An Iterative Algorithm for Single-pair K Shortest Paths Computation

GUISONG LIU*, ZHAO QIU, WENYU CHEN
School of Computer Science and Engineering
University of Electronic Science and Technology of China
Xiyuan Road 2006, West High-Tech Zone, Chengdu, China
*lgs@uestc.edu.cn

Abstract: - In this paper, we report a novel method to compute the k shortest paths between a given pair of nodes in a given directed weighted graph, where loops are allowed in the solution paths. Once the shortest path from source node to goal node has been computed, the algorithm finds the next $k-1$ shortest paths recursively. A* and on-the-fly search strategies are also applied to the proposed algorithm. The correctness of the presented algorithm is analyzed mathematically, and the simulative results confirming the superior performance of the algorithm to others in the literature for real road datasets are reported, especially when k is rather small.

Key–Words: - K shortest paths, Heuristic search, On-the-fly search

1 Introduction

The K-Shortest-Paths problem (KSP) is about finding the k shortest paths in a directed weighted graph for an arbitrary natural number k. Since first proposed by Hoffman and Pavley in the 1950s [1], KSP has got great attention and can be separated into two variants according to whether loops are allowed in the solution paths or not [2]. Recent application domain examples for KSP problems include network routing optimization[3], multiple object tracking[4], sequence alignment[5], gene network[6], scheduling[7], dynamic routing[8] and many other areas in which optimization problems need to be solved[9].

The goal of most of algorithms is to compute KSPs between two given nodes, which also can be called single-pair KSP problem [2, 10, 11, 12, 14, 19, 16]. The related problem is single-source KSP problem which aims to find KSPs from a given node to each other node[6, 9]. In this paper, we consider the variant of the KSP problem, where loops are allowed in the solution paths. In terms of asymptotic complexity, the most advantageous algorithm for solving this variant of KSP is Eppstein's Algorithm (EA)[9], with a complexity of $O(m + n\log n + k)$ in terms of both runtime and space, where n is the number of nodes and m is the number of edges in the problem graph. Jimenez and Marzal presented a Lazy Variant of Eppstein's Algorithm (LVEA) [21], which maintains the same asymptotic worst-case complexity as EA and outperformed EA in their experiment results. The K* algorithm proposed by Husain Aljazaar and Stefan Leue [10] is the most efficient algorithm known so far according to their experimental results. It maintains the same asymptotic worst-case complexity as the EA algorithm. Their experimental evaluation showed that K* is more efficient than LVEA in route planning and the computation of counterexamples in stochastic model checking. However, it is essential for EA, LVEA and K* to establish a complicated data structure before finding the k shortest paths. In this paper, we present an Heuristic and Recursive Algorithm by using on-the-Fly search (HRAF), aiming to solve the KSP problem, which loops and multiple edge between nodes are allowed. When the shortest path is computed, the algorithm is able to find the next $k-1$ shortest paths in a recursive way. HRAF is very efficient when k is small while time-consuming when k is large. But in a lot of KSP applications, the key point is to find out few best solutions rather than to enumerate all or a large number of shortest paths in a graph.

The remainder of this paper is organized as follows. Some preliminaries are described in Section 2, including A* and on-the-fly search strategies. Section 3 presents the proposed algorithm with its implementation in detail. An simple example to illustrate HRAF is carried out in Section 4. Section 5 gives the correctness and complexity analysis of HRAF. Simulations based on a benchmark data set and conclusions are presented in Section 6 and Section 7, respectively.

2 A* and On-the-fly Search

Let $G = (V, E)$ be a directed graph, where V is the set of nodes and $E \subseteq V \times V$ is the set of edges. Given an edge $e = (u, v) \in E$, we represent $tail(e)$ by u

and $head(e)$ by v. Let $w : E \rightarrow R > 0$ be a function mapping edges to non-negative real-valued weights or lengths. Let **s**, **t**∈V denote the source and target nodes, respectively. A path in G is denoted by P, without loss of generality, P_n denotes the n-th shortest **s**-**t** path in G. The length of a path $P = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$ is defined as the sum of the edge lengths, formally,

$$l(P) = \sum_{i=1}^{n-1} w(v_i, v_{i+1}) \quad (1)$$

For an arbitrary pair of nodes u and v, $d(u, v)$ denotes the length of the shortest path from u to v, and $d(\mathbf{s}, u)$ is abbreviated to $d(u)$. If there is no path from u to v, then $d(u, v)$ is equal to $+\infty$.

It is known that the A* search[23] is designed for solving the shortest path problem while making use of a heuristic estimate. It finds the shortest path from the start node **s** to each node in G. The set of these paths forms a tree called the shortest path tree T. A* stores nodes on the search front in a priority queue which is ordered according to a heuristic evaluation function f, computed as the sum of two functions d and h,

$$f = d + h. \quad (2)$$

The function d gives the shortest path length from node **s** to a node, whereas h is the heuristic estimate of the distance from the considered node to the target. As shown in Fig. 1, $f(v)$ is then an estimate of the length of an **s**-**t** path through v.

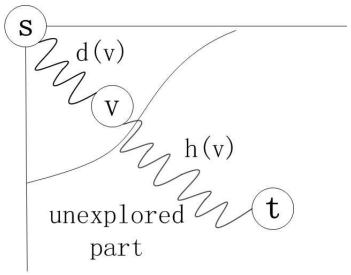

Figure 1: A* search: $f(v) = d(v) + h(v)$ determines the expansion order of nodes.

