_i^j$, and removed node set $\tilde{N}_i^j$.
**Return:** spur path $\bar{s}_i^j$
01: Remove $\tilde{A}_i^j$ and $\tilde{N}_i^j$ from G.
02: Call *forward one-to-one Dijkstra's algorithm* to calculate spur path $\bar{s}_i^j$.
03: Restore $\tilde{A}_i^j$ and $\tilde{N}_i^j$ to G.
04: Return $\bar{s}_i^j$.

Yen's algorithm can achieve a worst-case complexity of $\mathrm{O}\{\mathrm{K}|N|(|A| + |N|\mathrm{Log}|N|)\}$ when Fibonacci heaps (Fredman and Tarjan 1987) are used in Dijkstra's algorithm. This complexity is well known as the best worst-case complexity for solving the KSP problem in the literature. However, this algorithm in practice could be computationally intensive in large networks when K is relatively large (e.g., K>100) due to the huge number of spur path calculations using *the forward one-to-one Dijkstra's algorithm* in the *FindSpurPath-Yen* procedure.

### 2.3. V–F algorithm

As an improvement, Vanhove and Fack (2012) developed an exact KSP algorithm, called the V–F algorithm, by introducing a spur path reuse technique. The V–F algorithm follows the same generic procedure of Yen's algorithm, but with two modifications. The first modification is on Line 01 of Table 1, by using *the backward one-to-all Dijkstra's algorithm* to calculate the first shortest path $p^1$. This step generates a complete shortest path tree, including not only $p^1$ from the origin, but also shortest paths from all other nodes to the destination $d$.

The second modification is to use the *FindSpurPath-V&F* procedure (see Table 3) instead of the *FindSpurPath-Yen* procedure. This modified procedure tries to reuse the results of $p^1$ search to determine spur path $\bar{s}_i^j$. Fig. 2 illustrates such a procedure using a simple example. Let $\mathrm{SUCC}(n_i^j)$ be the set of successor links emanating from deviation node $n_i^j$ (e.g., Node 4). Any successor link, $\forall a(n_i^j, n_v) \in \{\mathrm{SUCC}(n_i^j) - \tilde{A}_i^j\}$, can lead to a suitable detour $a(n_i^j, n_v) \oplus q_v^0$ by concatenating link $a(n_i^j, n_v)$ and the shortest path $q_v^0$ from successor node $n_v$ to the destination $d$. Note that $q_v^0$ has been calculated in the $p^1$ search. Therefore, all concatenated detour paths can be easily determined. Among them, the path with the minimum cost can be identified as a candidate path, $q_{iv}^0$. If the candidate path $q_{iv}^0$ does not contain any node in $\tilde{N}_i^j$ (i.e., $q_{iv}^0 \cap \tilde{N}_i^j = \emptyset$), then $q_{iv}^0$ can be reused successfully, i.e., $\bar{s}_i^j = q_{iv}^0$. Otherwise, $q_{iv}^0 \oplus \bar{r}_i^j$ has cycle(s), and the *FindSpurPath-Yen* procedure is simply used to calculate spur path $\bar{s}_i^j$.

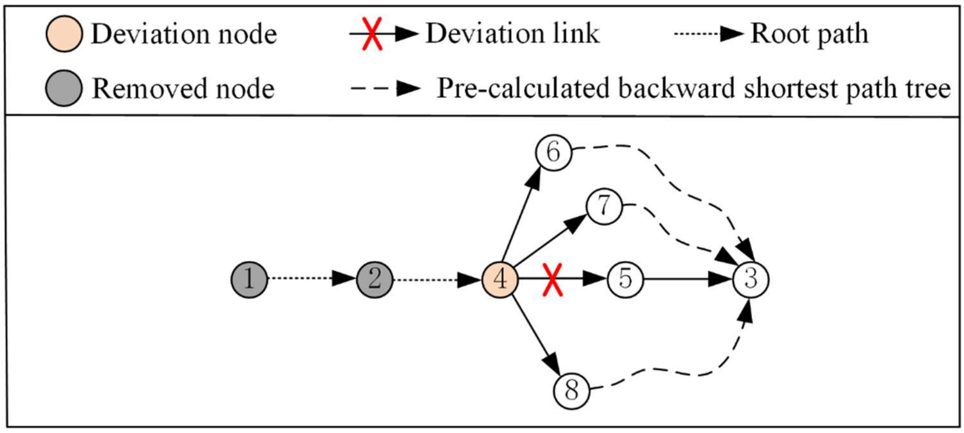


**Fig. 2.** Illustrative example of spur path reuse technique in the V-F algorithm.

**Table 3.** Spur path calculation procedure in the V-F algorithm

---

**Procedure: *FindSpurPath-V&F***
**Input:** Deviation links set $\tilde{A}_i^j$, and removed node set $\tilde{N}_i^j$.
**Return:** spur path $\bar{s}_i^j$
**Step 1.** Reuse of results in the generated shortest path tree:
01: Call *GetCandidatePath-V&F*($\tilde{A}_i^j, \tilde{N}_i^j$) to determine candidate path $q_{iv}^0$.
02: If $q_{iv}^0 \neq \emptyset$ and $q_{iv}^0 \cap \tilde{N}_i^j = \emptyset$, then Set $\bar{s}_i^j \coloneqq q_{iv}^0$ and Return $\bar{s}_i^j$.
**Step 2.** Spur path calculation for both single and multiple deviation link scenarios:
03: Call *FindSpurPath-Yen*($\tilde{A}_i^j, \tilde{N}_i^j$) to calculate spur path $\bar{s}_i^j$, and Return $\bar{s}_i^j$.

**Sub-Procedure: *GetCandidatePath-V&F***
**Input:** Deviation links set $\tilde{A}_i^j$, and removed node set $\tilde{N}_i^j$.
**Return:** Candidate path $q_{iv}^0$
01: Set candidate path $q_{iv}^0 \coloneqq \emptyset$ and its cost $t(q_{iv}^0) := \infty$.
02: For each successor link $a(n_i^j, n_v) \in \{\mathrm{SUCC}(n_i^j) - \tilde{A}_i^j\}$
03: If $t(n_i^j, n_v) + t(q_v^0) < t(q_{iv}^0)$, then
Set $q_{iv}^0 \coloneqq a(n_i^j, n_v) \oplus q_v^0$ and $t(q_{iv}^0) \coloneqq t(n_i^j, n_v) + t(q_v^0)$.
04: End for
05: Return $q_{iv}^0$

---

Clearly, the V–F algorithm can obtain the optimal solution of the KSP problem. Its computational performance depends on the spur path reuse rate, which is the percentage of spur paths that can be successfully identified. As the spur path reuse rate approaches zero, the V–F algorithm degrades to Yen's algorithm. Previous studies have found that a spur path reuse rate of about 10–50% can be achieved in practice (Vanhove and Fack, 2012; Chen *et al.*, 2020). To further improve the reuse rate, a generalized spur path reuse technique is proposed and described in the next section.

## 3. Proposed KSP-SPR algorithm

This section presents the proposed KSP-SPR algorithm using a generalized spur path reuse technique. According to the above definition, a spur path, $\bar{s}_i^j$, is the shortest path from deviation node $n_i^j$ to destination $d$ in the modified network $\mathrm{G}_i^j$, where all links in $\tilde{A}_i^j$ and all nodes in $\tilde{N}_i^j$

were removed from the network $\mathrm{G}$. Let $\widetilde{\mathrm{G}}_i^j = \{\tilde{A}_i^j,\ \tilde{N}_i^j\}$ be the set of removed links and nodes, i.e., $\mathrm{G}_i^j = \mathrm{G} - \widetilde{\mathrm{G}}_i^j$. Let $Q_i^j$ be the solution space consisting of all paths from deviation node $n_i^j$ to destination $d$ in the modified network $\mathrm{G}_i^j$. Obviously, spur path $\bar{s}_i^j$ is the path with minimum cost among all paths in $Q_i^j$. Given another solution space $Q_{iv}^u$ consisting of all paths from the same node to destination $d$ in the modified network $\mathrm{G}_{iv}^u$, the following proposition can be made.

