PAPER

Finding Useful Detours in Geographical Databases

Tetsuo Shibuya[†], *Nonmember*, **Hiroshi Imai**[††], *Member*, **Shigeki Nishimura**[†††], **Hiroshi Shimoura**[†††], *and* **Kenji Tenmoku**[†††], *Nonmembers*

SUMMARY

In geographical databases for navigation, users raise various types of queries concerning route guidance. The most fundamental query is a shortest-route query, but, as dynamical traffic information newly becomes available and the static geographical database of roads itself has grown up further, more flexible queries are required to realize a user-friendly interface meeting the current settings. One important query among them is a detour query which provides information about detours, say listing several candidates for useful detours.

In this paper, we first review algorithms for the shortest and k shortest paths, and discuss their extensions to detour queries. Algorithms for finding a realistic detour are given. The efficiency and property of the algorithms are examined through experiments on an actual road network.

key words: Geographical databases, car navigation, shortest paths, detour query

1. Introduction

In geographical databases, various types of queries arise, especially for spatial and topological queries. One such typical query is a shortest-path query, which is crucial for car navigation, etc. Recently, as dynamical traffic information newly becomes available such as ATIS (Advanced Traffic Information Service), and VICS (Vehicle Information & Communication System) in Japan, more sophisticated queries come to be required. Also, the static geographical database of roads itself has grown up further, and similarly in this respect advanced types of queries are necessary to realize a user-friendly interface meeting the current circumstances. One important query among them is a detour query which provides information about detours; for example, enumerating several candidates for useful detours.

From the algorithmic viewpoint, the shortest path problem itself has been studied very well for a long time. For example, the Dijkstra method is the most famous and traditional algorithm for this problem. To make this algorithm more efficient, many algorithms has been considered, such as the A* algorithm, the bidirectional Dijkstra method, and the bidirectional A* algorithm, which are often cited as AI (Artificial Intelligence) search techniques [2]–[5], [9].

The k shortest paths problem is the generalization of this shortest path problem. If you need "good" solutions other than the optimal one, or optimal solution under certain constraints, it is a good news that you can find the k shortest paths quickly. Hence, this problem is also very applicable in many fields, such as network connection routing, finding a detour in navigation systems, DNA alignment [8], etc. This problem is also studied very well, and, recently, Eppstein has proposed a very efficient algorithm for this. According to this, for a directed graph with n vertices and non-negative m edges, the k shortest paths from one source to l destinations are known to be obtained in $O(m+n\log n+lk)$ time, or in sorted form in $O(m+n\log n+lk\log k)$ time [1].

But this original algorithm requires searching with the Dijkstra method from the source to the other all vertices. Thus, reduction of the searched region is desired, especially in case the graph is very large. This paper extends the Eppstein's algorithm, using upper bound of the length of the suboptimal paths and the technique of bidirectional A* algorithm, to reduce the searched region in the 2-terminal k shortest paths problem.

This paper then discusses the detour problem as one of its applications. The 'detour' is a suboptimal path which is short but overlaps little with the shortest path. But this concept is ambiguous. Because of this ambiguity, the detour problem is not so studied as the shortest path problem. Hence this paper defines 'detour' precisely, and proposes algorithms for finding a realistic detour based on previous algorithms.

Then the efficiency of these algorithms are examined through experiments on an actual road network from a geographical database.

2. Preliminaries

In this paper, let the graph in assumption be a directed graph, $G=(V,E)$, in which $l(v,w)$ is the length of the edge (v,w) which is always non-negative, and s be the

Manuscript received 1998

Manuscript revised 1998

[†]The author is with IBM Tokyo Research Laboratory. This work was done while the author was at Department of Infomation Science, University of Tokyo.

[††]The author is with Department of Information Science, University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-0033, Japan

[†††]The authors are with Sumitomo Electric Industries, Ltd., Osaka, Japan

source and t be the destination when we consider the shortest path problem or the k shortest path problem. Add to this, let $d(u, v)$ be the shortest path length from u to v, n be $|V|$ and m be $|E|$.

2.1 Dijkstra Method

The Dijkstra method is a basic algorithm to solve the shortest path problem. The following is the outline of the Dijkstra method for two-terminal problem:

1. Let S be the empty set, and $p_s(v)$, the potential of a vertex v, be $+\infty$ except for the source s. Let $p_s(s)$ be zero.
2. Find the vertex v_0 which has the minimum potential in $V - S$ and add v_0 to S. If v_0 equals to t, halt.
3. For all vertices v such that (v_0, v) is in E, if $p_s(v_0) + l(v_0, v)$ is less than $p_s(v)$, replace the path from s to v with the path from s to v_0 and the edge (v_0, v), and let $p_s(v)$ be $p_s(v_0) + l(v_0, v)$.
4. Go to step 2.