We describe the traditional implementation of A* as follows and we will show the variant of A* implementation used in HRAF later. At the beginning of A* search, the search queue contains only the start node **s** with $f(\mathbf{s}) = h(\mathbf{s})$. In each search iteration, the head of the search queue, say u, is removed from the queue and expanded. More precisely, for each successor node v of u, if v has not been explored before, then $d(v)$ is set to $d(u) + w(u, v)$, and v is put into the search queue. If v has been explored before, then $d(v)$ is set to the smaller distance of the old $d(v)$ and $d(u) + w(u, v)$. We distinguish between two types of explored nodes, namely closed and open nodes. Closed nodes are those which have been explored and expanded, whereas open nodes are those which have been explored but not yet expanded.

Notice that there is a problem when use the above traditional version of A* to HRAF directly, remember that HRAF can be used to the graph which multiple edges between two nodes is allowed, thus, the weight $w(u, v)$ between node u and v is ambiguous. For efficiency consideration and the use of on-the-fly search strategy, we use the flowing implementation of A*.

The main difference between our implementation of A* and the traditional implementation of A* described above is that we store edges instead of nodes in the search queue, and the edge in the search queue is sorted according to the value of function $f(e)$ which can be calculate for edge e by:

$$f(e) = d(tail(e)) + h(head(e)) + w(e). \quad (3)$$

What is more, no relaxation(change the value of d in traditional implementation of A*) of our implementation of A* is required.

For simplicity, the process of our implementation can be described as flows (see Fig.1). At the beginning of the search, set $d(\mathbf{s}) = 0$ and set the state of **s** as closed, the search queue contains only those edge which outgoing from start node **s** with $f(\mathbf{e}) = h(head(e)) + d(\mathbf{s}) + w(e)$. In each search iteration, the head of the search queue, say e, is removed from the queue for processing, $head(e)$ is either closed, in this case e is marked as $sidetrack\ edge$, or not closed, when e is set as a $tree\ edge$ and the vertex $head(e)$ is expanded: for each edge $e*$ outgoing from $head(e)$, if $head(e*)$ is not closed, $f(e)$ is calculated and it is added to the search queue, otherwise, $e*$ is set as $sidetrack\ edge$. We continue to do so until the target node is found or the search queue is empty.

For each explored node v, $d(v)$ is always equal to the length of some path from **s** to v that has been discovered so far. We refer to this path as the solution base of v. The set of these solution bases forms a search tree T. A* ensures that the search tree T is a shortest path tree for all closed nodes. Notice that a shortest **s**-**t** path is found as soon as **t** is closed. In order to retrieve the selected shortest path to some node, a link $T(v)$ is attached to each explored node v referring to the parent of v in T . The solution path can then be constructed by following these links from t upwards to **s**.

The heuristic function h should be admissible, which guarantees that a shortest **s**-**t** path will be found by A*, referring to Ref[23] for more information.

Some search algorithms can be performed on-the-fly, including A*. This means that they can be applied to an implicit description of G, by using **s** and a function $succ : V \rightarrow \Pi(V)$, which returns the set of its successor nodes for each node u, i.e., $succ(u) = \{v|v \in V, (u, v) \in E\}$. The on-the-fly strategy enables the partial generation and processing of the problem graph as needed by the search algorithm. This strategy is able to improve the performance and scalability of many search algorithms because there is no need to process the entire graph. It also saves memory since the search algorithm does not need to manage the entire graph in its data structures. The on-the-fly feature finally allows the algorithm to handle graphs which are either infinite, or finite but too large to fit into main memory.

3 The Proposed Algorithm

The HRAF algorithm is designed to quickly find the k shortest paths as soon as the shortest path from source node **s** to goal node **t** is obtained. In HRAF, A* search is first applied to the problem graph G until **t** is chosen for expansion. At this moment, A* is suspended and the first shortest **s-t** path is acquired. Then the next k-1 shortest paths can be constructed in a recursive way as follows,

1. Exploring the i^{th} shortest path to obtain all candidate shortest paths.
2. Choosing a shortest path from candidate path list to be the $(i+1)^{th}$ shortest-path from **s** to **t**.
3. Recursively doing this until all k shortest paths are found.

As we mentioned earlier, HRAF is designed to perform on-the-fly and to be guided by heuristic search. In the process of recursively finding k-1 shortest paths, A* will be resumed as needed.

3.1 A* search on G

HRAF applies A* search to the problem graph G to compute a search tree T. Note that A* computes a search tree while searching for a shortest path from **s** to **t**, this tree is formed by the father-node links that are stored while A* is searching in order to be able to reconstruct the path from **s** to **t**. Every edge founded by A* will be marked as either $tree\ edge$ or $sidetrack\ edge$. If an edge (u, v) belongs to the shortest path tree T, we call it a $tree\ edge$, otherwise, it is a $sidetrack\ edge$. As shown in Fig.2, let s_0 be the start node and s_3 be the target node. The shortest

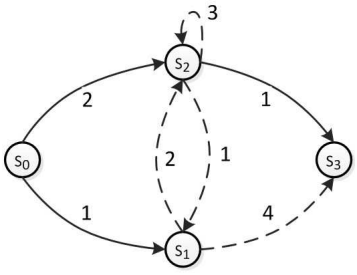

Figure 2: Example graph: the solid arrow lines represent the search tree computed by A*; the dashed arrow lines are the sidetrack edges.

path tree is highlighted by solid arrow lines. We apply A* to G in a forward manner, which yields a search tree T rooted at **s**. The forward search strategy is necessary in order to be able to work on an implicit description of the problem graph G using the successor function $succ$. Notice that HRAF can perform on-the fly search: HRAF first applies A* search to G until the goal node **t** is selected for expansion and resume A* when required. In the remainder of the paper, G is assumed to be a locally finite graph, if nothing else is explicitly stated.