**Proposition 1.** Given two solution spaces, $Q_i^j$ and $Q_{iv}^u$, then $Q_i^j \subset Q_{iv}^u$ if $\widetilde{\mathrm{G}}_i^j \supset \widetilde{\mathrm{G}}_{iv}^u$ holds.
**Proof.** This follows from the definition of a solution space. Because $\widetilde{\mathrm{G}}_i^j \supset \widetilde{\mathrm{G}}_{iv}^u$, the solution space $Q_{iv}^u$ relaxes the constraints on not passing certain nodes and links used in $Q_i^j$. Therefore, $Q_i^j \subset Q_{iv}^u$ holds. □

Let $q_{iv}^u$ be the path with minimum cost among all paths in $Q_{iv}^u$. According to Proposition 1, the calculated path, $q_{iv}^u$, can be a candidate path for spur path $\bar{s}_i^j$ if $\widetilde{\mathrm{G}}_i^j \supset \widetilde{\mathrm{G}}_{iv}^u$ holds. This candidate path can be reused successfully if $q_{iv}^u \cap \widetilde{\mathrm{G}}_i^j = \emptyset$ also holds, according to the following proposition.

**Proposition 2.** Given $q_{iv}^u$, then $\bar{s}_i^j = q_{iv}^u$, if $\widetilde{\mathrm{G}}_i^j \supset \widetilde{\mathrm{G}}_{iv}^u$ and $q_{iv}^u \cap \widetilde{\mathrm{G}}_i^j = \emptyset$ hold.
**Proof.** If $\widetilde{\mathrm{G}}_i^j \supset \widetilde{\mathrm{G}}_{iv}^u$ holds, then $Q_i^j \subset Q_{iv}^u$ according to Proposition 1. Then $\bar{s}_i^j \in Q_{iv}^u$. According to the definition, $t(q_{iv}^u) \leq t(q_{iv,w}^u)$ for any path $\forall q_{iv,w}^u \in Q_{iv}^u$. Hence, $t(q_{iv}^u) \leq t(\bar{s}_i^j)$. If $q_{iv}^u \cap \widetilde{\mathrm{G}}_i^j = \emptyset$ holds, then $q_{iv}^u$ is a feasible solution in $Q_i^j$. According to the definition of $\bar{s}_i^j$, $t(\bar{s}_i^j) \leq t(q_{i,w}^j)$ holds for any path $\forall q_{i,w}^j \in Q_i^j$. Hence, $t(q_{iv}^u) \geq t(\bar{s}_i^j)$. Therefore, $t(q_{iv}^u) = t(\bar{s}_i^j)$ holds. □

The V–F algorithm reuses the results maintained in the generated shortest path tree, i.e., $q_{iv}^0$ calculated when only deviation links were removed from G. However, $q_{iv}^0 \cap \widetilde{\mathrm{G}}_i^j \neq \emptyset$ can often happen, leading to reuse failure. According to Propositions 1 and 2, the spur path reuse technique can be generalized to reuse paths calculated when a certain number of nodes were also removed from G. Fig. 3(a) illustrates this idea using a simple example. The spur path $\bar{s}_4^3 = \{3, 7, 4\}$ was calculated when deviation link $a(3, 4)$ and $\tilde{N}_4^3 = \{1,2,5\}$ were included in $\widetilde{\mathrm{G}}_4^3$. In the later iteration, it is necessary to determine spur path $\bar{s}_5^7$ when deviation link $a(3, 4)$ and $\tilde{N}_5^7 = \{1,2,6,5\}$ are included in $\widetilde{\mathrm{G}}_5^7$. In this case, the calculated $\bar{s}_4^3$ can be reused successfully, i.e., $\bar{s}_5^7 = \bar{s}_4^3$, because both $\widetilde{\mathrm{G}}_5^7 \supset \widetilde{\mathrm{G}}_4^3$ and $\bar{s}_4^3 \cap \widetilde{\mathrm{G}}_5^7 = \emptyset$ hold. However, $\bar{s}_4^3$ cannot be a candidate path for determining spur path $\bar{s}_4^5$ because $\widetilde{\mathrm{G}}_4^5 \supset \widetilde{\mathrm{G}}_4^3$ does not hold. Fig. 3(b) shows the same example using a solution space perspective. Clearly, a path can be reused only in those cases whose solution spaces are contained by the path's solution space.

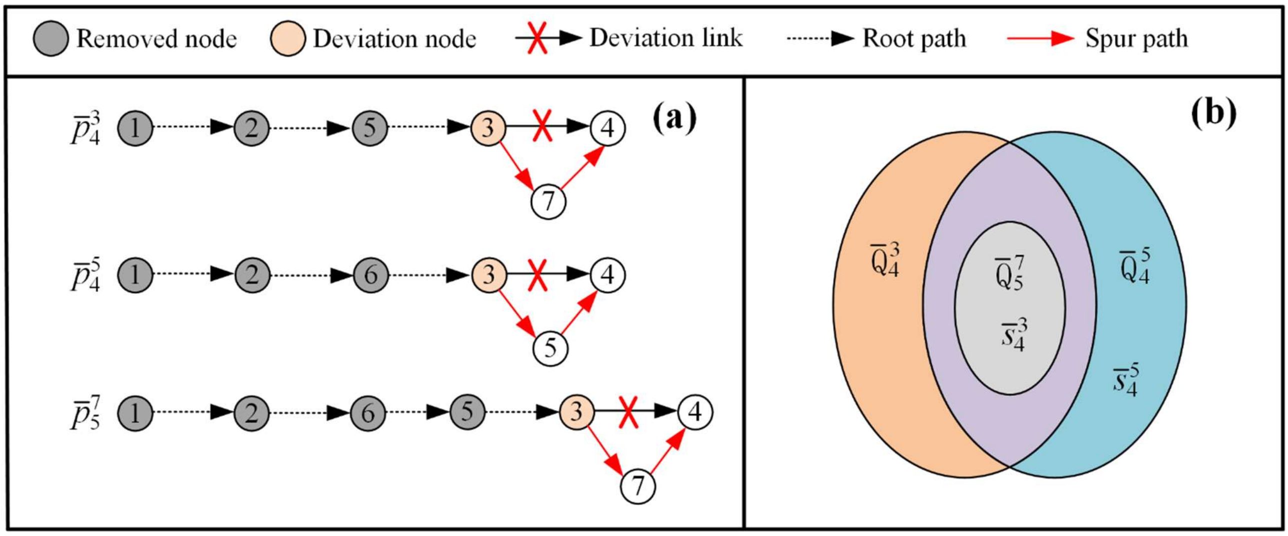


**Fig. 3**. Illustrative example of the generalized spur path reuse concept: (a) Spur paths; (b) Solution spaces.

Built on the idea described above, a generalized spur path reuse technique is proposed to reuse all possible candidate paths during the KSP finding process. The proposed technique consists of two procedures: candidate path calculation, and candidate path retrieval. To determine a spur path $\bar{s}_i^j$, the candidate path calculation procedure first tries to reuse the results maintained in the generated shortest path tree, i.e., $q_{iv}^0$ as the V–F algorithm. If the reuse of $q_{iv}^0$ fails, the procedure calculates a set of candidate paths, $q_{iv}^u$, by incrementally removing a few of the nodes that form cycles in $q_{iv}^u \oplus \bar{r}_i^j$ (i.e., $q_{iv}^u \cap \tilde{N}_i^j$). The spur path, $\bar{s}_i^j$, can be determined when the calculated candidate path $q_{iv}^u$ has no cycle in $q_{iv}^u \oplus \bar{r}_i^j$ (i.e., $q_{iv}^u \cap \tilde{N}_i^j = \emptyset$). In the worst case, all nodes along the root path are removed, and the procedure can guarantee to determine the optimal spur path. This candidate path calculation strategy follows the idea that the fewer the nodes removed, the larger is the solution space formed, and the higher is the possibility of candidate path reuse in later iterations.