In this algorithm, all the searched vertices has the information of the temporary shortest path from the source s, and the potentials of the searched vertices denote the length of the path. If we do not halt until $|S|$ be zero, we can get the shortest path tree from the source. This algorithm searches all directions regardless of where the destination is, which is a disadvantage of the Dijkstra method.

2.2 A* Algorithm

The A* algorithm finds the shortest path from source s to destination t efficiently, using a heuristic estimate for the length of the shortest path to the destination t, which is not longer than the actual shortest path to the destination, if the estimate can be given for each vertex[3][9]. The outline of the algorithm is as follows:

1. Let S be the empty set, and $p_s(v)$, the potential of a vertex v, be $+\infty$ except for the source s. Let $p_s(s)$ be zero.
2. Find the vertex v_0 which has the minimum value of $p_s(v) + h_s(v)$ in $V - S$ and add v_0 to S. If v_0 equals to t, then halt.
3. For all vertices v such that (v_0, v) is in E, if $p_s(v_0) + l(v_0, v)$ is less than $p_s(v)$, replace the path from s to v with the path from s to v_0 and the edge (v_0, v), and let $p_s(v)$ be $p_s(v_0) + l(v_0, v)$, and remove v from S if v is in S.
4. Go to step 2.

In this algorithm, $p_s(v)$ also denotes the length of the temporary shortest path from s to v. $p_s(v) + h_s(v)$ is the temporary estimate for the shortest path from s to t via v. The searched vertices by the A* algorithm is always within searched vertices by the Dijkstra method. In this way, the A* algorithm can get the shortest path more effectively.

In the A* algorithm, the shortest path from s may not appear first, and a shorter path may be found in the future search, which is the reason of the removal of vertices from S in step 3. It makes this algorithm rather inefficient. This can be avoided if the estimator is dual feasible. The definition of 'dual feasible' is as follows:

Definition 1: The estimator h_s for the shortest path to t is called dual feasible if and only if h_s satisfies the following constraint:

$$\forall (u, v) \in E \quad l(u, v) + h_s(v) \geqq h_s(u) \tag{1}$$

For example, Euclid distance between v and t can be used for a dual feasible estimator in a graph of a road network.

If the estimator is dual feasible, the A* algorithm can be easily translated to the Dijkstra method by modifying the length of the edges [2]:

Theorem 1: Let h_s be a dual feasible estimator for s. The Dijkstra method on a graph in which the length of the edge (u, v), or $l(u, v)$ is replaced by $l'(u, v)$ as follows is equivalent to the A* algorithm on the original graph.

$$l'(u, v) = l(u, v) + h_s(v) - h_s(u) \tag{2}$$

2.3 Bidirectional Methods

The Dijkstra method and the A* algorithm are unidirectional algorithms. This means one of the two vertices, that is the source or the destination, plays less role than the other. The bidirectional Dijkstra method solves this problem[4][5].

In the bidirectional Dijkstra method, searches are done both from the source and destination using the Dijkstra method. In outline, the algorithm is like the following:

1. Let S and T be the empty set, and the potentials $p_s(v)$ for s and $p_t(v)$ for t be both $+\infty$, except for $p_s(s)$ and $p_t(t)$. Let $p_s(s)$ and $p_t(t)$ be zero.
2. Add vertex v_0 which has the smallest potential for s in $V - S$. Then, go to step 7 if v_0 is in T.
3. For all vertices v such that (v_0, v) is in E, if $p_s(v_0) + l(v_0, v)$ is less than $p_s(v)$, replace the path from s to v with the path from s to v_0 and the edge (v_0, v), and let $p_s(v)$ be $p_s(v_0) + l(v_0, v)$.
4. Add vertex v_0 which has the smallest potential for s in $V - T$. Then, go to step 7 if v_0 is in S.
5. For all vertices v such that (v, v_0) is in E, if $l(v, v_0)$ $+p_t(v_0)$ is less than $p_t(v)$, replace the path from v

to t with the edge (v, v_0) and the path from v_0 to t, and let $p_t(v)$ be $l(v, v_0) + p_t(v_0)$.

6. Go to step 2.
7. Find the edge (u, v) minimizing $p_s(u) + l(u, v) + p_t(v)$ such that u is in S and v is in T. The shortest path from s to t consists of the path from s to u and the edge (u, v) and the path from v to t if $p_s(u) + l(u, v) + p_t(v)$ is less than $p_s(v_0) + p_t(v_0)$, otherwise it consists of the path from s to v_0 and the path from v_0 to t.

