Transportation Research Part E 133 (2020) 101819

Contents lists available at ScienceDirect

Transportation Research Part E

journal homepage: www.elsevier.com/locate/tre

Efficient algorithm for finding k shortest paths based on re-optimization technique

Bi Yu Chen[a,b,c], Xiao-Wei Chen[a,b], Hui-Ping Chen[a,b,c,*], 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, Hong Kong

ARTICLE INFO

Keywords:
K shortest path problem
Re-optimization technique
Lifelong planning A*

ABSTRACT

This study proposes an efficient deviation path algorithm for finding exactly k shortest simple paths without loops in road networks. The algorithm formulates the deviation path calculation process as repeated one-to-one searches for the shortest path in a dynamic network, where only a node and a link are restored at each search. Using this formulation, the proposed algorithm maintains and updates a single shortest path tree rooted at the destination. A re-optimization technique, lifelong planning A*, is incorporated into the algorithm to efficiently calculate each deviation path by reusing the shortest path tree generated at the previous search. To verify the efficiency of the proposed algorithm, computational experiments were conducted using several real road networks, and the results showed that the proposed algorithm performed significantly better than state-of-the-art algorithms.

1. Introduction

K shortest path (KSP) problems involve finding the shortest path, the second shortest path, and so on to the k^{th} shortest path between a given origin and a destination (O–D) pair. KSP problems have been the subject of intensive research for a half century (Clarke et al., 1963; Fox, 1973; Eppstein, 1998; Hershberger et al., 2007; Vanhove and Fack, 2012; Chen et al., 2016; Ye et al., 2018; Yu et al., 2019) owing to the breadth of their applications. For example, KSPs are usually provided in route guidance systems to satisfy various preferences that different users have for path choices (Vanhove and Fack, 2012). In addition, KSPs are often generated for approximating complicated network optimization problems with multiple objectives (Raith and Ehrgott, 2009) and/or constraints (Pugliese and Guerriero, 2013; Srinivasan et al., 2014). Recently, KSPs are also utilized for transport big data mining and complex network analysis (Kuijpers et al., 2016; Yue et al., 2019). Therefore, it is necessary to develop efficient algorithms for finding KSPs in real large-scale transport networks.

In light of work in the literature, the KSP problems can be classified into two variants: k shortest simple path problems, and k shortest non-simple path problems. In the first variant, the paths are restricted to being simple and without loops, i.e., no node can be repeated. Paths in the second variant can be non-simple paths with loops. Compared with the non-simple path variant, the simple path variant is more frequently applicable to empirical scenarios (e.g. route guidance systems), and is significantly more challenging due to the additional “loopless” constraint. This study focuses on the simple path variant.

A considerable amount of research effort has been devoted to developing KSP algorithms for exactly finding k shortest simple

* Corresponding author at: State Key Laboratory of Information Engineering in Surveying, Mapping and Remote Sensing, Wuhan University, 129 Luoyu Road, Wuhan 430079, China.
E-mail address: chenhuiping.gg@gmail.com (H.-P. Chen).

https://doi.org/10.1016/j.tre.2019.11.013
Received 27 May 2019; Received in revised form 20 November 2019; Accepted 26 November 2019
Available online 11 December 2019
1366-5545/

paths between a given O-D pair. Most KSP algorithms are based on the concept of the deviation path originally proposed by Yen (1971), based on the observation that the i^{th} shortest path always deviates at a certain node (called the deviation node) from a path previously determined. KSP algorithms are more complex than the well-known shortest path algorithm of Dijkstra (1959), because they typically perform many shortest path searches to calculate deviation paths for each previously determined path (Vanhove and Fack, 2012). Yen's original algorithm (Yen, 1971) performs $O(|N|)$ one-to-all shortest path computations to calculate the deviation paths of each previously determined path, where $|N|$ is the number of nodes in the road network. Using the F-Heap data structure (Fredman and Tarjan 1987), the one-to-all Dijkstra (1959) algorithm can achieve its best worst-case complexity, $O(|A| + |N| \log |N|)$, where $|A|$ is the number of links in the road network. Accordingly, Yen's original algorithm has complexity $O(k\ |N|(|A| + |N| \log |N|))$, which is the best worst-case complexity for finding k shortest simple paths.

Although the bound of the worst-case complexity of Yen's algorithm has been unparalleled for the past 40 years, there is considerable interest in developing efficient algorithms to improve his implementation (Yen, 1971). Because the one-to-all Dijkstra algorithm can lead to computational overheads in large networks, researchers (Hershberger et al., 2007) suggested to use one-to-one Dijkstra's algorithm with state-of-the-art data structures, e.g., the F-Heap (Fredman and Tarjan, 1987). This improved version is hereinafter called the improved Yen's algorithm to distinguish it from Yen's original algorithm. Martins and Pascoal (2003) proposed an efficient KSP algorithm, called M-P algorithm in this study for convenience. In the M-P algorithm, the deviation paths are computed in a reverse order, from the destination to the deviation nodes, so that a single shortest path tree rooted at the destination is generated and updated. It formulates the deviation path calculation process as a problem of one-to-all shortest path searches in a dynamic updating network, where only a node and a link are restored in every search. Using this formulation, the M-P algorithm reuses the results of the shortest path tree in the previous search, and thus has a significant computational advantage over Yen's original algorithm. However, the M-P algorithm requires updating the entire shortest path tree (i.e., using the one-to-all Dijkstra's algorithm) when calculating each deviation path, and this leads to computational overhead in large networks. Recent studies (Vanhove and Fack, 2012) have found that the M-P algorithm is in most cases slower than the improved Yen's algorithm. Vanhove and Fack (2012) proposed another exact KSP algorithm, in which the backward one-to-all Dijkstra's algorithm is used to pre-calculate shortest paths from all nodes to the destination. During the calculation of each deviation path, the algorithm uses the pre-calculated shortest path as sub-path of the deviation path if the latter does not have cycles.

The performance of KSP algorithms relies heavily on that of the shortest path algorithms used to calculate the deviation paths. Numerous heuristic techniques offer an effective means of speeding-up one-to-one shortest path calculations (Fu et al., 2006; Li et al., 2015; Qian, et al., 2011). A popular technique of this sort is the A* technique (Hart et al., 1968). It adopts a heuristic function to assign higher priorities to nodes closer to the destination, thereby reducing the number of visited nodes far from the destination. Empirical research (Zeng and Church, 2009) has revealed that the A* technique using Euclidean distance as heuristic function can reduce computation time by 30–50% in road networks compared with its Dijkstra algorithm-based counterpart. As an extension of the A* technique, Koenig et al. (2004) proposed a lifelong planning A* (LPA*) technique for dynamic updating networks in which traffic conditions are updated in real time. The proposed LPA* technique is a re-optimization approach that reuses the results of the shortest path tree generated in the previous search to enhance the efficiency of shortest path computations. The LPA* technique can achieve the optimal solution when the heuristic function satisfies the triangle inequality. Huang et al. (2007) found that the LPA* technique can improve the efficiency of the A* algorithm by approximately 13–30% on road networks, and its advantage becomes clear when fewer network links update their costs. Although LPA* is an effective re-optimization technique for speeding-up shortest path computations in dynamic updating networks, it has rarely been employed to develop efficient KSP algorithms.

By using an additional preprocessing phase, several advanced techniques recently have been developed for speeding-up subsequent phase of shortest path calculations. They include ALT (A*, landmarks, and triangle inequality) (Goldberg and Harrelson, 2005), transit node routing (Bast et al., 2007), arc flags (Hilger et al., 2009), contraction hierarchies (Geisberger et al., 2012), customizable route planning (Delling et al., 2017) and etc. (see Bast et al. (2014) for a comprehensive review). In these techniques, the preprocessing phase typically takes several minutes (or even hours) in large-scale road networks to produce auxiliary data, which are then used to calculate the shortest path in a millisecond or less. Due to their efficiency, such advanced techniques have found their way into commercial routing systems, e.g., Google Maps, Bing Maps and TomTom (Bast et al., 2014). However, it remains challenge for these techniques to speed-up shortest path calculations in dynamic updating networks, because the auxiliary data have to be reproduced or updated leading to a considerable computational burden. Therefore, such techniques are not applicable for developing efficient KSP algorithms with numerous deviation path calculations in the dynamic updating network.