3.2 Path Representation

We represent every **s-t** path as a list of $sidetrack\ edges$. Each edge either belongs to the shortest path tree T, $tree\ edge$ or is a $sidetrack\ edge$, as we mentioned above. For any **s-t** path P in G, we denote the subsequence of sidetrack edges using $\xi(P)$, which are taken in P. In this way, P can be unambiguously described by the sequence $\xi(P)$. In other words, the mapping $\xi(.)$ from subsequence of $sidetrack\ edges$ to **s-t** path is injective. Consequently, there is a partial injective inverse mapping $\chi(.)$ so that $\chi(\xi(\text{P}))=P$. The mapping $\xi(.)$ establishes this unique way of completing the sequence of $sidetrack\ edges$ ξ(P) by adding the missing tree edges in order to obtain P(see Ref[9]). In Fig. 2, let P be the path $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$. Notice that $\pi=\xi(\text{P})= \{(s_1, s_2)\}$ and $next(\pi)=s_1$. From the sidetrack sequence $\{(s_1, s_2)\}$ we can obtain the pre-image $\chi\{(s_2, s_1)\}=P$, P=$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$ as follows. We start at the goal node s_3 and add the tree edge (s_2, s_3) to P. It is clear that s_2 is the head node of the sidetrack edge(s_1, s_2), so we add the sidetrack edge (s_1, s_2) to P. Then the $treeedge$ (s_0, s_1) is added to P. This results in completing the path P = $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3$.

In the following section, we denote π to be the subsequence of $sidetrack\ edge$ of an **s-t** path in G, and $\pi_i = \xi(P_i)$ to be the subsequence of sidetrack

edges of the i^{th} shortest path P_i in G. Let $next(\pi_i)$ be the next node of the start node **s** in path P_i.

Representing paths by $sidetrack\ edge$ is very important in our algorithm. As we have described, we can represent a path by sequence of $sidetrack\ edge$, which is a very *cheap way*. In this way, all candidate paths can possibly be explored efficiently.

3.3 Computing k Shortest Paths Recursively

In order to clarify our procedure, we first illustrate how HRAF computes the second shortest path from the (first) shortest path, and then from the i^{th} shortest path to the $(i+1)^{th}$ shortest path.

After the shortest-path tree on G is constructed, we obtain the shortest **s-t** path $\pi_1 = \xi(P_1)$, as well as every other node's (closed in A*) shortest path to **s**. Note that our shortest path tree T is rooted at the start node **s**. We iterate over all nodes except **s** in the shortest path to produce all candidate paths for the second shortest path in the following way: for every node v in the shortest path except **s**, we consider every sidetrack edge e which is incoming to v, as a candidate path, whose length is calculated by $d(\mathbf{t}) - d(v) + w(e) + d(tail(e))$. Then, we add this candidate path to the candidate path list C ordered by path length. Finally, we choose a shortest path from C to be the second shortest path π_2 and remove this path from C. π_{i+1} can be constructed from π_i in a similar way. We iterate over all nodes in P_i from $next(\pi_i)$ to the tail of the last sidetrack edge of π_i. Let lv be the tail node of the last sidetrack edge in π_i. For every incoming sidetrack edge e to v (v is the node in P_i from $next(\pi_i)$ to lv), by adding e to the end of π_i, a candidate path π_c can be constructed whose length is $l(\pi_c) - d(v) + w(e) + d(tail(e))$. Then π_c is added to the candidate path list C. We regard all these candidate paths found in this step as the candidate paths explored by π_i. Finally, we choose the shortest path from C to be π_{i+1} and remove it from C.

3.4 Implementation of the Algorithm

The algorithm principle of HRAF is shown in Fig. 3. First, we run A* on graph G until the target node **t** is selected for expansion. In this moment, we get the first shortest path from **s** to **t**, which is consist of $tree\ edges$ and is a empty sequence when represented by $sidetrack\ edges$. Then, we explore the next k-1 shortest path tree recursively. Algorithm 1 shows the pseudo-code of HRAF.

The code from line 12 to line 29 forms the main loop of HRAF. The loop terminates when the candidate path list C is empty or the kth shortest path is produced. The lines before line 12 perform some preparation tasks. After some initialization statements, A* is started at line 8 until **t** is selected for expansion, in which case the first shortest **s-t** path has been found. If **t** is not reachable, then the algorithm terminates without a solution. Notice that it would not terminate on an infinite graph. Otherwise, the algorithm assigns the goal node **t** to lv, which represents the tail node of the last sidetrack edge of the previous shortest path. HRAF then enters its main loop. After having got the first shortest **s-t** path π_1, we compute the next k-1 shortest **s-t** paths recursively.

HRAF iterates over all nodes v in P_i from lv to $next(\pi)$(the second node in **s-t** path $\chi(\pi)$) (see line 13), and for each incoming edge(not marked as $tree\ edge$) e to v. If $tail(e)$ is not closed in A*, then we resume A* until $tail(e)$ is selected for expansion. Notice that every closed node has exactly one incoming $tree\ edge$, thus, if e is not set as $sidetrack\ edge$ set e as sidetrack edge. Line 19 to 21 construct candidate paths that relate to e, and decide if this candidate path should be added to the candidate path list C or not. Notice that in line 21, in order to make sure the candidate path list C contain at most k-1 paths. If C is not full, add the new candidate path to C, otherwise, choose the shortest one of the new candidate path and longest path in C. In line 24 to 28, the parameters is updated to prepare for the next loop. If we have pro-

Algorithm 1: The HRAF Algorithm.