Fig. 4 illustrates the candidate path calculation procedure using the same example as in Fig. 3. To determine spur path $\bar{s}_4^3$, the deviation link $a(3,4)$ is first removed from the network, and candidate path $q_{34}^0 = \{3,7,2,4\}$ is retrieved from the generated shortest path tree. Because $\tilde{N}_4^3 = \{1,2,5\}$ and $q_{34}^0 \cap \tilde{N}_4^3 = \{2\}$, reuse of $q_{34}^0$ fails. Then Node 2, which forms a cycle, is removed from the network, and a candidate path $q_{34}^1 = \{3,6,5,4\}$ is calculated. Because $q_{34}^1 \cap \tilde{N}_4^3 = \{5\}$ holds, the procedure continues. Node 5, which forms a cycle, is further removed, and $q_{34}^2 = \{3,7,4\}$ is calculated. Because $q_{34}^2 \cap \tilde{N}_4^3 = \emptyset$ holds, the spur path $\bar{s}_4^3 = q_{34}^2$ is determined.

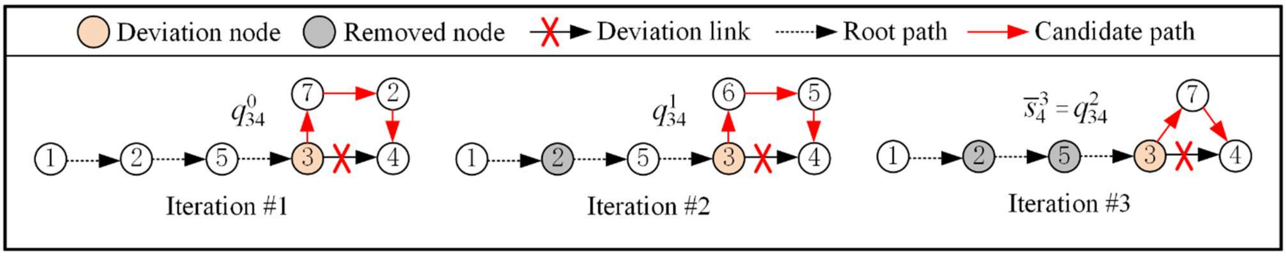


**Fig. 4.** Illustrative example of candidate path calculation procedure.

In the proposed technique, all calculated candidate paths are stored in every iteration and retrieved for potential reuse in later iterations. A tree structure, denoted by $T_{iv}$, is constructed to store all candidate paths, from the same deviation node $n_i$ to the destination $d$, calculated when the same deviation link $a(n_i, n_v)$ is removed from G. Fig. 5 illustrates the structure of the candidate path tree using the sample example of Figs. 3 and 4. The tree has a single root node, $n_{iv}^0$, storing the candidate path (e.g., $q_{34}^0$) calculated when no node, but only the deviation link (e.g., $a(3{,}4)$) is removed. A direct child node (e.g., Node 2) of the root node stores the candidate path (e.g., $q_{34}^1$) calculated when the node (i.e., Node 2) is removed. Without loss of generality, a child node $n_{iv}^w$ (e.g., Node 5) stores the candidate path (e.g., $q_{34}^2$) calculated when the node (i.e., Node 5) and all its parent nodes (i.e., Node 2) along the path from the root node are removed. Therefore, using this tree structure, candidate paths can be stored and maintained at the deviation link. Note that the proposed technique is not applicable to the scenario with multiple deviation links removed. This is due to its difficulties in storing candidate paths and its small number of cases compared to the single deviation link scenario.

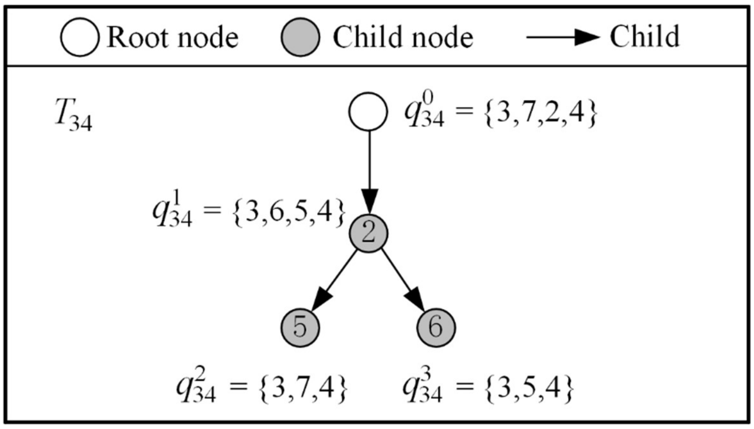


**Fig. 5.** Illustrative example of the candidate path tree structure.

In the proposed technique, the candidate path retrieval procedure tries to reuse candidate paths stored in the tree, $T_{iv}$, using a breadth-first-search strategy. The procedure first searches the root node $n_{iv}^0$ by adding it to a *list* structure using the first-in-first-out principle. In each iteration, the first node $n_{iv}^w$ of the *list* is selected and removed. If candidate path $q_{iv}^w$ stored at the selected node satisfies $q_{iv}^w \cap \widetilde{N}_i^j = \emptyset$, then the optimal spur path $\bar{s}_i^j$ is determined as $q_{iv}^w$, and the procedure terminates. Otherwise, all child nodes of selected node $n_{iv}^w$ belonging to $\widetilde{N}_i^j$ are identified as candidate nodes and added to the *list* for further search. The procedure continues the search until either spur path $\bar{s}_i^j$ is determined or the *list* is empty (i.e., reuse failure). When reuse fails, the procedure retrieves the last node $q_{iv}^u$ in the *list* with the maximum depth among all candidate nodes in $T_{iv}$. The retrieved $q_{iv}^u$ is used as an initial candidate path to speed up the candidate path calculation procedure because computational efforts for calculating $q_{iv}^u$ and all paths stored at its parent nodes in $T_{iv}$ are saved.

Fig. 5 illustrates an example of reusing candidate paths calculated in Fig. 4 to determine $\bar{s}_4^5$ with $\widetilde{N}_4^5 = \{1,2,6\}$, as shown in Fig. 3. Initially, three calculated candidate paths (i.e., $q_{34}^0$, $q_{34}^1$, $q_{34}^2$) are stored in the tree, $T_{34}$. The candidate path retrieval procedure first searches the root node $n_{34}^0$ storing $q_{34}^0$. Because $q_{34}^0 \cap \widetilde{N}_4^5 = \{2\}$ holds, $q_{34}^0 \neq \bar{s}_4^5$. Then the procedure continues to search its child node $n_{34}^1$ (i.e., Node 2) satisfying $n_{34}^1 \in \widetilde{N}_4^5$. Here, $q_{34}^1 \neq \bar{s}_4^5$ holds because $q_{34}^1 \cap \widetilde{N}_4^5 = \{6\}$. The child node of $n_{34}^1$, $n_{34}^2 = \{5\}$, does not belong to $\widetilde{N}_4^5$, and therefore it is not added to the *list* for further search. Consequently, the *list* is empty, and the procedure terminates. The last node in the *list* is $n_{34}^1$, and hence candidate path $q_{34}^1$ is retrieved as the initial candidate path. Subsequently, the candidate path calculation procedure can use $q_{34}^1$ without having to calculate it, as well as $q_{34}^0$ stored at its parent node in the tree. The procedure calculates a new candidate path $q_{34}^3$, which is added to tree $T_{34}$ and stored at node $n_{34}^3$=$\{6\}$ as the child node of $n_{34}^1$.