This study proposes an efficient KSP algorithm based on the LPA* technique. Inspired by the M-P algorithm, the proposed algorithm maintains and updates a single shortest path tree rooted at the destination in reverse order. Unlike the one-to-all shortest path searches used in the M-P algorithm, the KSP-LPA* algorithm formulates the deviation path calculation process as a problem of one-to-one shortest path searches in the dynamic updating network. The LPA* technique is incorporated to improve the efficiency of deviation path calculation by reusing the result of the previous one-to-one search. The proposed algorithm can thus avoid the computational overhead of the M-P algorithm, incurred because the entire shortest path tree needs to be updated for the calculation of every deviation path. To verify the effectiveness and efficiency of the proposed algorithm, computational experiments using several road networks were carried out. Four state-of-the-art algorithms for exactly calculating k shortest simple paths (Yen, 1971; Martins and Pascoal, 2003; Hershberger et al., 2007; Vanhove and Fack, 2012) were implemented for comparison with the proposed one. The results show that the proposed algorithm is significantly faster than the other KSP algorithms on all tested networks under various values of k.

The remainder of this paper is organized as follows: Section 2 briefly introduces conventional deviation path algorithms to provide the necessary background, and Section 3 describes the proposed KSP-LPA* algorithm. Section 4 reports computational

experiments using road networks to test the proposed method, and Section 5 contains the conclusions of this study together with recommendations for future research in the area.

2. Conventional deviation path algorithms

2.1. General principle

Consider a directed graph G(N, A) consisting of a set of nodes N and a set of links A. Each link $a = (n_i, n_j) \in A$ has a tail node n_i, a head node n_j, and a non-negative link cost $t(a)$, which typically represents the travel time or length of the link. Let $o \in N$ and $d \in N$ be the origin and destination nodes, respectively. Each node $n_i \in N$ has a set of successor nodes SUCC(n_i) and a set of predecessor nodes PRED(n_i).

Let p_{od}^u (or p^u for short) be a path between the O–D pair, consisting of l nodes ($o = n_1^u, \cdots, n_l^u = d$) and $l-1$ links ($a_1^u, \cdots, a_{l-1}^u$). The cost of traversing the path can be calculated by the summation of the cost of its corresponding links as:

$$t(p^u) = \sum_{i=1}^{l-1} t(a_i^u) \tag{1}$$

p^u is said be a simple path if all nodes along it are distinct; otherwise, it is a non-simple path with cycles. Let P be a path set consisting of all simple paths between the same O–D pair. The problem of finding k shortest simple paths can then be defined as below.

Definition 1. Given an integer $k \geq 1$, *the problem is to find the set of k shortest simple paths, denoted by* $P^k = \{p^1, \cdots, p^k\}$, satisfying:

(1) $t(p^i) \leq t(p^{i+1})$ *for* $\forall\, p^i \in P^k$;
(2) $t(p^k) \leq t(p^u)$, $\forall\, p^u \in P - P^k$.

Most algorithms for finding k shortest simple paths are based on the concept of the deviation path. Fig. 1 illustrates this concept through a simple example. The shortest path $p^1 = \{n_1^1, \cdots, n_4^1\}$ comprises four nodes. Let D^1 be the set of deviation paths of p^1. There are three possible deviation paths, $\{\bar{p}_1^1, \bar{p}_2^1, \bar{p}_3^1\} \in D^1$. Each deviation path $\bar{p}_i^1$ coincides with the shortest path for the first i nodes, deviates to a different sub-path, and finally reaches the destination. The i^{th} node is called the deviation node, n_i^1 (*e.g.*, node 2), the i^{th} link is called the deviation link, a_i^1, the sub-path from the first node to the deviation node is called the root path, $\bar{r}_i^1 = \{n_1^1, \cdots, n_i^1\}$, and the remaining sub-path from the deviation node to the destination is called the spur path, $\bar{s}_i^1$. The deviation path is the concatenation of the root and spur paths, $\bar{p}_i^1 = \bar{r}_i^1 \oplus \bar{s}_i^1$, where $\oplus$ is the path concatenation operator. To calculate the spur path $\bar{s}_i^1$, any shortest path algorithm, *e.g., one-to-one Dijkstra's algorithm, can be applied to a modified network* G_i^1, where the deviation link a_i^1 and all nodes preceding n_i^1 in p^1 are removed from the network G. The network is modified to guarantee a spur path without cycles different from the corresponding sub-path of p^1. After all deviation paths in D^1 have been calculated, the second shortest simple path, p^2, can be determined as the deviation path, $\bar{p}_i^1 \in D^1$, at the minimum cost.

Analogously, we can determine the $j+1^{th}$ ($\forall\, j = 1, \cdots, k-1$) shortest simple path as one of the deviation paths of the j shortest simple paths that have been determined. Compared with the case of p^1, the first deviation node of the j^{th} shortest simple path $p^j = \{n_1^j, \cdots, n_m^j, \ldots, n_l^j\}$ may not be its first node. As shown in Fig. 2, the first deviation node n_m^j (e.g., node 2 of p^3) is defined as the last node of the longest sub-path (e.g., nodes 1 and 2) coinciding with the $j-1$ shortest simple paths that have been determined (e.g., p^1 and p^2). Consequently, only $l-m$ (e.g., 2) deviation paths $\{\bar{p}_m^j, \ldots, \bar{p}_{l-1}^j\} \in D^j$ are computed for the j^{th} shortest simple path. The calculation of deviation paths for nodes preceding n_m^j (e.g., Node 1) can be omitted because these paths have been calculated for the $j-1$ shortest simple paths. To calculate the spur path $\bar{s}_m^j$ for n_m^j, the possible deviation links on p^j and the $j-1$ paths (e.g., a_2^3 and a_2^1) should be removed from the road network.

Using the above concept of the deviation path, most deviation path algorithms can be described according to the generic procedure in Table 1. Deviation path algorithms maintain two sorted collections of simple paths: a determined path collection L

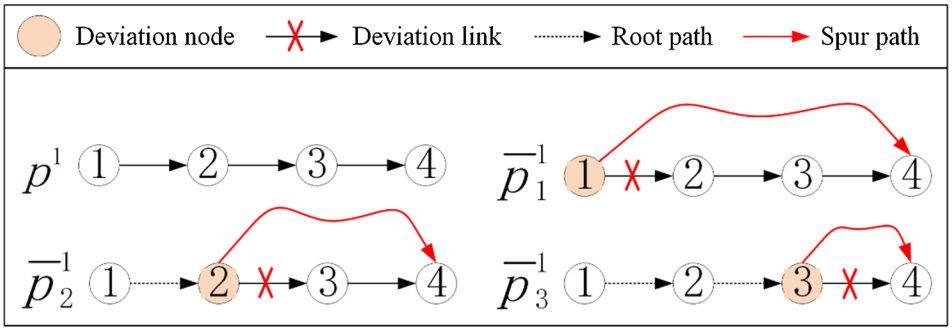

Fig. 1. An illustrative example of the concept of the deviation path for the first shortest path.

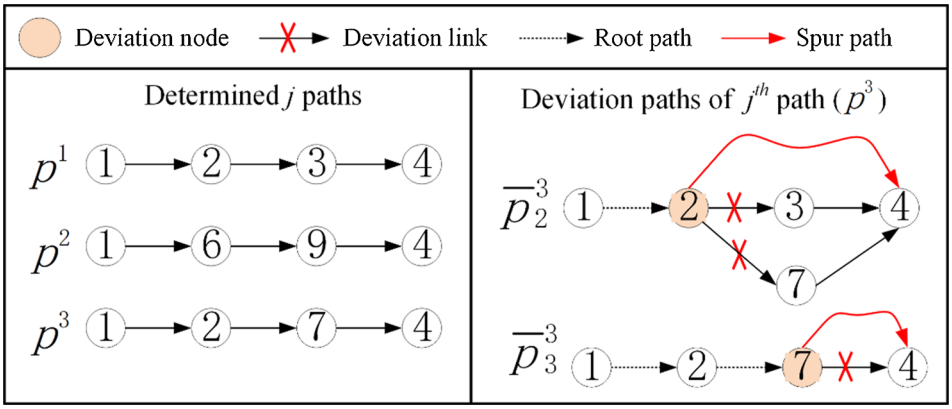

Fig. 2. An example of the concept of the deviation path for the j^{th} shortest path.