Data: A graph given by its start node **s** and goal node **t** and a function *succ* and a natural number k

Result: A list R containing k *sidetrack edge* sequences representing k shortest paths

```
1  Function succ(v) return all edges outgoing or incoming from v.
2  openA : empty priority queue, store opened vertices in A*.
3  closeA: empty hash table, store closed vertices in A*.
4  C: empty priority queue, candidate path list, ordered by path length.
5  R: empty list, result paths list.
6  π:empty list, represent a path(consist of sidetrack edge).
7  lv ← t.
8  run A* on G until the goal node t is selected for expansion.
9  if t was not reached then
10     exit without a solution.
11 end
12 do
13     foreach v in χ(π) from lv to next(π) do
14         foreach incoming edge e in succ(v) and is not tree edge do
15             if tail(e) is not in closeA then
16                 run A* until tail(e) is added to closeA.
17             end
18             if e is not set as sidetrack edge then set e as sidetrack edge
19             add e to π form P_c
20             calculate length of P_c
21             add P_c to C
22         end
23     end
24     π ← shortest path on C
25     erase π from C
26     lv ← tail of last sidetrack edge of π
27     add π to R
28     if R.size == k then break
29 while C is not empty
30 return R
```

Figure 3: HRAF Algorithm Structure

duced the kth shortest path or candidate path list C is empty, HRAF terminates, otherwise, it continues with the main loop.

4 An Example Using HRAF

We use the following example to illustrate how HRAF works. Consider the directed, weighted graph G in Fig. 4 which is similar to the example graph used in [10]. The start node is s_0 and the target node is s_6. We use HRAF to find the 8 shortest paths from s_0 to s_6. We assume that a heuristic estimate exists. The heuristic values are given by the labels $h(s_0)$ to $h(s_6)$ in Fig. 4. It is easy to see that this heuristic function is admissible. We first apply A* search to G until s_6 is found. The part of G explored so far is illustrated in Fig. 5. Note that the solid arrow lines represent the search tree computed by A*; the dashed arrow lines are the $sidetrackedges$. A* is suspended after the goal node s_6 is found, and the first shortest path ($s_0 \rightarrow s_2 \rightarrow s_4 \rightarrow s_6$) is obtained, then HRAF begins to find next k-1 shortest paths. Initially, the candidate path list C is empty. We explore the first shortest path ($s_0 \rightarrow s_2 \rightarrow s_4 \rightarrow s_6$) to obtain candidate paths by iterating over each node from s_6 to s_2 along the shortest path tree T. As we explore s_6, it has three edges, one $tree\ edge$ (s_4,s_6), then the other two edges, (s_1,s_6) and (s_5,s_6), are set to $sidetrack\ edges$. When edge (s_1,s_6) is processed, it is noticed that s_1 is not closed in A* so we should resume A* until s_1 is closed, Figure 6 shows the result graph when s_1 is closed. Then, we consider $\{(s_1, s_6)\}$ as a candidate path with a length $w(s_1, s_6)$ + $d(s_1)$ = 10, and this candidate path is added to the candidate path list. Similarly, when edge (s_5,s_6) is processed, it is set as $sidetrack\ edge$, consider that s_5 is not closed in A*, we resume A* until s_5 is closed. Fig. 7 illustrates the updated graph when s_5 is closed in A*. We then consider $\{(s_5, s_6)\}$ as a candidate path with length $w(s_5, s_6) + d(s_5)$= 20 and add this candidate path to C. After s_6 is completely explored, we then consider s_4 which is father node of s_6 in the shortest path tree T. We continue to explore s_2. The result of the exploration in this step is illustrated in Table 1.

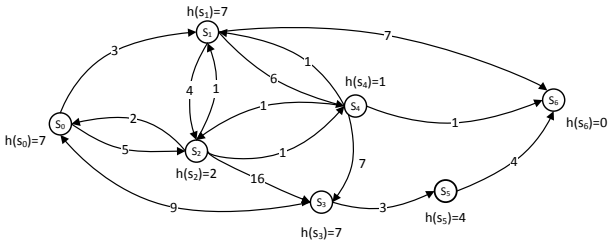

Figure 4: Example graph with 7 nodes and 16 edges.

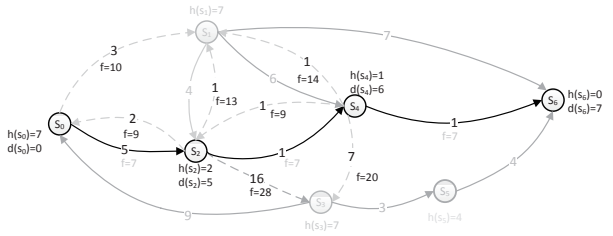

Figure 5: The explored graph when the goal node is found using A*: solid arrow lines represent tree edges; dotted arrow lines represent $sidetrackedges$; gray dotted lines represent the edges added to open set in A*; the gray solid lines represent the unexplored part of the graph; the f value of edges is written below lines.

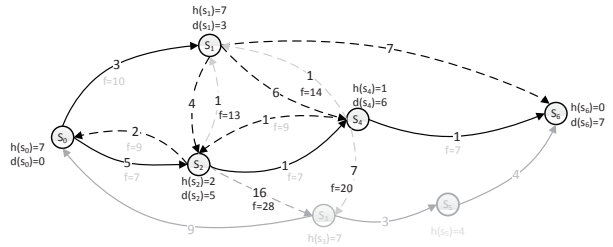

Figure 6: Result graph when s_1 is closed in resumed A*.

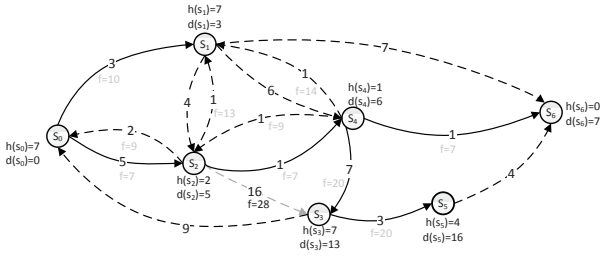

Figure 7: Result graph when s_5 is closed in resumed A*.