Using this generalized spur path reuse technique, an efficient KSP algorithm, called KSP-SPR, is proposed in this paper for exactly solving the KSP problem. In a similar manner to the V–F algorithm, the proposed KSP-SPR algorithm follows the same generic procedure as Yen's algorithm, but with two modifications. The first modification is on Line 01 of Table 1, by using *the backward one-to-all Dijkstra's algorithm* to generate the first shortest path $p^1$. The second modification is to use the *FindSpurPath-SPR* procedure instead of the *FindSpurPath-Yen* procedure to determine spur path $\bar{s}_i^j$.

Table 4 gives the detailed steps of the *FindSpurPath-SPR* procedure. It classifies the spur path calculation problem into single and multiple deviation link scenarios. For the first scenario (i.e., $|\tilde{A}_i^j| = 1$), the procedure first retrieves the candidate path tree, $T_{iv}$, stored at deviation link $a(n_i^j, n_{i+1}^j)$. If $T_{iv}$ is empty, the procedure calls the *GetCandidatePath-V&F* sub-procedure (see Table 3) to determine $q_{iv}^0$ and add it to $T_{iv}$ as the root node. Subsequently, the procedure calls the *RetrieveCandidatePath* sub-procedure, of which the detailed steps are shown in Table 4 and described in the introduction to candidate path retrieval. If the sub-procedure determines spur path $\bar{s}_i^j \neq \emptyset$, the procedure terminates. Otherwise, the sub-procedure returns an initial candidate path $q_{iv}^u \neq \emptyset$. Using $q_{iv}^u$ as an input, the procedure finally calls the *CalculateCandidatePaths* sub-procedure to calculate spur path $\bar{s}_i^j$.

In the second scenario (i.e., $|\tilde{A}_i^j| > 1$), the procedure simply applies a procedure similar to the V–F algorithm to calculate spur path $\bar{s}_i^j$. The only difference is the use of the *A* algorithm* instead of the *forward one-to-one Dijkstra's algorithm* to speed up the spur path $\bar{s}_i^j$ calculation. This A* algorithm uses a heuristic function to assign higher priorities to nodes closer to the destination (Zeng and Church, 2009). Let $h(n_u)$ be the heuristic function of node $n_u$ representing the estimated lower bound for the shortest path cost from node, $n_u$, to the destination. Such an A* algorithm can determine the optimal shortest path if $h(n_u)$ is admissible and satisfies the triangle inequality property, i.e., $h(n_u) \leq h(n_v) + a(n_u, n_v)$. Because shortest paths from all nodes to the destination were calculated during the $p^1$ search, they are used as admissible $h(n_u)$. The technique of avoiding label initialization proposed by Chen *et al.* (2014) is also incorporated into the A* algorithm to further improve spur path calculation performance. In the conventional A* algorithm, the initialization step is required to set the null labels of all nodes before carrying out the shortest path search. However, this label initialization step can lead to a heavy computational burden on the KSP finding process, with very many shortest path searches in large networks. This label initialization step can be avoided by assigning a unique ID to each shortest path search (Chen et al., 2014).

**Table 4.** Detailed steps of the *FindSpurPathSPR* procedure

**Procedure: *FindSpurPath-SPR***
**Input:** Deviation links set $\tilde{A}_i^j$, and removed node set $\tilde{N}_i^j$.
**Return:** Spur path $\bar{s}_i^j$.
**Step 1.** Spur path determination for the single deviation link scenario:
01: If $|\tilde{A}_i^j| = 1$, Then
02: Get candidate path tree $T_{iv}$ stored at deviation link $a(n_i^j, n_{i+1}^j)$.
03: If $T_{iv} = \{\emptyset\}$, Then Call *GetCandidatePath-V&F*$(\tilde{A}_i^j, \tilde{N}_i^j)$ to determine candidate path $q_{iv}^0$, and Set $T_{iv} := \{q_{iv}^0\}$.

Call *RetrieveCandidatePath*($T_{iv}$, $\widetilde{N}_i^j$) to determine $\bar{s}_i^j$ and initial candidate path $q_{iv}^u$.
If $\bar{s}_i^j \neq \emptyset$, Then Return $\bar{s}_i^j$.
Call *CalculateCandidatePaths*($T_{iv}$, $\widetilde{N}_i^j$, $q_{iv}^u$) to calculate $\bar{s}_i^j$, and Return $\bar{s}_i^j$.
End If

**Step 2.** Spur path calculation for the multiple deviation link scenario:
Call *GetCandidatePath-V&F*($\tilde{A}_i^j, \widetilde{N}_i^j$) to determine candidate path $q_{iv}^0$.
If $q_{iv}^0 \neq \emptyset$ and $q_{iv}^0 \cap \widetilde{N}_i^j = \emptyset$, then Set $\bar{s}_i^j \coloneqq q_{iv}^0$ and Return $\bar{s}_i^j$.
Remove $\tilde{A}_i^j$ and $\widetilde{N}_i^j$ from G.
Call the $A*$ *algorithm* to calculate spur path $\bar{s}_i^j$.
Restore $\tilde{A}_i^j$ and $\widetilde{N}_i^j$ to G.
Return $\bar{s}_i^j$.

**Sub-Procedure: *RetrieveCandidatePath***
**Input:** Candidate path tree $T_{iv}$, and Removed node set $\widetilde{N}_i^j$.
**Return:** Spur path $\bar{s}_i^j$, and Initial candidate path $q_{iv}^u$
Set $\bar{s}_i^j := \emptyset$, and $q_{iv}^u := \emptyset$.
Add root node $n_{iv}^0$ of $T_{iv}$ into $list \coloneqq \{n_{iv}^0\}$.
While $list \neq \emptyset$
Select the first node $n_{iv}^w$ from $list$, and Set $list \coloneqq list - \{n_{iv}^w\}$.
Retrieve the candidate path $q_{iv}^w$ stored at $n_{iv}^w$.
If $q_{iv}^w \cap \widetilde{N}_i^j = \emptyset$, Then Set $\bar{s}_i^j := q_{iv}^w$ and Return.
For each child node $n_{iv}^\theta$ of $n_{iv}^w$
If $n_{iv}^\theta \in \widetilde{N}_i^j$, Then Set $list \coloneqq list \cup \{n_{iv}^\theta\}$.
End for
End while
Set $q_{iv}^u \coloneqq q_{iv}^w$, and Return.

**Sub-Procedure: *CalculateCandidatePaths***
**Input:** Candidate path tree $T_{iv}$, Removed node set $\widetilde{N}_i^j$, Initial candidate path $q_{iv}^u$.
**Return:** Spur path $\bar{s}_i^j$.
**Step 1.** Initialization:
Set parent node $n_{iv}^u$ as the node storing $q_{iv}^u$.
Set current node $n_{iv}^w \coloneqq n_{iv}^u$, and Set removed node set $\mathrm{N}_{iv}^u := \{\emptyset\}$.
While $n_{iv}^w \neq \emptyset$
Set $\mathrm{N}_{iv}^u \coloneqq \mathrm{N}_{iv}^u \cup \{n_{iv}^w\}$, and Remove $n_{iv}^w$ from G.
Set $n_{iv}^w$ as its direct parent node.
End while
Remove deviation link $a(n_i^j, n_{i+1}^j)$ from G.