Table 1
Generic procedure of deviation path algorithms.

Input: *O-D* pair, *k*
Return: Determined path collectionL
01: Calculate the first shortest path p^1 between the O-D pair.
02: Set candidate path collection C: ={p^1} and set determined path collection L: =∅.
03: For $j := 1$ to k
04: If C = ∅, Then stop and return L.
05: Set p^j as the path at the top of C; and remove p^j from C.
06: Add p^j into L.
07: Call *CalDevPaths* Procedure to calculate deviation path set D^j. (algorithms different here)
08: Set C: =C ∪ D^j.
09: End for
10: Return L.

containing the shortest simple paths that have been found so far, and a candidate path collection C containing the calculated deviation paths. C is implemented by the priority queue structure, e.g., the Fibonacci heap (Fredman and Tarjan, 1987). Initially, the first shortest path, p^1, between the O–D pair is calculated and added to C. To determine the j^{th} shortest path p^j, the path with the minimum cost (at the top of C) is fetched and removed from C and added to L. Then, the set of deviation paths D^j is calculated and added to C. Deviation path algorithms continue the shortest path determination and deviation path calculation process until k shortest simple paths have been determined. Note that there may be fewer than k shortest simple paths between the O–D pair if C is empty. Although deviation path algorithms have the same generic procedure, they are different in the ways they calculate deviation paths in Line 07.

2.2. Yen's algorithm

Yen's algorithm (Yen, 1971) is a classical deviation path algorithm that follows the generic procedure shown in Table 1. It uses the *CalDevPaths-Yen* procedure given in Table 2 to calculate deviation paths. Initially, Yen's algorithm calls the *FindFirstDevNode* procedure to find the longest sub-path coinciding with the $j-1$ shortest simple paths that have been determined within collection L to find the first deviation node n_m^j and the set of deviation links E_m^j at the deviation node n_m^j. This procedure can be efficiently implemented by representing L as a deviation tree structure (Roddity and Zwick, 2005), in which n_m^j and E_m^j can be looked up quickly. Yen's algorithm calculates deviation paths in a forward order from n_m^j to n_{l-1}^j, and uses the forward Dijkstra's algorithm to calculate the spur path from each deviation node n_i^j to the destination. For each n_i^j, all nodes (n_1^j,.., n_{i-1}^j) preceding it as well as its deviation link(s) are removed from network G. Then, spur path $\bar{s}_i^j$ can be calculated in the modified network by using the forward Dijkstra's algorithm. The deviation path $\bar{p}_i^j$ is determined as the concatenation of $\bar{s}_i^j$ and $\bar{r}_i^j$. The procedure terminates after all deviation paths in D^j have been determined.

Note that Yen's original algorithm (Yen, 1971) adopts the forward one-to-all Dijkstra's algorithm to calculate spur paths. The improved Yen's algorithm (Hershberger et al., 2007) uses the forward one-to-one Dijkstra's algorithm to improve the calculation of the spur path. Both implementations of Yen's algorithm have the same worst-case computational complexity of $O(k\,|N|(|A| + |N| \log |N|))$, which is the best worst-case complexity for finding k shortest simple paths in road networks.

Table 2
Deviation path calculation procedure in Yen's algorithm.

Procedure: ***CalDevPaths-Yen***
Input: Network G, the j^{th} shortest simple path p^j, the determined path collectionL
Return: Deviation path setD^j
01: Call *FindFristDevNode*(p^j, L) to determine the first deviation node n_m^j of p^j and the associated deviation link set E_m^j at n_m^j.
02: For i: =1 to $m-1$
03: Remove n_i^j from G.
04: End for
05: Remove all deviation links in E_m^j from G.
06: For$i$: =m to $l-1$ (l is the number of links of p^j)
07: Remove deviation link a_i^j: =(n_i^j, n_{i+1}^j) from G.
08: Set root path $\bar{r}_i^j$: =($n_1^j, \cdots, n_i^j$).
09: Calculate spur path $\bar{s}_i^j$ using forward one-to-all (or one-to-one) Dijkstra's algorithm.
10: Determine deviation path $\bar{p}_i^j := \bar{r}_i^j \oplus \bar{s}_i^j$.
11: Add $\bar{p}_i^j$ into D^j.
12: Remove n_i^j from G.
13: End for
14: Restore network G.
15: Return D^j.

2.3. The M-P algorithm

Martins and Pascoal (2003) proposed a deviation path algorithm called M-P algorithm in this study. Unlike Yen's algorithm, the M-P algorithm calculates deviation paths in a reverse order from the destination to the deviation nodes (see CalDevPaths-MP procedure in Table 3). It first generates a shortest path tree rooted at the destination by using the backward one-to-all Dijkstra's algorithm (see Line 06). In each subsequent calculation of the deviation path, only one node, n_i^j, and one link, a_{i+1}^j, are restored in the network. With these changes, the M-P algorithm calculates the spur path $\bar{s}_i^j$ by maintaining and updating a single shortest path, rather than performing a one-to-all shortest path search from scratch. The entire shortest path tree is updated by correcting all affected shortest paths (see FindSpurPath-MP procedure in Table A1). This re-optimization process is implemented by the *BackwardStar* procedure shown in Table A1, which is an incremental version of the backward one-to-all Dijkstra's algorithm without label initialization.

Although the M-P algorithm has a computational advantage over Yen's original algorithm, it requires updating the entire shortest path tree in each calculation of the deviation path, and this can lead to a significant computational overhead. Recent studies (Vanhove and Fack, 2012) have shown that the M-P algorithm runs even slower in most cases than the improved Yen's algorithm, which directly uses the one-to-one Dijkstra's algorithm to calculate every deviation path from scratch. To address this problem, the KSP-LPA* algorithm is proposed and described in the next section.

Table 3
Deviation path calculation procedure in the M-P algorithm.

Procedure: ***CalDevPaths-MP***
Input: Network G, the j^{th} shortest simple path p^j, the determined path collectionL
Return: Deviation path setD^j
01: Call *FindFristDevNode*(p^j, L) to determine the first deviation node n_m^j of p^j and the associated deviation link set E_m^j at n_m^j.
02: For i: =1 to $l-1$ (l is the number of links of p^j)
03: Remove node n_i^j and link a_i^j from G.
04: End for
05: Remove all links in E_m^j from G.
06: Call the backward one-to-all Dijkstra's algorithm to calculate the shortest path $p_{n_v d}$ from every node n_v to destination d.
07: For$i$: =$l-1$ to$m$
08: Restore node n_i^j to G.
09: Set root path $\bar{r}_i^j$: =($n_1^j, \cdots, n_i^j$).
10: Call *FindSpurPath-MP*(n_i^j, a_i^j) to calculate $\bar{s}_i^j$.
11: Determine deviation path $\bar{p}_i^j := \bar{r}_i^j \oplus \bar{s}_i^j$.
12: Add $\bar{p}_i^j$ into D^j.
13: End for
14: Restore network G.
15: Return D^j.

Table 4
Deviation path calculation procedure in the KSP-LPA* algorithm.

Procedure: ***CalDevPaths-LPA****
Input: Network G, the j^{th} shortest simple path p^j, the determined path collectionL
Return: Deviation path setD^j
01: Call *FindFristDevNode*(p^j, L) to determine the first deviation node n_m^j of p^j and the associated deviation link set E_m^j at n_m^j.
02: For i: =1 to$l-1$
03: Remove node n_i^j and link a_i^j from G.
04: End for
05: Remove all links in E_m^j from G.
06: Set *isFirstSearch* := True, and set A_i^j: ={∅}.
07: For i: =$l-1$ to$m$
08: Restore n_i^j to G, and set N_i^j: ={n_i^j}.
09: Set root path $\bar{r}_i^j$: =($n_1^j, \cdots, n_i^j$).
10: Call *FindSpurPathLPA*(*isFirstSearch*, N_i^j, A_i^j, n_i^j, n_l^j) to calculate $\bar{s}_i^j$.
11: Determine deviation path $\bar{p}_i^j := \bar{r}_i^j \oplus \bar{s}_i^j$.
12: Add $\bar{p}_i^j$ into D^j.
13: Restore a_i^j to G, and set A_i^j: ={a_i^j}.
14: End for
15: Restore network G.
16: Return D^j.