After having explored the first shortest path, we have got five candidate paths with the lengths in candidate path list C, i.e., ($s_2 \rightarrow s_4$, 9),($s_1 \rightarrow s_2$, 9),($s_1 \rightarrow s_6$, 10), ($s_1 \rightarrow s_4$, 10) and ($s_5 \rightarrow s_6$, 20). We then we

Table 1: Explored results of the first shortest path

Node	s_6		s_4	s_2	
Incoming sidetrack edge	(s_1, s_6)	(s_5, s_6)	(s_1, s_4)	(s_4, s_2)	(s_1, s_2)
Candidate path	$\{(s_1, s_6)\}$	$\{(s_5, s_6)\}$	$\{(s_1, s_4)\}$	$\{(s_4, s_2)\}$	$\{(s_1, s_2)\}$
Path length	10	20	10	9	9

choose a shortest path from C to be the second shortest path π_2. In this case, π_2=$\{(s4, s2)\}$, and π_2 is erased from C. Then we go on exploring π_2. We iterate over each node from s_4, which is the tail of the last sidetrack edge of π_2, to s_2 along T. Note that s_4 has one incoming $sidetrack\ edge$ (s_1, s_4) from which we obtain one candidate path $\{(s_4, s_2), (s_1, s_4)\}$, we add it to C. We then process s_2 which has two incoming $sidetrack\ edges$ $(s_4, s_2), (s_1, s_2)$. We add them to π_2 and obtain two candidate paths: $\{(s_4, s_2), (s_4, s_2)\}$ and $\{(s_4, s_2), (s_1, s_2)\}$, then we add them to C. The result of this step is listed in Table 2, and the content of C is listed in Table 3.

We repeat the above step until the k^{th} shortest path is found or candidate path list C is empty. The result of this example is enumerated in Table 4.

5 Algorithm Analysis

5.1 Correctness Analysis

Lemma 1: If π_i has j $sidetrackedge$, then the candidate paths explored by π_i have $j+1$ $sidetrackedges$.

Proof: Remember that the candidate paths explored by π_i are constructed by adding a sidetrack edge to π_i. Thus, in this way, if π_i has j sidetrack edges, the candidate paths explored by π_i have $j+1$ sidetrack edges.□

Lemma 2: The paths in candidate path list C have at most k-1 $sidetrack\ edges$.

Proof: Note that π_1 is empty, in other words, P_1 has no $sidetrackedge$, because it is exactly the first shortest path with all its edges in the shortest path tree. Note that π_i is not necessarily chosen from the candidate paths explored by π_{i-1}. Thus, according to Lemma 1, π_i has at most $i-1$ $sidetrackedges$, and π_k has at most $k-1$ $sidetrack\ edges$. So the path in candidate path list C has at most $k-1$ $sidetrackedges$ in the process of our algorithm. □

The importance of Lemma 2 lies in the space complexity boundary. From Lemma 2 we can conclude that the candidate path list C will at most contain k paths, we have more to say about it in Section 5.2. According to Lemma 2, each path has at most k-1 $sidetrack\ edge$, and we are intend to compute k shortest paths, thus, the worst-case space complexity of C is $O(k^2)$.

Lemma 3: The length of each path explored by π_i is larger than that of π_i.

Proof: Recall how we calculate the candidate path π_c explored from π_i, the length of π_c is $l(\pi_i) - d(v) + d(tail(e)) + w(e)$, where v is one of the nodes in P_i from $next(\pi_i)$ to the tail of the last sidetrack edge of π_i, e is an incoming sidetrack edge to v. Note that v is $head(e)$ since e is a sidetrack edge, thus, $d(tail(e)) + w(e) > d(v)$ and $l(\pi_i) - d(v) + d(tail(e)) + w(e) > l(\pi_i)$. □

Lemma 4: There is a one-to-one correspondence between paths in C and **s-t** paths in G.

Proof: We prove this lemma by induction. We first prove that there is a one-to-one correspondence between paths explored by π_1 and **s-t** paths in G, then prove that there is a one-to-one correspondence between paths explored by π_i and **s-t** path in G.

The first shortest path is the path from **s** to **t** along the shortest path tree T, in other words, π_1 is empty, thus, there is a one-to-one correspondence between π_1 and **s-t** paths in G. Next, we explore π_1. Let v be a node in P_1 from $next(\pi_i)$ to **t**, and let e be an incoming sidetrack edge of v. Then, we form a candidate path $\pi_c = e$, the corresponding **s-t** path in G is from **t** to head(e) along T, through e, then from $tail(e)$ to **s** along T, which is implicitly defined. Thus, there is a one-to-one correspondence between paths explored by π_1 and **s-t** paths in G. Let lv be the tail of the last sidetrack edge in π_i, similarly, there is a path π_i in G from **t** along P_i to lv. Let v is one of nodes in P_i from $next(\pi_i)$ to lv, and e is a sidetrack edge incoming to

Table 2: Result after having explored the second shortest path.