**Step 2.** Candidate path calculation:
Set *isFirstSearch* := True.
While $q_{iv}^u \cap \widetilde{N}_i^j \neq \emptyset$
Select node $n_{iv}^w$ nearest to deviation node $n_i^j$ in $q_{iv}^u \cap \widetilde{N}_i^j$.
Set $\mathrm{N}_{iv}^u \coloneqq \mathrm{N}_{iv}^u \cup \{n_{iv}^w\}$, and Remove $n_{iv}^w$ from G.
Call the $A*$ *algorithm* to calculate candidate path $q_{iv}^w$.
Store $q_{iv}^w$ at $n_{iv}^w$, and Add $n_{iv}^w$ into $T_{iv}$ as direct child node of $n_{iv}^u$.

14: Set $n_{iv}^{u} \coloneqq n_{iv}^{w}$.
15: End While
**Step 3.** Network restoration:
16: Restore deviation link $a(n_i^j, n_{i+1}^j)$ and all nodes in $N_{iv}^{u}$ to G.
17: Set $\bar{s}_i^j \coloneqq q_{iv}^{u}$, and Return $\bar{s}_i^j$.

---

The detailed steps of the *CalculateCandidatePaths* sub-procedure are shown in Table 4 and described below. The sub-procedure consists of three steps. In the first step, node $n_{iv}^{u} \in T_{iv}$ (storing initial candidate path $q_{iv}^{u}$) and all its parent nodes in $T_{iv}$ as well as the deviation link $a(n_i^j, n_{i+1}^j)$ are removed from the network G. In the second step, a set of nodes forming cycles are identified as $q_{iv}^{u} \cap \tilde{N}_i^j$, where $\tilde{N}_i^j = \bar{r}_i^j - \{n_i^j\}$ consists of all nodes along the root path excluding the deviation node $n_i^j$. Among all nodes in $q_{iv}^{u} \cap \tilde{N}_i^j$, only one node $n_{iv}^{w}$ nearest to the deviation node $n_i^j$ is selected and removed from G. Then candidate path $q_{iv}^{w}$ is calculated as the shortest path in G from deviation node $n_i^j$ to the destination $d$ using the $A*$ *algorithm*, which also uses the $p^1$ search results as $h(n_u)$ and the technique of avoiding label initialization. The calculated $q_{iv}^{w}$ is stored at node $n_{iv}^{w}$, which is further added to candidate path tree $T_{iv}$ as a direct child node of $n_{iv}^{u}$ storing $q_{iv}^{u}$. The candidate path calculations iterate until the calculated candidate path does not contain any node in $\tilde{N}_i^j$, i.e., the spur path $\bar{s}_i^j$ is determined. In the last step of the sub-procedure, all removed nodes and links are restored to G, and the determined spur path $\bar{s}_i^j$ is returned.

The optimality of the proposed KSP-SPR algorithm can be proved as follows.

**Proposition 3**. The proposed KSP-SPR algorithm can exactly solve the problem of finding k shortest simple paths.
**Proof.** See Proposition A1 in the Appendix. □

The worst-case complexity of the proposed KSP-SPR algorithm is analyzed. The *FindShortestPath-LPA** sub-procedure requires $\mathrm{O}\{|A| + |N|\mathrm{Log}|N|\}$ with implementation of the F-heap data structure (Fredman and Tarjan 1987) as the priority queue. Accordingly, the *CalculateCandidatePaths* sub-procedure runs $\mathrm{O}\{|N|(|A| + |N|\mathrm{Log}|N|)\}$. Because the *RetrieveCandidatePath* sub-procedure requires $\mathrm{O}\{|N|\}$ and the $A*$ *algorithm* procedure requires $\mathrm{O}\{(|A| + |N|\mathrm{Log}|N|)\}$, the proposed KSP-SPR algorithm has worst-case complexity $\mathrm{O}\{\mathrm{K}|N|^2(|A| + |N|\mathrm{Log}|N|)\}$. Although this complexity is not as good as Yen's algorithm in the theoretical worst case, it is shown in the following section that the proposed KSP-SPR algorithm in practice runs much faster than Yen's algorithm.

## 4. Computational Experiments

This section reports the computational performance of the proposed KSP-SPR algorithm. Three state-of-the-art KSP algorithms were also implemented for comparison, including the improved Yen's algorithm (Yen, 1971; Hershberger et al., 2007), the V–F algorithm (Vanhove and Fack, 2012), and the KSP-LPA* algorithm (Chen et al., 2020). All four algorithms were coded in the same C# programming language and used the same F-heap data structure (Fredman and Tarjan 1987) as the priority queue in all shortest path calculation procedures. All computational experiments ran on a desktop equipped with an Intel quad-core CPU at 2.6 GHz and 8 GB of memory, and only a single processor was used.

Five real road networks with different sizes were collected and tested. Among them, three networks, Winnipeg, Austin, and Chicago, were downloaded from a well-known network repository for transportation research (https://github.com/bstabler/TransportationNetworks). Link travel times were provided in these three networks and utilized for computational experiments. Two larger road networks, Wuhan and Shenzhen, were downloaded from OpenStreetMap Data Extracts (http://download.geofabrik.de/index.html). Link lengths were used in these two networks for computational experiments. Table 5 summarizes the characteristics of all five test networks. For each road network, 100 O-D pairs were randomly generated and tested for all four algorithms using different K values, i.e., 10, 50, 100, 500, and 1000.

**Table 5**. Basic characteristics of testing road networks

| Network | Winnipeg | Austin | Chicago | Wuhan | Shenzhen |
|---|---|---|---|---|---|
| Number of nodes | 1,052 | 7,388 | 12,982 | 40,347 | 57,229 |
| Number of links | 2,836 | 18,961 | 39,018 | 156,142 | 162,756 |

Table 6 reports the computational performance of the proposed KSP-SPR algorithm in all five test networks when K = 1000. The computational time consumed by the proposed KSP-SPR algorithm can be divided into three parts, including the first shortest path ($p^1$) search and the spur path calculations for the single ($|\tilde{A}_i^j| = 1$) and multiple ($|\tilde{A}_i^j| > 1$) deviation link scenarios. The $p^1$ search was performed only once, although it was relatively computationally intensive in large road networks. The single deviation link scenario dominated the spur path calculation process for all test networks. For example, the single deviation link scenario accounted for 97.6% of total spur path calculations and consumed 92.7% of total computation time for the Wuhan network. Note that 99.5% of spur paths under the single deviation link scenario can be directly determined by reusing stored candidate paths and that only 0.5% of spur paths were determined by shortest path searches to calculate candidate paths. This promising reuse rate ($\tilde{r}$) was achieved because of the introduced generalized spur path reuse technique. It also justified the focus of this introduced technique on the single deviation link scenario because the multiple deviation link scenario was minor for all test networks. Without such a technique, low reuse rates were achieved for the multiple deviation link scenario by only reusing the $p^1$ search results, such as $\tilde{r}$=24.1% on the Wuhan network.

**Table 6**. Computational performance of the KSP-SPR algorithm when K = 1000

| Network | $p^1$ search | $\lvert\tilde{A}_i^j\rvert = 1$ scenarios | | | $\lvert\tilde{A}_i^j\rvert > 1$ scenarios | | | Total |
|---|---|---|---|---|---|---|---|---|
| | $\tilde{t}$ | $\widetilde{m}$ | $\tilde{r}$ | $\tilde{t}$ | $\widetilde{m}$ | $\tilde{r}$ | $\tilde{t}$ | $\tilde{t}$ |
| Winnipeg | 0.004 | 6,028 | 98.3% | 0.22 | 439 | 26.8% | 0.02 | 0.24 |
| Austin | 0.10 | 17,077 | 98.9% | 2.15 | 328 | 18.0% | 0.09 | 2.34 |
| Chicago | 0.23 | 20,270 | 99.1% | 2.16 | 696 | 20.6% | 0.13 | 2.52 |
| Wuhan | 0.90 | 28,345 | 99.5% | 15.4 | 709 | 24.1% | 1.10 | 17.4 |
| Shenzhen | 1.40 | 36,222 | 99.4% | 108.6 | 614 | 18.2% | 0.30 | 110.3 |

$\widetilde{m}$: Average number of spur path calculations (100 runs);
$\tilde{r}$: Average percentage of spur paths reused from stored candidate paths (100 runs);
$\tilde{t}$: Average computational times in seconds (100 runs).