3. Proposed k shortest path algorithm based on lifelong planning A* technique

This section introduces the proposed KSP-LPA* algorithm for exactly finding k shortest simple paths between a given O–D pair. The proposed algorithm follows the generic procedure shown in Table 1 but uses the *CalDevPaths-LPA** procedure (see Table 4) to calculate deviation paths. Similarly to the above *CalDevPaths-MP* procedure (see Table 3), it calculates spur paths by updating a single shortest path tree rooted at the destination in reverse order from the destination to the deviation nodes. These two procedures, however, are different at Lines 06 and 10, where the *FindSpurPath-LPA** procedure is introduced to calculate spur paths. It formulates spur path calculation process as the problem of one-to-one shortest path searches in a dynamic network, where only one link and one node are restored in each search. Then, a re-optimization technique, called lifelong planning A* (LPA*) (Koenig et al., 2004), is used to efficiently calculate the spur paths by reusing the results of the previous one-to-one shortest path search. In this way, the *FindSpurPath-LPA** procedure can avoid the computational overhead of the M-P algorithm.

The original LPA* technique (Koenig et al., 2004) uses forward search from the origin to the destination. During subsequent path searches, the origin node is fixed, and a heuristic function to estimate the distance to the destination is used to guide the shortest path search. To incorporate it into the *FindSpurPath-LPA** procedure, three major modifications are made. First, backward search from the destination to the deviation node is used instead of forward search. Second, the origin node (i.e., the deviation node) is changed at each subsequent shortest path search, rather than fixed. Third, the heuristic function is abandoned (i.e., set to zero) in this study. This is because a change in the origin node in each subsequent path search leads invalidates the heuristic function values generated in the previous search. Updating these values for all nodes in the network can introduce unnecessary computational overhead. The *FindSpurPath-LPA** procedure incorporated with the LPA* technique is introduced below.

Fig. 3 illustrates the effectiveness of the LPA* technique using a simple example shown in Fig. 2. Fig. 3(a) shows the cost of the corresponding link and the results of calculation of the first spur path $\bar{s}_3^3$. Unlike the backward one-to-one Dijkstra's algorithm, the LPA* technique for finding the shortest path maintains two values at each node n_u (given in the parentheses near the node in the figure). The first value, denoted by $g(n_u)$, is the calculated least cost of the path from n_u to the destination, and yields the same value as the backward Dijkstra's algorithm. The second value, denoted by $rhs(n_u)$, is the one-step look-back value defined as

$$rhs(n_u) = \begin{cases} 0, & n_u = d \\ \mathrm{Min}_{\forall n_v \in SUCC(n_u)}(g(n_v) + t(n_u, n_v)), & n_u \neq d \end{cases} \quad (2)$$

The minimum of these two values is used as the key to order nodes in the priority queue (also called scan eligible, denoted by SE):

$$key(n_u) = \mathrm{Min}\{g(n_u), rhs(n_u)\} \quad (3)$$

Initially, the *FindSpurPath-LPA** procedure sets these two value of all nodes to infinite, ∞. During shortest path calculation, the LPA* technique selects the node at the top of the SE with the minimum $key(n_u)$ for path extension until the key value of the selected node is larger than that of the deviation node. In the calculation of the first spur path without any change to the network, the relationship $g(n_u) \geq rhs(n_u)$ always holds for all nodes. Thus, the procedure for finding the first spur path is identical to the classical backward one-to-one Dijkstra's algorithm using $key(n_u) = rhs(n_u)$. $g(n_u) = rhs(n_u)$ holds for nodes (e.g. Nodes 4, 3, 7 and 9) selected from the SE.

For calculating the subsequent spur path $\bar{s}_i^j$(e.g., $\bar{s}_2^3$), the *FindSpurPath-LPA** procedure runs in an incremental way starting from

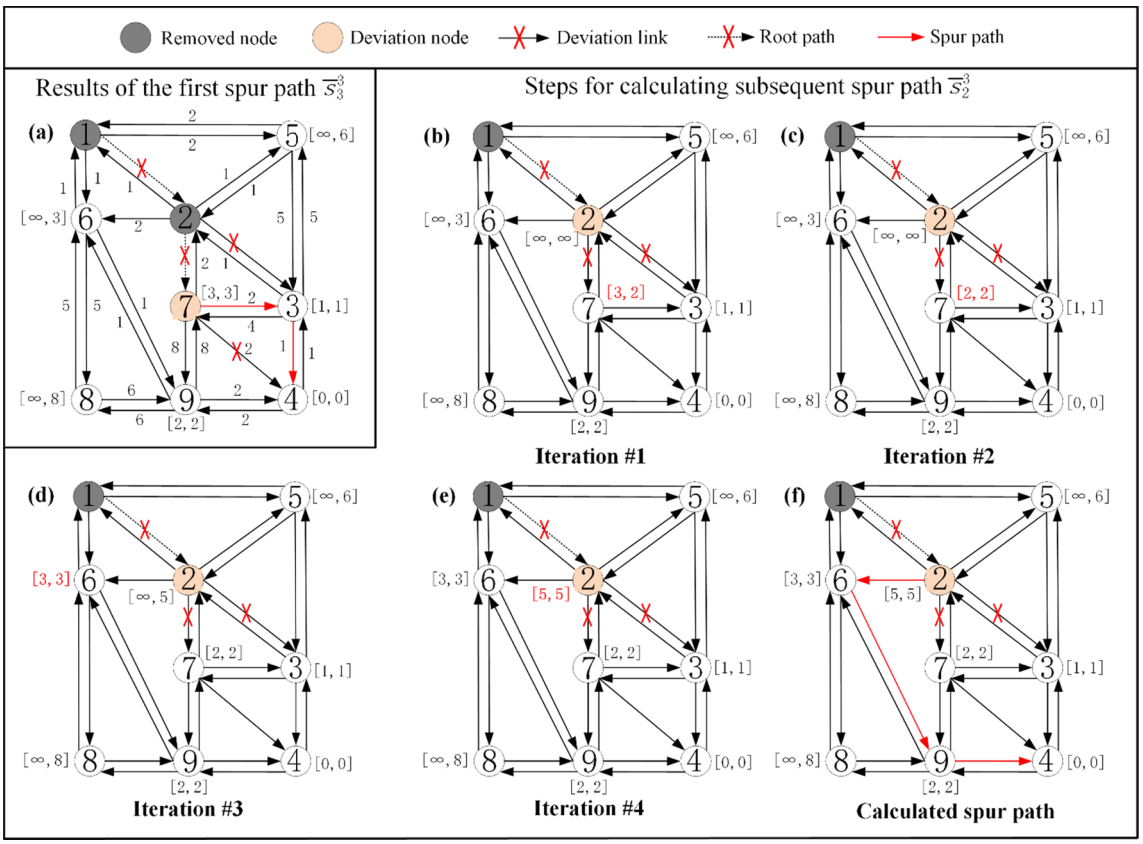

Fig. 3. An illustrative example of the *FindSpurPath-LPA** procedure.

previous results of spur path $\bar{s}_{i+1}^j$ (e.g., $\bar{s}_3^3$) calculation. In the subsequent calculation of the spur path, a deviation node n_i^j (e.g., Node 2) and a deviation link a_{i+1}^j (e.g., $a(7, 4)$) are restored into the network (e.g., Fig. 3(b)). These network changes can affect the $rhs(n_u)$ value of certain nodes (e.g., Node 7). Consequently, $g(n_u) \geq rhs(n_u)$ may not hold for some nodes in the network. In the original LPA* technique, there are three possible relationships between values of $g(n_u)$ and $rhs(n_u)$ for a node n_u:

(1) If $g(n_u) = rhs(n_u)$, then node n_u is a consistent node.
(2) If $g(n_u) > rhs(n_u)$, then node n_u is an over-inconsistent node.
(3) If $g(n_u) < rhs(n_u)$, then node n_u is an under-inconsistent node.