Node	s_4	s_2	
Incoming sidetrack edge	(s_1, s_4)	(s_4, s_2)	(s_1, s_2)
Candidate path	$\{(s_4, s_2)$,$(s_1, s_4)\}$	$\{(s_4, s_2)$,$(s_4, s_2)\}$	$\{(s_4, s_2)$,$(s_1, s_2)\}$
Path length	12	11	11

Table 3: Contents of candidate path list C after having explored the second shortest path

Candidate path	Length	Candidate path	Length
$\{(s_1, s_2)\}$	9	$\{(s_4, s_2)$,$(s_4, s_2)\}$	11
$\{(s_1, s_4)\}$	10	$\{(s_4, s_2)$,$(s_1, s_4)\}$	12
$\{(s_1, s_6)\}$	10	$\{(s_5, s_6)\}$	20
$\{(s_4, s_2)$,$(s_1, s_2)\}$	11		

Table 4: The results of 8 shortest paths on the graph in Fig. 4

	$\pi = \xi(P)$	P	length
1	$\{\}$	$s_0 \to s_2 \to s_4 \to s_6$	7
2	$\{(s_4, s_2)\}$	$s_0 \to s_2 \to s_4 \to s_2 \to s_4 \to s_6$	9
3	$\{(s_1, s_2)\}$	$s_0 \to s_1 \to s_2 \to s_4 \to s_6$	9
4	$\{(s_1, s_6)\}$	$s_0 \to s_1 \to s_6$	10
5	$\{(s_1, s_4)\}$	$s_0 \to s_1 \to s_4 \to s_6$	10
6	$\{(s_4, s_2)$,$(s_4, s_2)\}$	$s_0 \to s_2 \to s_4 \to s_2 \to s_4 \to s_2 \to s_4 \to s_1$	11
7	$\{(s_4, s_2)$,$(s_1, s_2)\}$	$s_0 \to s_1 \to s_2 \to s_4 \to s_2 \to s_4 \to s_6$	11
8	$\{(s_4, s_2)$,$(s_1, s_4)\}$	$s_0 \to s_1 \to s_4 \to s_2 \to s_4 \to s_6$	12

v, then, the candidate path we get now is $\pi_i + e$, the corresponding **s-t** path in G is from **t** along P_i to lv, then through e to **s** along T.□

The correctness of HRAF can be stated as follows. Let G be a locally finite graph, Lemma 4 ensures a one-to-one correspondence between paths in candidate paths list C and **s-t** paths in G. This implies the correctness of HRAF since any path from C results in a valid **s-t** path. In other words, the result of HRAF consists of valid **s-t** paths. Lemma 3 ensures us to compute the k shortest paths in a non-descending way. Since G is a locally finite graph, HRAF will complete after a finite number of iterations.

5.2 Theoretical Complexity

As we described in Section 3, the computation of HRAF comprises two steps, including applying A* search to G and finding the k shortest paths iteratively. It is known that performing A* has an asymptotic worst-case time complexity of $O(m + n \log n)$. In finding the k shortest paths by HRAF, we need to iterate over all incoming edges of the nodes which rang from the second node of the previous shortest path to the tail of the last sidetrack edge of previous shortest path. In the worst case, HRAF needs to iterate over m edges. Then, for each sidetrack edge, we add the corresponding paths to path list C. In order to limit the size of C (by using max-heap) to $k - 1$, we should determine the maximum element of C, which needs $O(m)$ time in worst-case (in practice very much less than $O(m)$). Decreasing the key of the maximum element of C and adjusting C need $O(m \log k + m)$ time. We also need to find the minimum path in C which needs $O(k)$ time, then we delete it from C and adjust the heap again which needs $O(\log(k))$ time. All the operations need to be performed $k - 1$ times, the total time complexity on C is $O((k-1)(m \log(k) + m + k + \log(k)))$, i.e., $O(km \log(k) + k^2)$. Thus, the worst case time complexity of HRAF is $O(mk \log k + k^2 + n \log n)$. For space complexity, in the worst case, we need to store n nodes and m edges, and maintain a candidate path list C with its size limited to $k - 1$. As we mentioned earlier, list C contains at most $k - 1$ candidate paths and each path consists of at most $k - 1$ $sidetrack$ $edges$ (see Lemma 2). Therefore, the worst case space complexity of HRAF is $O(m + n + k^2)$. Notice that some efficient implementation of heap or other data structure also can be used to implement list C, which can lower the practical complexity of HRAF.

The theoretical complexity of the HRAF apparently is not so promising especially when we compare it with EA[9], LVEA[21] and K*[10]. It is known that all the three algorithms have the same asymptotic complexities of $O(m + n \log(n) + k)$, for both time and space. We should point out that the complexities of HRAF are analyzed in the worst case, nevertheless when k is rather small, the complexities for both time and space are also comparable. For example, in practical Transportation Navigation System, the road map contains more than millions of nodes and edges, while the system only needs to provides several options to the users (maybe no more than 10) because much more selections for users mean lower usability. So the performance of the HRAF algorithm for

KSPs computation is considerable when k is very small. The following experiments demonstrate our standpoints.

6 Simulations

As we discussed above, K* is more efficient than EA and LVEA for single-pair KSPs computation according to their experiments, although the three famous algorithms have same theoretical complexities. Therefore, we compare HRAF only with K* in the simulations. We implement HRAF and K* using C++ in Microsoft Visual Studio 10.0. In our simulations, the graph data are first loaded into the main memory stored as an adjacency list in which we can access edges by node. The CPU time is obtained in milliseconds by using the C++ function GetSystemTime(). Note that we neglect the time needed for loading graph data into main memory, and the memory consumption is measured by PagefileUsage, which is part of process information returned by the function GetProcessMemoryInfo. We run all experiments on a desktop computer equipped with an Intel Pentium Dual-Core CPU (2GHz) and 2GB memory. No parallel implementations are taken into account in the experiments.

Our simulations of both algorithms are based on four maps: New York City, San Francisco Bay Area, Colorado and Florida, as shown in Table 5. The maps data are available from the home page of the 9th DIMACS Implementation Challenge[24]. We use airline distance, which is computed by the law of cosines, as a heuristic estimate for both algorithms. Four different pairs of start-goal nodes(from center of the map to the edge, or from a side to another side according to its coordinates) are randomly selected for each map, we then run the algorithms 20 times for each pair. In order to fully compare the above two algorithms, K* and HRAF, we first compare needed runtime using both algorithms when k is equal to one, then compare runtime when k is small (less than 50). The overall comparisons about runtime and space consumption of both algorithms are also listed.