Table 7 summarizes the candidate path calculations and storage for the single deviation link scenario for all test networks. As shown, few candidate paths were calculated and stored for all test networks. For example, in the Wuhan network, only 160 links were maintained in a candidate path tree. The average number of nodes in the candidate path tree was 2.15, indicating that on average only 2–3 candidate paths were sufficient to represent spur paths under various root paths. The computer memory for storing candidate paths was approximately 154.3 KB. Compared to the

Wuhan network, more candidate paths were calculated and stored on the Shenzhen network. There were 202 links with candidate path trees, and the average number of nodes was 4.24. Nevertheless, the computer memory for storing these candidate paths was satisfactory, at 746.4 KB. This result confirmed the storage feasibility of using the generalized spur path reuse technique in large networks.

**Table 7**. Spur path calculations and reuses in the KSP-SPR algorithm for $\left|\tilde{A}_i^j\right| = 1$ scenarios

| Network | Stored candidate paths | | | Computational times in seconds | | |
|---|---|---|---|---|---|---|
| | No. of links stored trees | Average tree size | Memory requirement | This study | A* | Dijkstra |
| Winnipeg | 77 | 2.03 | 17.5 KB | 0.22 | 0.23 | 0.48 |
| Austin | 124 | 1.69 | 39.6 KB | 2.15 | 2.43 | 5.38 |
| Chicago | 189 | 1.52 | 53.0 KB | 2.16 | 2.59 | 9.14 |
| Wuhan | 160 | 2.15 | 154.3 KB | 15.4 | 16.9 | 83.95 |
| Shenzhen | 212 | 4.24 | 746.4 KB | 108.6 | 115.5 | 631.4 |

No. of links stored trees: Number of links stored candidate path trees (100 runs);
Average tree size: Average number of nodes in stored candidate path trees (100 runs).

Table 7 also summarizes the computation times required for candidate path calculations for the single deviation link scenario. To speed up the candidate path calculations, the proposed KSP-SPR algorithm used both the A* algorithm and the technique for avoiding label initialization. To distinguish their effectiveness, the A* algorithm and the forward one-to-to Dijkstra's algorithm were also used to calculate the same set of candidate paths for comparison. The A* algorithm used the $p^1$ search results as $h(n_u)$, whereas Dijkstra's algorithm set $h(n_u) = 0$. Comparing their results, it is clear that use of the $p^1$ search results as $h(n_u)$ significantly enhanced candidate path calculation performance for all test networks by as much as 4.0 times (i.e., 83.95/16.9 –1) on the Wuhan network. This was the case because using such a heuristic function can speed up shortest path search by assigning higher priorities to nodes closer to the destination. Comparing the results of "A*" and "this study" in Table 7 shows the effectiveness of the technique to avoid label initialization. Using this technique further improved candidate path calculation performance for all test networks, e.g., by 9.7% for the Wuhan network.

Fig. 6 reports the computational performance of four algorithms on the Wuhan network using the same set of 100 OD pairs and setting K = 1000. The horizontal axis represents the 100 random queries, and the vertical axis represents the computational times on a logarithmic scale. The computational performance of all four algorithms was compared to that of the improved Yen's algorithm. The improved Yen's algorithm delivered the worst performance among the four algorithms tested because it repeatedly performed a forward one-to-one shortest path search to calculate every spur path, leading to considerable computational overhead. The V–F algorithm ran about 64.4% (i.e., 1734.3/1055.0 – 1) (see Table 8) faster than the improved Yen's algorithm because the V–F algorithm saved 51% of spur path calculations by reusing the $p^1$ search results. As shown in Fig. 6, the proposed KSP-SPR algorithm further sped up the V–F algorithm by 60.6 times (i.e., 1055.0 / 17.4 – 1) on the Wuhan network, as expected. The proposed KSP-SPR algorithm using the generalized spur path reuse technique reused not only the $p^1$ search results, but also the stored candidate paths, and thereby achieved a much higher reuse rate (99.5%). In addition, both the A* algorithm and the technique to avoid label initialization were used to improve the candidate path calculations (see Table 7).

Fig. 6 also presents the computational performance of the KSP-LPA* algorithm on the Wuhan network. As shown, the KSP-LPA* algorithm sped up the improved Yen's algorithm by 1.2 times (i.e. 1734.3 / 794.7 – 1). However, the proposed KSP-SPR algorithm was still the best algorithm, with its computation time being 44.6 times (i.e. 794.7 / 17.4 – 1) shorter than that of the KSP-LPA* algorithm.

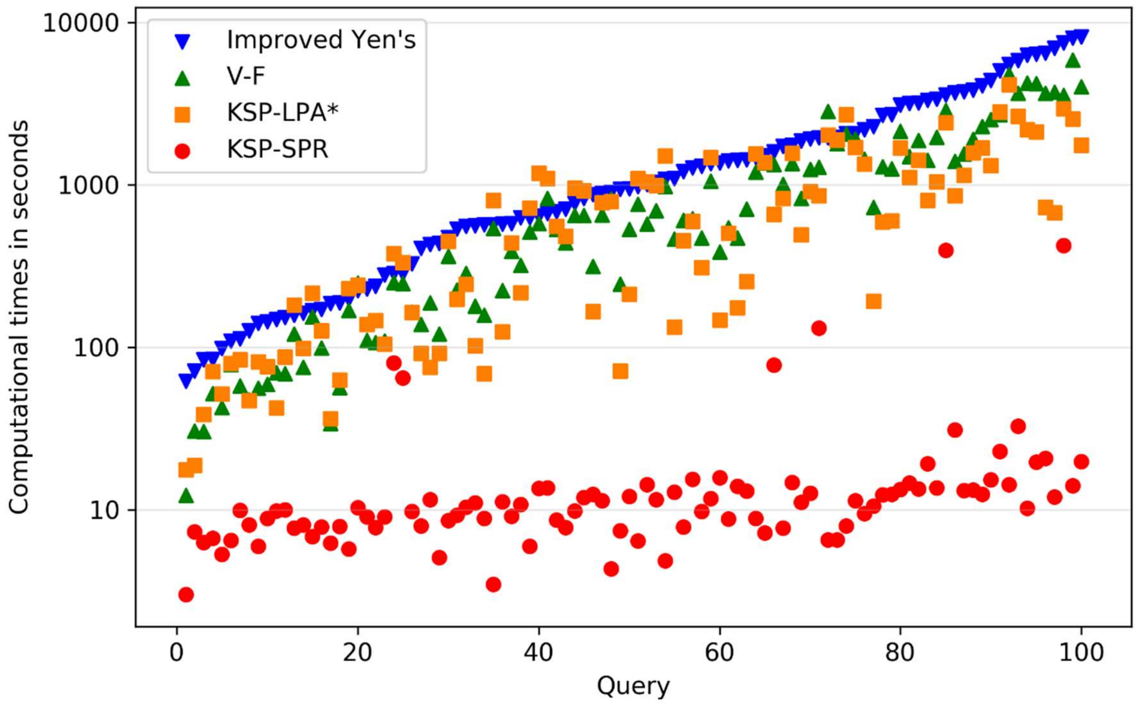


**Fig. 6**. Computational times(seconds) of all four algorithms on the Wuhan network.

Table 8 shows the computational performance of the four KSP algorithms on five test networks of different sizes when K = 1000. As expected, the computational performance of all algorithms degraded with increasing network size. For instance, the improved Yen's algorithm consumed 4.4 s to calculate K shortest paths on the Winnipeg network with 1052 nodes. When the Shenzhen network was used, the number of nodes was increased by 53.4 times, but the computation times of the improved Yen's algorithm increased by 927.2 times. The proposed KSP-SPR algorithm had a moderate degradation rate. For example, when the Shenzhen network was used, the computation times of the proposed KSP-SPR algorithm increased by only 458.6 times.