A node is said to be inconsistent when $g(n_u) \neq rhs(n_u)$, including both over-inconsistency, $g(n_u) > rhs(n_u)$, and under-inconsistency, $g(n_u) < rhs(n_u)$, scenarios. However, in the *FindSpurPath-LPA** procedure, only the over-inconsistent scenario is feasible because a node and a link are restored in each subsequent spur path search, leading to a reduction in values of $rhs(n_u)$. Using the LPA* technique, all consistent nodes can be ignored and only over-inconsistent nodes are further processed. If inconsistent nodes have been selected and removed from SE, they are re-inserted into SE (e.g., Node 7 in Iteration #1); otherwise they are re-ordered in SE based on their new key values. Similarly to the backward one-to-one Dijkstra's algorithm, in each iteration, the procedure selects the inconsistent node with the minimum key value at the top of the SE for further inconsistency correction (or path extension). For example, Nodes 7, 6, and 2 in the figure are selected in ascending order in Iterations #2, #3, and #4, respectively. The procedure terminates when the deviation node $n_i^j$is consistent and the key value of the selected node is larger than that of the deviation node n_i^j. In this way, the *FindSpurPath-LPA** procedure maintains and updates a single shortest path tree rooted at the same destination during subsequent spur path calculations. Note that the change in the deviation noden_i^j in the subsequent search does not affect the values of $g(n_u)$ and $rhs(n_u)$ of each node n_u.

Using the LPA* technique, the *FindSpurPath-LPA** procedure can reuse a considerable part of the shortest path tree in terms of consistent nodes generated in the previous spur path search, given that only a small part of the network (i.e., a node and a link) is changed in every subsequent search. Therefore, the LPA* technique can significantly speed-up the calculation of spur paths from scratch when used in the improved Yen's algorithm. Compared with the M-P algorithm, the LPA* technique can efficiently determine the spur path when the termination conditions are satisfied. Therefore, it can avoid the computational overhead of the M-P algorithm, where the entire shortest path tree needs to be updated by correcting all nodes to be consistent.

The detailed steps of the *FindSpurPath-LPA** procedure are described below (Table 5). In Step 1, different initialization approaches

Table 5
Detailed steps of the *FindSpurPath-LPA** procedure.

Procedure: *FindSpurPath-LPA**

Input: Flag *isFirstSearch,* updated node set N_i^j, updated link set A_i^j, node n_i^j, and destination d.
Return: Spur path $\bar{s}_i^j$.
Step 1. Initialization:
01: If *isFirstSearch* = True, Then (The first spur path search case)
02: For each node n_u inG
03: Set $g(n_u)$: =∞ and $rhs(n_u)$: =∞.
04: End For
05: Set $g(d)$: =0, $rhs(d)$: =0, and SE = {d}.
06: Set *isFirstSearch* := False.
07: Else (The subsequent spur path search case)
08: For each updated link$a(n_u, n_v) \in A_i^j$
09: Call *UpdateNode*(n_u, d).
10: End for
11: For each updated node$n_u \in N_i^j$
12: For each node$n_v \in$ PRED(n_u)
13: Call *UpdateNode*(n_v, d).
14: End For
15: End for
16: End If
Step 2. Inconsistent Node Selection:
17: If SE = ∅, Then stop and return ∅.
18: Select n_u as the top of SE, and remove n_u from SE.
19: If $key(n_u) \geq key(n_i^j)$ and $g(n_i^j) = rhs(n_i^j)$, Then stop and return path at n_i^j.
Step 3. Inconsistency Correction:
20: If $g(n_u) > rhs(n_u)$, Then (Over-inconsistent case)
21: Set $g(n_u)$: =$rhs(n_u)$.
22: For each$n_v \in$ PRED(n_u)
23: Call *UpdateNode*(n_v, d).
24: End For
25: End If
26: Go back to Step 2.
Procedure: *UpdateNode*
Input: node n_u and destination$d$
01: If $n_u \neq d$, Then set $rhs(n_u)$: =$\text{Min}_{n_v \in SUCC(n_u)}(g(n_v) + t(n_u, n_v))$.
02: If $n_u \in$ SE, Then remove n_u from SE.
03: If $rhs(n_u) \neq g(n_u)$, Then insert n_u into SE.

are used for the first and subsequent spur path searches according to the input *isFirstSearch*. This value is true, which indicates the first spur path search, and the procedure sets $g(n_u) = \infty$ and $rhs(n_u) = \infty$ for all nodes including destination d. Otherwise, indicating a subsequent spur path search, the procedure maintains the results of $g(n_u)$ and $rhs(n_u)$at all nodes generated in the last search, and calls the *UpdateNode* procedure to update their values of $rhs(n_u)$ for nodes because of changes to the network. The *UpdateNode* procedure inserts only inconsistent nodes, i.e., $rhs(n_u) \neq g(n_u)$, into the SE and ignores consistent nodes. An inconsistent node already in the SE, i.e., $n_u \in$ SE, should be reordered based on its updated $rhs(n_u)$ by removing and then re-inserting it into the SE. In Step 2, the inconsistent node n_u at the top of the SE is selected for further inconsistency correction. In Step 3, the procedure corrects the selected node and makes it consistent by simply correcting its $g(n_u)$ to $rhs(n_u)$ and calling the *UpdateNode* procedure to update adjacent nodes. The *FindSpurPath-LPA** procedure iterates between Steps 2 and 3 until termination in case the deviation node is consistent, i.e., $g(n_i^j) = rhs(n_i^j)$, and the key value of the selected node is larger than that of the deviation node, i.e., $key(n_u) \geq key(n_i^j)$. Note that if SE = ∅, the procedure can return $\bar{s}_i^j$=null to reflect that no spur path can be determined between deviation node n_i^j and destination d.

The proposed KSP-LPA* algorithm has the same worst-case complexity as Yen's algorithm, i.e., $O(\text{k } |N|(|A| + |N| \log |N|))$. The optimality of the *FindSpurPath-LPA** procedure and the KSP-LPA* algorithm are proved below.

Proposition 1. *The FindSpurPath-LPA* procedure can determine the optimal shortest path between the deviation node and the destination in every spur path calculation.*

Proof. See Proposition A1 in the Appendix.

Lemma 1. *The KSP-LPA* algorithm can exactly solve the problem of finding k shortest simple paths.*

Proof. See Lemma A1 in the Appendix.

Table 6
Basic characteristics of testing networks.

Network	Hong Kong RTIS	Chicago regional	Wuhan network	Beijing network	Shenzhen network
Number of nodes	1367	12,982	19,306	59,541	69,198
Number of links	3655	39,018	46,757	114,737	192,582

4. Computational experiments

This section reports the computational performance of the proposed KSP-LPA* algorithm using several road networks of different sizes. The KSP-LPA* algorithm was coded in C# and the priority queue was implemented using the F-heap data structure (Fredman and Tarjan 1987). For comparison, four state-of-the-art KSP algorithms were implemented: Yen's original algorithm (Yen, 1971), the improved Yen's algorithm (Hershberger et al., 2007), the M-P algorithm (Martins and Pascoal, 2003), and the V-F algorithm (Vanhove and Fack, 2012). Both the improved Yen's algorithm and the V-F algorithm employed the one-to-one Dijkstra's algorithm using the F-heap data structure, whereas both Yen's original algorithm and the M-P algorithm used the one-to-all Dijkstra's algorithm with the data structure of a linked list using the first-in-first-out principle. These algorithms were also coded in C#. All experiments were conducted on a desktop with a four-core Intel 2.6 GHz CPU (only a single processor was used) and 8 GB of RAM on a Windows 8 operating system.

Five road networks were used for the computational tests. Their details are described in Table 6. For each network, 100 O–D pairs were randomly generated and were used for all five algorithms. All algorithms produced identical results for the k shortest simple paths between every O–D pair.

Figs. 4 and 5 report the computational performance of all five testing algorithms on the Shenzhen network. Computational performance was evaluated in terms of computational times (denoted by t in seconds) and the total number of visited nodes (denoted by n in 10^6). The value was k was set to 100. The x-axis represents 100 random queries and the y-axis the computational performance using a logarithmic scale.