Table 5: Datasets of 4 maps used in our simulations.

Map	Abbr.	No. Nodes	No. Edges
Colorado	COL	435,666	1,057,066
San Francisco	BAY	321,270	800,172
New York City	NY	264,346	733,846
Florida	FLA	$1,070,376$	$2,712,798$

6.1 Runtime Comparisons

For HRAF and K*, setting $k = 1$ means finding the shortest path in a map. In each map, four **s-t** nodes are randomly selected with 20 times running on both algorithms. Then we obtain an average runtime for each **s-t** path computation and finally the average runtime is calculated over four computations. All the following simulations use the same testing method.

For finding only the shortest path, the runtime comparison for both algorithms is listed in Table 6. We can conclude that HRAF outperforms K* by about 11% to 28% when we just compute one shortest path from related maps. This is due to the fact that HRAF just uses A* to compute shortest path tree and obtain the first shortest path while K* needs to establish a complicated path graph when it uses the A* search.

An overall runtime comparison is shown in Fig. 8. It takes more runtime when k increases for the both algorithms. However, HRAF always performs better in the NY map and BAY map, while when k is larger than 78 in COL map and 95 in FLA map, K* is the better choice.

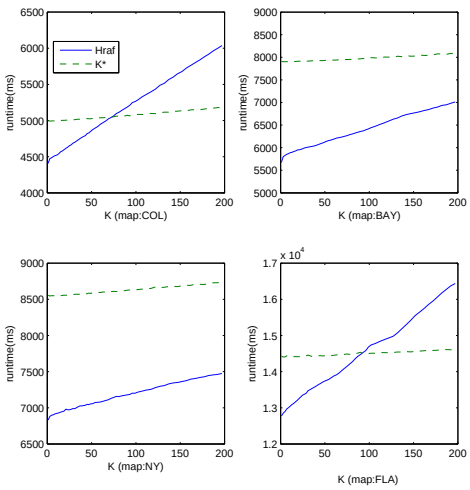

Figure 8: Runtime comparisons on four maps.

Table 7 shows the full notes of the execution of both algorithms with the detailed improvement when k is less than 50. HRAF is able to obtain an average improvement on the four maps with more than 12% to about 18%. Clearly, there exists a tendency that the improvement becomes smaller and smaller as the number of routes needed to be computed increases. Furthermore, we can see that different maps resulted in different performance, this is due to the difference of the four maps and the influence of random selections of the **s-t** node pairs in each map.

It is easy to conclude that the value of k has a

great influence on HRAF, i.e, the larger the value of k, the more runtime is needed to find the routes. However, K* is less sensitive to k, i.e, time consumed using K* changed little when k increased. It is clear that HRAF outperforms K* when k is less than a threshold, which depends on the number of nodes and the number of edges of the explored graph. When the number of nodes in the map increases, HRAF performs better with bigger k than K* does.

It is known that many practical applications do not need too many best path selections, just like Transportation Navigation System. Usually, it is enough for an algorithm to provide a limited number of optimal solutions, which is the key point of HRAF compared with K*.

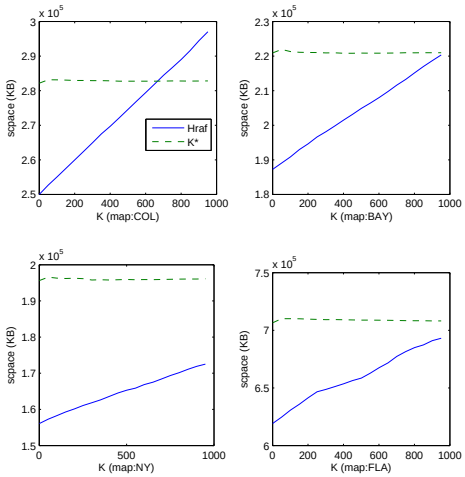

Figure 9: Space performance comparisons on four maps.

6.2 Space consumption comparison

Space consumption is also important for an algorithm when solving the KSP problem. Fig. 9 shows the space usage for both algorithms obtained when running once each for different value of k.

It is clear that HRAF is able to save about 15% space compared with K* when k is no more than 50. We can also conclude that the space consumption for HRAF grows proportionally when k increases in our experiments, just like runtime. Meanwhile, HRAF consumes less memory than K* when k is rather small. Notice that the worst-case space complexity of HRAF is $O(m+n+K^2)$, which means that the worst-case space required using HRAF grows quadratically with K. However, this worst-case bound is very hard to achieve, because the candidate path explored by i^{th} shortest path has at most $i-1$ $sidetrackedges$ and is much lower than i in practice.

7 Conclusions

There are many publications aiming to solve the single-pair KSP problem. We reported a new algorithm named HRAF, which operates on-the-fly strat-

Table 6: Runtime (ms) comparison when k = 1 with 4 randomly selected **s-t** pairs.

s-t	COL		BAY		NY		FLA	
	K*	HRAF	K*	HRAF	K*	HRAF	K*	HRAF
1	7291	6454	8030	6125	8729	7081	15738	13908
2	5620	4988	7966	5684	8415	6583	10714	9489
3	3589	3112	8020	5751	8437	6653	15638	13834
4	3485	3075	7615	5132	8641	7001	13861	15620
Avg	4996	4407	7908	5673	8555	6829	14428	12773
Ipv	11.8%		28.3%		20.2%		11.5%	

Table 7: Average runtime (ms) for finding less than 50 routes on four maps for both algorithms.