**Table 8**. The experiment results of KSP algorithms on the five testing road networks (K=1000).

| Network | Improved Yen's | KSP-LPA* | V-F | | KSP-SPR | |
|---|---|---|---|---|---|---|
| | $\tilde{t}$ | $\tilde{t}$ | $\tilde{t}$ | $\tilde{r}$ | $\tilde{t}$ | $\tilde{r}$ |
| Winnipeg | 4.4 | 3.7 | 3.0 | 40.6% | 0.24 | 98.3% |
| Austin | 143.3 | 39.8 | 95.7 | 45.8% | 2.34 | 98.9% |
| Chicago | 188.1 | 64.2 | 119.9 | 51.6% | 2.52 | 99.1% |
| Wuhan | 1734.3 | 794.7 | 1055.0 | 60.1% | 17.4 | 99.5% |
| Shenzhen | 4084.2 | 1404.8 | 2404.8 | 59.4% | 110.3 | 99.4% |

$\tilde{t}$: Average computational times in seconds (100 runs)

$\tilde{r}$: Average spur path reuse rates on $\left|\tilde{A}_i^j\right| = 1$ scenarios (100 runs)

Fig. 7 shows the computational performance of the four algorithms under different K values on the Wuhan network. As expected, the computational performance of all four algorithms degraded with increasing K because more shortest paths were calculated. The proposed KSP-SPR algorithm ran consistently fastest among all algorithms under various K values, and its computational advantage became more obvious with larger K. For example, on the Wuhan network, the proposed KSP-SPR algorithm ran 34.1 times (i.e., 56.9 / 1.62 – 1) faster than the improved Yen's algorithm when K = 10 was used. This improvement ratio increased to 98.7 times when K = 1000.

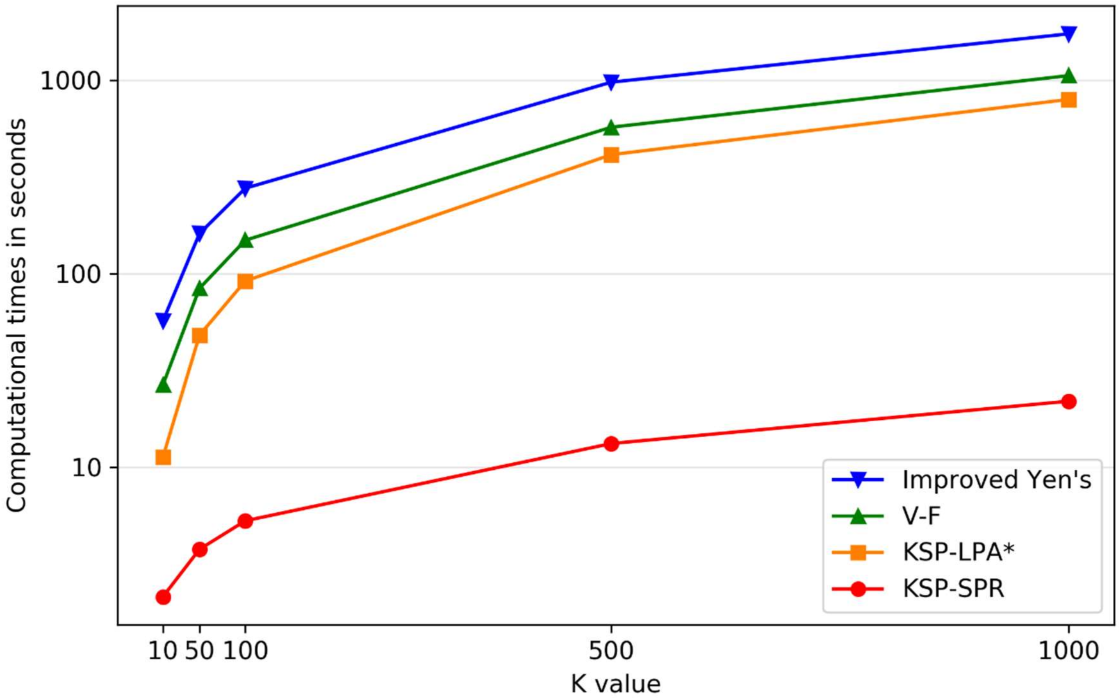


**Fig. 7**. The computational times of four testing algorithms under different K values on Wuhan network.

## 5. Conclusions

This study has proposed an efficient solution algorithm, called KSP-SPR, for exactly solving the K shortest simple paths (KSP) problem in large-scale road networks. The proposed KSP-SPR algorithm followed the generic procedure of Yen's algorithm (1971), but introduced a generalized spur path reuse technique that was proved in Propositions 1 and 2. Using this technique, the proposed KSP-SPR algorithm not only reused the $p^1$ search results like the V–F algorithm (Vanhove and Fack, 2012), but also reused candidate paths calculated during the KSP finding process. A tree structure was designed to store the candidate paths, and a breadth-first-search strategy was used to retrieve stored candidate paths. To improve candidate path calculation performance, the $p^1$ search results were used as the heuristic function of the A* algorithm, and a technique to avoid label initialization was also used. To demonstrate the efficiency of the proposed KSP-SPR algorithm, a case study was carried out. The case study results showed that the introduced generalized spur path reuse technique saved more than 98% of spur path calculations for all test networks. The A* algorithm and the label initialization avoidance technique significantly sped up candidate path calculations (by 3.2 times and 9.7%, respectively) on the Wuhan network. The proposed KSP-SPR algorithm ran consistently faster than the improved Yen's algorithm (Yen, 1971; Hershberger et al., 2007) under various K values for all test networks, e.g., approximately 98.7 times on the Wuhan network.

Several further studies are worth noting. First, link costs in this study were assumed to be time-invariant and deterministic. Link travel times in real road networks, however, are time-dependent and uncertain. Further investigation into extending the proposed algorithm to dynamic stochastic networks is thus warranted (Chen et al., 2016; Chen et al., 2013; Yang and Zhou, 2017; Androutsopoulos et al., 2008). Second, the proposed algorithm only considers KSP findings in road networks. How to extend KSPs to public transport networks is a topic for further study (Khani et al., 2015; Huang, 2007). Finally, KSP algorithms are commonly used to solve complex network optimization problems with multiple objectives and/or constraints. Further studies are therefore required to apply the proposed algorithm to solve such optimization problems in large road networks (Reinhardt and Pisinger, 2011; Raith and Ehrgott, 2009).

**Acknowledgements**

The work described in this paper was supported by research grants from the National Key Research and Development Program (No. 2017YFB0503600), the National Natural Science Foundation of Hubei Province (No. 2020CFA054), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. R5029-18), and the Research Committee of the Hong Kong Polytechnic University (No. 4-ZZFY).