As shown in Fig. 4, Yen's original algorithm delivered the worst performance of the five KSP algorithms because it required computing a one-to-all shortest path search (using one-to-all Dijkstra's algorithm) to calculate every deviation path, which led to a significant computational overhead. The M-P algorithm ran approximately 30 times faster than Yen's original algorithm on the Shenzhen network. This improvement occurred because the M-P algorithm employs a re-optimization technique to speed-up deviation path calculation by reusing the results of the shortest path tree generated in the previous search. Fig. 5 shows that the number of nodes, n, visited by the M-P algorithm was approximately 1.2 times smaller than Yen's original algorithm on the Shenzhen network. However, the M-P algorithm requires updating the entire shortest path tree for every deviation path calculation. In most cases (see Fig. 5), this can lead to the exploration of many unnecessary nodes compared with the improved Yen's algorithm, which executed only a one-to-one shortest path search for every deviation path calculation. Consequently, as shown in Table 7, the average computational time, t, of the M-P algorithm was approximately 3.4 times (i.e., 2538.4/571.3-1) longer than that of the improved Yen's algorithm on the Shenzhen network.

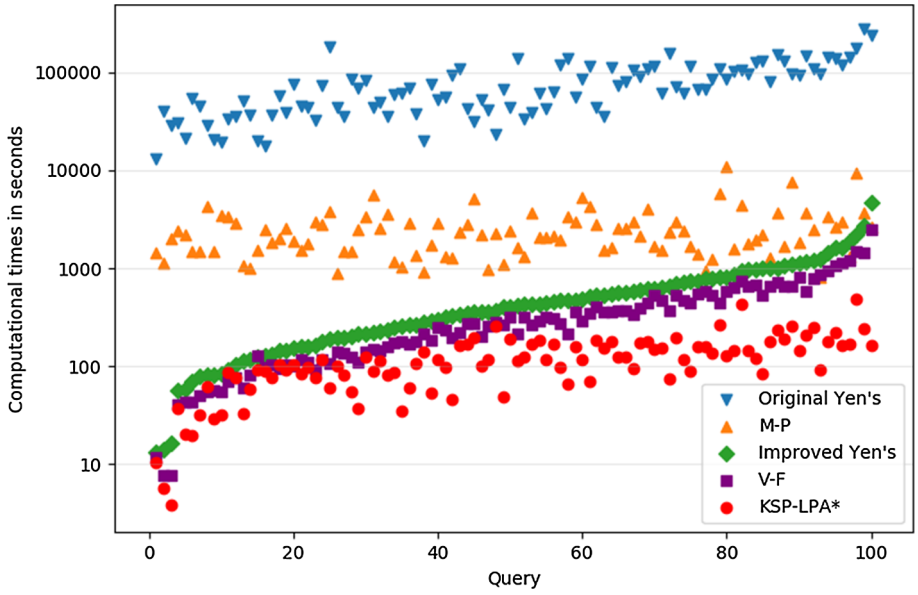

Fig. 4. Computational times (seconds) of all five algorithms on the Shenzhen network.

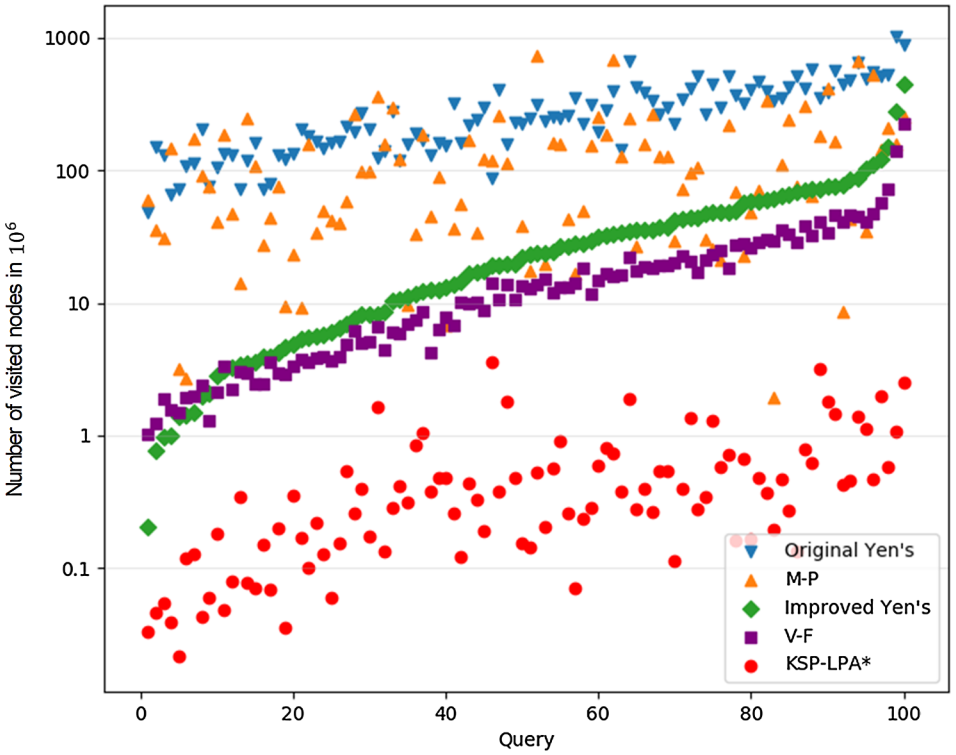

Fig. 5. Total number of visited nodes in all five algorithms on the Shenzhen network.

Table 7
The average computational performance of KSP algorithms in five road networks.

Network	Original Yen's		M-P		Improved Yen's		V-F		KSP-LPA*	
	t	n	t	n	t	n	t	n	t	n
Hong Kong RTIS	10.5	1.1	1.9	0.4	1.1	0.2	0.9	0.1	0.8	0.04
Chicago regional	1203.0	34.7	177.8	8.4	34.7	3.1	19.3	1.6	8.6	0.2
Wuhan network	850.6	33.6	185.8	9.3	96.0	4.2	81.5	3.0	37.7	0.2
Beijing network	57483.5	192.3	1643.6	96.7	306.2	19.2	224.8	12.0	93.4	0.4
Shenzhen network	76623.3	284.2	2538.4	128.9	571.3	37.2	372.2	19.2	164.1	0.5

t : Average computational times in seconds (100 runs)
n: Average numbers of visited nodes in 10^6 (100 runs)

Fig. 4 shows that the proposed KSP-LPA* algorithm was consistently the fastest of the algorithms tested on all queries on the Shenzhen network. It speeded-up the improved Yen's algorithm by approximately 2.5 times (i.e., 571.3/164.1-1) for calculating k shortest simple paths on the Shenzhen network. This speed-up rate was achieved due to its incorporation of the LPA* technique in deviation path calculation process. Accordingly, every deviation path was efficiently determined by reusing the results of the shortest path tree generated in the previous search. Few nodes were thus explored in each calculation of the deviation path. As summarized in Table 7, KSP-LPA* explored only $n = 0.5 \times 10^6$, approximately 73.4 times smaller than those explored by the improved Yen's algorithm. Compared with the re-optimization technique used in the M-P algorithm, the LPA* technique can avoid the computational overhead caused by updating the entire shortest path tree in every deviation path calculation. Therefore, the KSP-LPA* algorithm can significantly improve computational performance compared with the M-P algorithm in terms of n and t by approximately 256.8 and 14.5 times, respectively.

Fig. 4 also shows the computational performance of the V-F algorithm on the Shenzhen network. As summarized in Table 7, it ran approximately 53.5% (i.e., 571.3/372.2-1) faster than the improved Yen's algorithm because the V-F algorithm pre-calculates the shortest path from every node to the destination and tries to use them to speed-up deviation path calculation. Nevertheless, the proposed KSP-LPA* algorithm was still the best one as its computational time was 1.3 times (i.e., 372.2/164.1-1) shorter than that of the V-F algorithm.