K	COL		BAY		NY		FLA		Avg.
	K*	HRAF	K*	HRAF	K*	HRAF	K*	HRAF	Ipv
1	4996.3	4407.3	7908.1	5673.0	8555.5	6829.6	14427.9	12773.2	17.9%
5	4994.1	4484.6	7900.3	5827.3	8550.3	6898.0	14407.2	12896.8	16.6%
20	5002.3	4596.8	7911.1	5950.2	8557.4	6955.4	14414.5	13210.7	15.0%
30	5009.0	4688.2	7915.0	6002.2	8563.0	6989.8	14440.0	13406.5	14.0%
40	5018.3	4767.0	7920.6	6043.8	8576.7	7025.8	14431.7	13582.1	13.2%
50	5025.1	4852.1	7928.2	6114.9	8581.5	7055.4	14435.4	13729.4	12.3%

egy and can be guided using heuristic estimates. We proved its correctness and determined its asymptotic worst-case complexities for runtime and space consumption. Our experimental results show that HRAF outperforms the famous KSP algorithm K* when k is small. The performance advantage compared to K* becomes smaller when k increases. When k arrives at a threshold (usually with a very large k), HRAF exhibits performance worse than K*.

Acknowledgements: This work was supported by the Fundamental Research Funds for the Central Universities (grant No.ZYGX2013J076), Sichuan Science and Technology Department (grant No.2015SZ0045) and National Science Foundation of China (grant No.61273308).

References:

[1] W. Hoffman, R. Pavley, A method of solution of the Nth best path problem, *Journal of the ACM*, Vol.6, No.4, 1959, pp.506-14.

[2] J.Y. Yen. Finding the K shortest loopless paths in a network, *Management Science*, Vol.17, No.11, 1971, pp.712-6.

[3] Y. Honma, M. Aida, H. Shimonishi, New routing Methodology Focusing on the Hierarchical Structure of Control Time Scale, *WSEAS Transactions on Communications*, Vol.13, No.1, 2014, pp.505-12.

[4] Berclaz J, Fleuret F, Turetken E, Multiple object tracking using k-shortest paths optimization. *IEEE Trans on Pattern Analysis and Machine Intelligence*, Vol.33, No.9, 2011, pp.1806-1819.

[5] Ozer B, Gezici G, Meydan C, et al, Multiple sequence alignment based on structural properties, *2010 5th International Symposium on Health Informatics and Bioinformatics (HIBIT)*, IEEE, pp.39-44.

[6] Y.K. Shih, S. Parthasarathy, A single source k-shortest paths algorithm to infer regulatory pathways in a gene network, *Bioinformatics*, 2011, Vol.28, No.12, pp.49-58.

[7] W. Xu, S. He, R. Song, Finding the K shortest paths in a schedule-based transit network, *Computers & Operations Research*, Vol.39, No.8, 2012, pp.1812-1826.

[8] X. Wan, L. Wang, N. Hua, et al, Dynamic routing and spectrum assignment in flexible optical path networks, *National Fiber Optic Engineers Conference*, Optical Society of America, 2011.

[9] D. Eppstein. Finding the k shortest paths, *SIAM J. Computing*, Vol.28, No.2, 1998, pp.652-73.

[10] H. Aljazzar, S. Leue, K*: a heuristic search algorithm for finding the K shortest paths, *Artificial Intelligenc*, Vol.175, No.18, 2011, pp.2129-54.

[11] A. Sedeno-Noda, An efficient time and space K point-to-point shortest simple paths algorithm, *Applied Mathematics and Computation*, Vol.218, No.20, 2012, pp.10244-57.

[12] A. Sedeno-Noda , J.J. Espino-Martin, On the K best integer network flows, *Computers & Operations Research*, Vol.40, No.2, 2013, pp.616-26.

[13] Z. Gotthilf, M. Lewenstein, Improved algorithms for the kshortest paths and the replacement paths problems, *Information Processing Letters*, Vol.109, No.7, 2009, pp.352-55.

[14] J. Hershberger, M. Maxel, S. Suri, Finding the k Shortest Simple Paths: a new algorithm and its implementation. *ACM Transactions on Algorithms*, Vol.3, No.4, 2007.

[15] A. Perko, Implementation of algorithms for K shortest loopless paths, *Networks*, Vol.16, No.2, 1986, pp.149-60.

[16] V.M. Jimenez, A. Marzal, Computing the k shortest paths: A new algorithm and an experimental comparison, *Lecture Notes in Computer Science*, Vo.1668, 1999, pp.15-29.

[17] Y.L. Chen, H.H. Yang, Finding the first K shortest paths in a time-window network, *Computers & Operations Research*, Vol.31, No.4, 2004, pp.499-513.

[18] L.R. Nielsen, K.A. Andersen, D. Pretolani, Finding the K shortest hyperpaths, *Computers & Operations Research*, Vol.32, No.6, 2005, pp.1477-97.

[19] H.H. Yang, Y.L. Chen, Finding K shortest looping paths in a traffic-light network, *Computers & Operations Research*, Vol.32, No.3, 2005, pp.571-81.

[20] E. Martins, M. Pascoal, J. Santos, A new algorithm for ranking loopless paths, *Technical report*, Universidade de Coimbra, Portugal 1997.

[21] V.M. Jimenez, A. Marzal, A lazy version of Eppstein's shortest paths algorithm, *Lecture Notes in Computer Science*, Vol.2647, 2003, pp.179-90.

[22] E.W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik*, Vol.1, No.1, 1959, pp.269-71.

[23] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, *Addision Wesley*, 1986.

[24] The 9th DIMACS implementation challenge: The shortest path problem, University of Rome. *http://www.dis.uniroma1.it/ challenge9/*, 2006.