**Appendix**

**Proposition A1**. The proposed KSP-SPR algorithm can exactly solve the problem of finding k shortest simple paths.

**Proof.** The proposed KSP-SPR algorithm follows the generic procedure of Yen's algorithm, which has been proved to determine the optimal solution (Yen, 1971). The KSP-SPR algorithm makes two modifications to Yen's algorithm. Firstly, the KSP-SPR algorithm employs the *backward one-to-all algorithm* instead of *the forward one-to-one algorithm* on Line 01 of Table 1. Clearly, both algorithms can determine the optimal first shortest path. Secondly, the KSP-SPR algorithm employs *FindSpurPath-SPR* procedure instead of *FindSpurPath-Yen* procedure on Line 05 of Table 1. Therefore, the proof of the optimality of the KSP-SPR algorithm is equivalent to prove that the *FindSpurPath-SPR* procedure can determine the optimal spur path $\bar{s}_i^j$ as below.

The *FindSpurPath-SPR* procedure classifies the $\bar{s}_i^j$ determination into $|\tilde{A}_i^j| = 1$ and $|\tilde{A}_i^j| > 1$ scenarios. For $|\tilde{A}_i^j| = 1$ scenario, the procedure firstly call *RetrieveCandidatePath* sub-procedure to retrieve candidate path $q_{iv}^w$. Let $\mathrm{N}_{iv}^w$ be a node set containing the node $n_{iv}^w \in T_{iv}$ storing $q_{iv}^w$ and all its parent nodes in $T_{iv}$. According to sub-procedure, the retrieved $q_{iv}^w$ can satisfy either $q_{iv}^w \cap \tilde{N}_i^j = \emptyset$ and $\tilde{N}_i^j \supset \mathrm{N}_{iv}^w$ or $\tilde{N}_i^j \neq \emptyset$ and $\tilde{N}_i^j \supset \mathrm{N}_{iv}^w$. If $q_{iv}^w \cap \tilde{N}_i^j = \emptyset$ and $\tilde{N}_i^j \supset \mathrm{N}_{iv}^w$ hold, the retrieved $q_{iv}^w$ can be identified as the spur path $\bar{s}_i^j$ according to Proposition 2. Otherwise, the retrieved $q_{iv}^w$ is used as the input of initial candidate path $q_{iv}^u$ to call *CalculateCandidatePaths* sub-procedure. This sub-procedure incrementally removes nodes from $\tilde{N}_i^j$ to calculate candidate path $q_{iv}^w$ until $q_{iv}^w \cap \tilde{N}_i^j = \emptyset$ holds. According to the Proposition 2, the calculated $q_{iv}^w$ can be identified as the spur path $\bar{s}_i^j$. Therefore, the *FindSpurPath-SPR* procedure can determine the optimal spur path $\bar{s}_i^j$ for $|\tilde{A}_i^j| = 1$ scenario. For $|\tilde{A}_i^j| > 1$ scenario, the procedure firstly retrieve $q_{iv}^0$ using the GetCandidatePath-V&F procedure. If $q_{iv}^0 \cap \tilde{N}_i^j = \emptyset$ hold, $q_{iv}^0$ can the optimal $\bar{s}_i^j$. Otherwise, the A* algorithm is used to determine optimal $\bar{s}_i^j$. Therefore, The *FindSpurPath-SPR* procedure can determine the optimal spur path $\bar{s}_i^j$ for both $|\tilde{A}_i^j| = 1$ and $|\tilde{A}_i^j| > 1$ scenarios. □

**References**

Androutsopoulos, K.N. and Zografos, K.G., 2008, Solving the k-shortest path problem with time windows in a time varying network. *Operations Research Letters*, 36, pp. 692-695.

Bellman, R., 1958, On a routing problem. *Quarterly of Applied Mathematics*, 16, pp. 87-90.

Chen, B.Y., Lam, W.H.K., Sumalee, A., Li, Q.Q., Shao, H. and Fang, Z.X., 2013, Finding reliable shortest paths in road networks under uncertainty. *Networks & Spatial Economics*, 13, pp.

123-148.
Chen, B.Y., Yuan, H., Li, Q.Q., Lam, W.H.K., Shaw, S.-L. and Yan, K., 2014, Map matching algorithm for large-scale low-frequency floating car data. *International Journal of Geographical Information Science*, 28, pp. 22-38.
Chen, B.Y., Li, Q.Q. and Lam, W.H.K., 2016, Finding the k reliable shortest paths under travel time uncertainty. *Transportation Research Part B*, 94, pp. 189-203.
Chen, B.Y., Chen, X.-W., Chen, H.-P. and Lam, W.H.K., 2020, Efficient algorithm for finding K shortest paths based on re-optimization technique. *Transportation Research Part E*, 133, p. 101819.
Clarke, S., Krikorian, A. and Rausen, J. 1963. Computing the N Best Loopless Paths in a Network. *Journal of the Society for Industrial & Applied Mathematics,* 11(4), 1096-1102.
Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik,* 1(1), 269-271.
Eppstein, D. 1998. Finding the k shortest paths. *SIAM Journal on Computing,* 28(2), 652-673.
Fredman, M.L. and Tarjan, R.E., 1987, Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34, pp. 596-615.
Fu, L., Sun, D. and Rilett, L.R., 2006, Heuristic shortest path algorithms for transportation applications: State of the art. *Computers & Operations Research*, 33, pp. 3324-3343.
Hershberger, J., Maxel, M. and Suri, S., 2007, Finding the k shortest simple paths: A new algorithm and its implementation. *ACM Transactions on Algorithms*, 3, p. 45.
Hoffman, W. and Pavley, R. 1959. A method for the solution of the nth best path problem. *Journal of the ACM,* 6(4), 506-514.
Huang, R.H., 2007, A schedule-based pathfinding algorithm for transit networks using pattern first search. *Geoinformatica*, 11, pp. 269-285.
Khani, A., Hickman, M. and Noh, H., 2015, Trip-based path algorithms using the transit network hierarchy. *Networks and Spatial Economics*, 15, pp. 635-653.
Lawler, E.L., 1972, A procedure for computing the K best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18, pp. 401-405.
Liu, F., Andrienko, G., Andrienko, N., Chen, S., Janssens, D., Wets, G. and Theodoridis, Y., 2020, Citywide traffic analysis based on the combination of visual and analytic approaches. *Journal of Geovisualization and Spatial Analysis*, 4, p. 15.
Martins, E.Q.K., 1984, An algorithm for ranking paths that may contain cycles. *European Journal of Operational Research*, 18, pp. 123-130.
Martins, E. Q. V. and Pascoal, M. M. B. 2003. A new implementation of Yen's ranking loopless paths algorithm. *4OR,* 1(2), 121-133.
Minieka, E., 1974, On computing sets of shortest paths in a graph. *Communications of the ACM*, 17, pp. 351-353.
Raith, A. and Ehrgott, M., 2009, A comparison of solution strategies for biobjective shortest path problems. *Computers & Operations Research*, 36, pp. 1299-1331.
Reinhardt, L.B. and Pisinger, D., 2011, Multi-objective and multi-constrained non-additive shortest path problems. *Computers & Operations Research*, 38, pp. 605-616.
Roddity, L. and Zwick, U., 2005, Replacement paths and k simple shortest paths in unweighted directed graphs. In: *Proceedings of the International Conference on Automata, Languages and Programming (ICALP)*, pp. 249-260.
Sester, M., 2020, Analysis of mobility data: A focus on mobile mapping systems. *Geo-spatial Information Science*, 23, pp. 68-74.
Vanhove, S. and Fack, V. 2012. An effective heuristic for computing many shortest path alternatives in road networks. *International Journal of Geographical Information Science,* 26(6), 1031-1050.
Yang, L. and Zhou, X., 2017, Optimizing on-time arrival probability and percentile travel time for

elementary path finding in time-dependent transportation networks: Linear mixed integer programming reformulations. *Transportation Research Part B*, 96, pp. 68-91.

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

Zeng, W. and Church, R.L., 2009, Finding shortest paths on real road networks: the case for A*. *International Journal of Geographical Information Science*, 23, pp. 531-543.