Table 7 summarizes the computational performance of the five testing algorithms on all road networks with different sizes. As shown, the performance of all algorithms degraded with an increase in network size. For example, the improved Yen's algorithm took 1.1 s for calculating k shortest simple paths on the Hong Kong RTIS network with 1367 nodes. When using the Shenzhen network, the

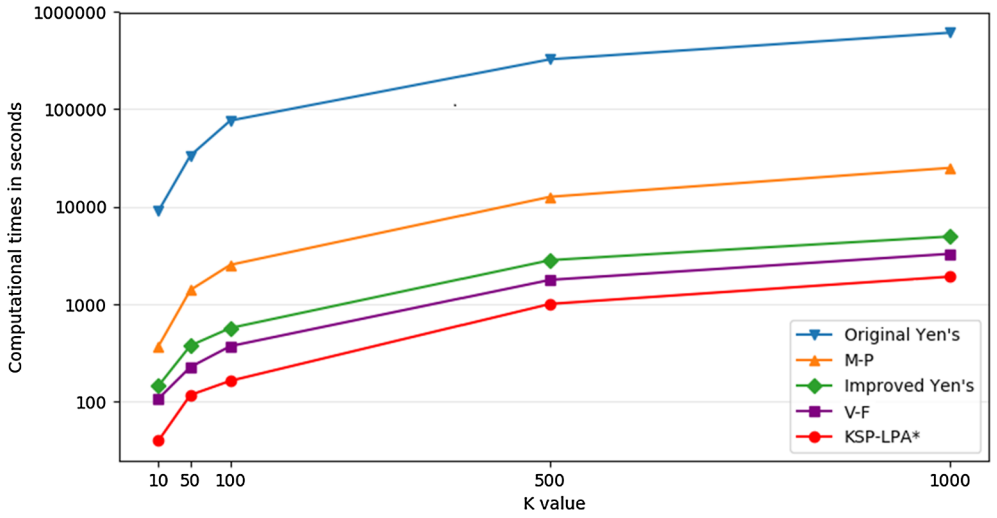

Fig. 6. The average computational times of KSP algorithms in the Shenzhen network with different values of k.

number of nodes increased 49.6 times but the computational time of the improved Yen's algorithm increased 518.3 times. Of the five algorithms tested, the proposed KSP-LPA* algorithm had the lowest rate of degradation with increasing network size. For instance, when the Shenzhen network was analyzed, the computational time of the KSP-LPA* algorithm degraded by only 204.1 times. This result is expected because the KSP-LPA* algorithm explores fewer nodes and thus is less sensitive to network size than the other four algorithms.

Fig. 6 shows the average computational times of all algorithms on the Shenzhen network with varying values of k. As is expected, the performance of all algorithms degraded with an increase in the value of k, since the number of calculated deviation paths (stored in L and C) increased with the k value. The rate of performance degradation was reduced when k become large, e.g. $k = 100$. This is because, when k become large, many deviation paths has been calculated and stored in L and C and do not need to be re-calculated in the k shortest path search process. It can be seen that the proposed KSP-LPA* algorithm was consistently the fastest of all algorithms under various values of k.

5. Conclusions

This study proposed a deviation path algorithm called KSP-LPA* for exactly finding k shortest simple paths in a road network between a given origin and a destination. Inspired by the M-P algorithm (Martins and Pascoal, 2003), the proposed KSP-LPA* algorithm calculates deviation paths by updating a single shortest path tree rooted at the destination. Unlike one-to-all shortest path searches used in the M-P algorithm, it formulates the process of calculation of deviation paths as the problem of one-to-one shortest path searches in a dynamic updating network, where only a node and a link are restored in each search. Using this formulation, the LPA* technique is used to speed-up deviation path calculation by reusing results of shortest path tree generated at the previous search. It can thus avoid the large computational overhead of the M-P algorithm incurred because the entire shortest path tree needs to be updated during each calculation of the deviation path. The KSP-LPA* algorithm can achieve the optimal solution of finding k shortest simple paths, and its optimality has been proven.

To demonstrate the efficiency of the proposed KSP-LPA* algorithm, computational experiments were conducted using five road networks of different sizes. For comparisons, four state-of-the-art algorithms for finding k shortest simple paths were implemented in addition to the propose one: Yen's original algorithm (Yen, 1971), the improved Yen's algorithm, the M-P algorithm (Martins and Pascoal, 2003), and the V-F algorithm (Vanhove and Fack, 2012). The results showed that the proposed KSP-LPA* algorithm was significantly faster than the other four algorithms on all networks with varying values of k. For example, its computation time was 14.5 and 2.5 times shorter than that of the M-P algorithm and the improved Yen's algorithm, respectively. The results also showed that the computational performance of the KSP-LPA* algorithm was less sensitive to an increase in network size than the other four algorithms.

Several directions for future research are worth pursuing. First, the F-heap data structure and C# programming language were used for implementing the proposed KSP-LPA* algorithm. Using C++ and alternative priority queue structures may further improve its computational performance. Second, the proposed algorithm considers only the road transport mode. Extending it to multi-mode networks (Xu et al., 2012; Khani et al., 2015), such as transit and bus, needs to be further investigated. Finally, the proposed algorithm can be extended to solve the reliable k shortest paths problem (Chen et al., 2016; Chen et al., 2013; Zeng et al., 2015; Zhang et al., 2016; Chen et al., 2017; Yang and Zhou, 2017) to explicitly consider reliability concerns in the face of travel time uncertainties.

CRediT authorship contribution statement

Bi Yu Chen: Conceptualization, Funding acquisition, Methodology, Writing - original draft. **Xiao-Wei Chen**: Conceptualization, Methodology. **Hui-Ping Chen**: Conceptualization, Methodology, Writing – review editing. **William H. K. Lam**: Conceptualization, Funding acquisition.

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 China (No. 41571149), 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 A

See Table A1.

Proposition A1. *The FindSpurPath-LPA* procedure can determine the optimal shortest path between the deviation node and the destination in every calculation of the spur path.*

Proof. As mentioned in Section 3, the *FindSpurPath-LPA** procedure makes three modifications to the original LPA* technique (Koenig, 2004), which has been proved to achieve the optimal shortest path. First, backward search is adopted instead of the forward search used in the original LPA* technique. According to Daganzo (2002), the shortest path problem is reversible, and the solution obtained by backward search is equivalent to that obtained by forward search. Second, the deviation node changes in each subsequent spur path calculation instead of remaining fixed as in the original LPA* technique. Because the *FindSpurPath-LPA**

Table A1
Spur path calculation procedure in the M-P algorithm.

Procedure: *FindSpurPath-MP*

Input: restored node n_i^j and linka_i^j
Return: Spur path $\bar{s}_i^j$.
01: Set node label $p_{n_i^j d}$: =∅ and cost $t(p_{n_i^j d})$: =∞.
02: For each node$n_v \in \mathrm{SUCC}(n_i^j)$
02: If $p_{n_v d} \neq \emptyset$ and $t(p_{n_i^j d}) > t(p_{n_v d}) + t(a(n_i^j, n_v))$ then
03: Set $p_{n_i^j d}$: $=p_{n_v d} \oplus a(n_i^j, n_v)$ and $t(p_{n_i^j d})$: $=t(p_{n_v d}) + t(a(n_i^j, n_v))$.
04: End If
05: End For
06: If $p_{n_i^j d} = \emptyset$ Then
07: Call *BackwardStar*(n_i^j) procedure.
08: End If
09: Set $\bar{s}_i^j := p_{n_i^j d}$.
10: Restore a_i^j toG.
11: If $t(p_{n_i^j d}) > t(p_{n_{i+1}^j d}) + t(a_i^j)$ Then
12: Call *BackwardStar*(n_{i+1}^j) procedure.
13: End If
14: Return $\bar{s}_i^j$.
Procedure: *BackwardStar*
Input: node n_u.
01: Set *list* := $\{n_u\}$.
02: While *list*≠ ∅
03: Select n_i from *list*, and set *list* := *list*− $\{n_i\}$.
04: For each node$n_v \in \mathrm{PRED}(n_i)$
05: If$t(p_{n_v d}) > t(p_{n_i d}) + t(a(n_v, n_i))$
06: Set $p_{n_v d}$: $=p_{n_i d} \oplus a(n_v, n_i)$ and $t(p_{n_v d}) = t(p_{n_i d}) + t(a(n_v, n_i))$
07: *list* := *list* ∪ $\{n_v\}$.
08: End If
09: End For
10: End While

procedure maintains a single shortest path tree rooted at the destination, the change in the deviation node in each subsequent calculation of the spur path does not affect the values of $g(n_u)$ and $rhs(n_u)$ for any node, and thus does not affect the optimality of the LPA* technique. Finally, the heuristic function is set to zero for every node in the network. According to Koenig (2004), the LPA* technique can achieve the optimal solution when the heuristic function is zero. Therefore, the *FindSpurPath-LPA** procedure can determine the optimal shortest path in every spur path calculation. □

Lemma A1. *The KSP-LPA* algorithm can exactly solve the problem of finding k shortest simple paths.*

Proof. The KSP-LPA*algorithm follows the same generic procedure (in Table 1) as Yen's algorithm (Yen, 1971), which has been proven to obtain the optimal k shortest simple paths. There are two differences between the KSP-LPA* algorithm and Yen's algorithm. First, the KSP-LPA* algorithm adopts the *CalDevPaths-LPA** procedure to calculate the deviation path in reverse order from the destination to the deviation node, instead of the forward order from the deviation node to the destination. It is clear that the calculation of the deviation path in the reverse order can determine the same deviation path set D^j as that in the forward order. Second, the KSP-LPA* algorithm adopts the *FindSpurPath-LPA** procedure to calculate the spur path instead of the forward Dijkstra algorithm used in Yen's algorithm. According to Proposition 1, the *FindSpurPath-LPA** procedure can determine the optimal shortest path just as the forward Dijkstra's algorithm does. Therefore, the KSP-LPA* algorithm can exactly solve the problem of finding k shortest simple paths. □

Appendix B. Supplementary material

Supplementary data to this article can be found online at https://doi.org/10.1016/j.tre.2019.11.013.

References

Bast, H., Funke, S., Sanders, P., Schultes, D., 2007. Fast routing in road networks with transit nodes. Science 316 566-566.
Bast, H., Delling, D., Goldberg, A., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R., 2014. Route planning in transportation networks. Microsoft Res. Rep.
Chen, B.Y., Lam, W.H.K., Sumalee, A., Li, Q.Q., Shao, H., Fang, Z.X., 2013. Finding reliable shortest paths in road networks under uncertainty. Netw. Spatial Econ. 13, 123–148.
Chen, B.Y., Li, Q.Q., Lam, W.H.K., 2016. Finding the k reliable shortest paths under travel time uncertainty. Transp. Res. Part B 94, 189–203.
Chen, B.Y., Shi, C., Zhang, J., Lam, W.H.K., Li, Q.Q., Xiang, S., 2017. Most reliable path-finding algorithm for maximizing on-time arrival probability. Transportmetrica B 5, 253–269.
Clarke, S., Krikorian, A., Rausen, J., 1963. Computing the N best loopless paths in a network. J. Soc. Ind. Appl. Math. 11 (4), 1096–1102.
Daganzo, C.F., 2002. Reversibility of the time-dependent shortest path problem. Transp. Res. Part B 36, 665–668.
Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F., 2017. Customizable route planning in road networks. Transport. Sci. 51, 566–591.
Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. Numer. Math. 1 (1), 269–271.
Eppstein, D., 1998. Finding the k shortest paths. SIAM J. Comput. 28, 652–673.
Fox, B.L., 1973. Calculating Kth shortest paths. Infor Inform. Syst. Operat. Res. 11, 66–70.
Fredman, M.L., Tarjan, R.E., 1987. Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM 34, 596–615.
Fu, L., Sun, D., Rilett, L.R., 2006. Heuristic shortest path algorithms for transportation applications: state of the art. Comput. Oper. Res. 33, 3324–3343.
Geisberger, R., Sanders, P., Schultes, D., Vetter, C., 2012. Exact routing in large road networks using contraction hierarchies. Transport. Sci. 46, 388–404.
Goldberg, A.V., Harrelson, C., 2005. Computing the shortest path: A* search meets graph theory. In: Proceeding of ACM-SIAM Symposium on Discrete Algorithms, pp. 156–165.
Hart, P.E., Nilsson, N.J., Raphael, B., 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybernetics SSC-4(2), 100–107.
Hershberger, J., Maxel, M., Suri, S., 2007. Finding the k shortest simple paths: a new algorithm and its implementation. ACM Trans. Algor. 3, 45.
Hilger, M., Köhler, E., Möhring, R., 2009. Fast point-topoint shortest path computations with arc-flags. In: The Shortest Path Problem: Ninth DIMACS Implementation Challenge, pp. 41–72.
Huang, B., Wu, Q., Zhan, F.B., 2007. A shortest path algorithm with novel heuristics for dynamic transportation networks. Int. J. Geograph. Inform. Sci. 21, 625–644.
Khani, A., Hickman, M., Noh, H., 2015. Trip-based path algorithms using the transit network hierarchy. Netw. Spatial Econ. 15, 635–653.
Koenig, S., Likhachev, M., Furcy, D., 2004. Lifelong planning A. Artif. Intell. 155 (1–2), 93–146.
Kuijpers, B., Moelans, B., Othman, W., Vaisman, A., 2016. Uncertainty-based map matching: the space-time prism and k-shortest path algorithm. ISPRS Int. J. Geo-Inf. 5, 204.
Li, Q.Q., Chen, B.Y., Wang, Y., Lam, W.H.K., 2015. A hybrid link-node approach for finding shortest paths in road networks with turn restrictions. Trans. GIS 19, 915–929.
Martins, E.Q.V., Pascoal, M.M.B., 2003. A new implementation of Yen's ranking loopless paths algorithm. 4OR 1 (2), 121–133.
Pugliese, L.D.P., Guerriero, F., 2013. A reference point approach for the resource constrained shortest path problems. Transport. Sci. 47, 131–294.
Qian, C., Chan, C.-Y., Yung, K.-L., 2011. Reaching a destination earlier by starting later: revisited. Transp. Res. Part E 47, 641–647.
Raith, A., Ehrgott, M., 2009. A comparison of solution strategies for biobjective shortest path problems. Comput. Oper. Res. 36, 1299–1331.
Roddity, L., 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.
Srinivasan, K.K., Prakash, A.A., Seshadri, R., 2014. Finding most reliable paths on networks with correlated and shifted log-normal travel times. Transp. Res. Part B 66, 110–128.
Vanhove, S., Fack, V., 2012. An effective heuristic for computing many shortest path alternatives in road networks. Int. J. Geograph. Inform. Sci. 26 (6), 1031–1050.
Xu, W.T., He, S.W., Song, R., Chaudhry, S.S., 2012. Finding the K shortest paths in a schedule-based transit network. Comput. Oper. Res. 39, 1812–1826.
Yang, L., 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. Transp. Res. Part B 96, 68–91.
Ye, X., She, B., Benya, S., 2018. Exploring regionalization in the network urban space. J. Geovisual. Spatial Anal. 2, 4.
Yen, J.Y., 1971. Finding the k shortest loopless paths in a network. Manage. Sci. 17 (11), 712–716.
Yu, H., Fang, Z., Lu, F., Murray, A.T., Zhang, H., Peng, P., Mei, Q., Chen, J., 2019. Impact of oil price fluctuations on tanker maritime network structure and traffic flow changes. Appl. Energy 237, 390–403.
Yue, H., Guan, Q., Pan, Y., Chen, L., Lv, J., Yao, Y., 2019. Detecting clusters over intercity transportation networks using K-shortest paths and hierarchical clustering: a case study of mainland China. Int. J. Geograph. Inform. Sci. 33, 1082–1105.
Zhang, Y., Shen, Z.-J.M., Song, S., 2016. Parametric search for the bi-attribute concave shortest path problem. Transp. Res. Part B 94, 150–168.
Zeng, W., Church, R.L., 2009. Finding shortest paths on real road networks: the case for A*. Int. J. Geograph. Inform. Sci. 23, 531–543.
Zeng, W., Miwa, T., Wakita, Y., Morikawa, T., 2015. Application of Lagrangian relaxation approach to a-reliable path finding in stochastic networks with correlated link travel times. Transp. Res. Part C 56, 309–334.