The paths which pass vertices in neither S nor T are always longer than $p_s(v_0) + p_t(v_0)$. Thus, the obtained path is guaranteed as the shortest path.

If the graph is homogeneous, the number of vertices searched with this algorithm is half of that with the Dijkstra method, because these searched vertices are in two cycles whose radii in this algorithm are half of the radius in the unidirectional Dijkstra method.

But this bidirectional Dijkstra method searches all directions regardless of where the other terminal is. If we have dual feasible estimators for both of the source and the destination, we can overcome this disadvantage, using a technique like in Theorem 1 based on the following theorem:

Theorem 2: By searching the shortest path with the bidirectional Dijkstra method on the graph in which the length of edge (u, v), or $l(u, v)$ is replaced by the following $l'(u, v)$, we can get the shortest path. Let h_s and h_t be dual feasible estimators for source s and destination t.

$$l'(u,v) = l(u,v) + \frac{1}{2}(h_s(v) - h_s(u)) + \frac{1}{2}(h_t(u) - h_t(v)) \ . \quad (3)$$

In this algorithm, the forward search is equivalent to that of the A* algorithm using estimator of $(1/2)(h_s(v) - h_t(v))$, and backward $(1/2)(h_t(v) - h_s(v))$.

2.4 Eppstein's Algorithm

As for finding the k shortest simple paths (*i.e.* paths without cycles), the best known bound is $O(k(m + n \log n))$ [7]. On the other hand, Eppstein proposed an algorithm which finds the k shortest paths implicitly regardless of cycles, in time $O(m + n \log n + k)$, or $O(m + n \log n + k \log k)$ if the output paths are sorted[1]. We discuss the latter algorithm here.

At first, we define $\delta(u, v)$ for the edge (u, v) as follows:

$$\delta(u, v) = l(u, v) + d(v, t) - d(u, t) \quad (4)$$

This $\delta(u, v)$ means how much longer it will take than the optimal way if we go to the edge (u, v), and therefore this value is always non-negative.

If we search by the Dijkstra method from the destination t, a shortest path tree to t can be made. If an edge (u, v) is on this tree, $\delta(u, v)$ is zero. If an edge is not on the shortest path tree, it is called a sidetrack. If we go along a s-t path p other than the shortest path, there must be sidetracks on the path, and we define $sidetrack(p)$ as the nearest sidetrack to t within them.

We can suppose a heap, in which the parent of a path p is a path which is same as p until $sidetrack(p)$ and go along the shortest path instead of going to $sidetrack(p)$. We define this parent of p as $parent(p)$. The root of the heap is the shortest path, and all the path from s to t appear in the heap once. In this heap, p is $\delta(sidetrack(p))$ longer than $parent(p)$.

We call p-heap if the node of the heap has only p children at most. The basic concept of the Eppstein's algorithm is to modify this path heap to 4-heap. Once the 4-heap has made, we can get the k shortest paths in $O(k)$ time, or $O(k \log k)$ time if we sort the output paths[6]. The following is the outline of this algorithm:

1. Make the shortest path tree from all the vertices to t by the Dijkstra method.
2. For each vertex v, construct $H_G(v)$, that is, a 3-heap of sidetracks (u, u'), such that u is on the shortest path from v to t, ordered by $\delta(u, u')$ defined above, as follows:
   a. For each vertex v, make $H_{out}(v)$, that is a 2-heap in which the root has only one child, of sidetracks (v, v') ordered by $\delta(v, v')$.
   b. For each vertex v, make $H_T(v)$, that is, a 2-heap of vertices on the shortest path from v to t ordered by the value δ of the root in the heap made in step 2-(a). For the detail of this step, see [1]. Then merge $H_{out}(v)$ and $H_T(v)$ to make $H_G(v)$.
3. For each v in G, make a pointer from each node in $H_G(v)$, which represents a sidetrack (u, u') in G, to the root of $H_G(u')$, and define the length of this new edge as the value of the root.
4. Make a node for each v in G, and make a pointer from this new node to the root of $H_G(v)$. Let the length of the new edge be the value of the root. Let this new graph be P(G).
5. We can find a heap $H_v(G)$ in $P(G)$ for any v, thinking the root as the node made in step 4 for v, and the value of a node as the length from the root to the node. Find the k smallest nodes in this virtual heap $H_v(G)$. There is a one-to-one correspondence between the nodes in $H_v(G)$ and the paths from v to t in G, and we can easily restore the path from the node of the heap.

This algorithm is very efficient for finding the k shortest paths from all the vertices to one destination,

or one source to the other vertices. But, for the 2-terminal problem, this algorithm may search much more vertices than necessary.

3. New Approach for the 2Terminal k Shortest Path Problem

3.1 How to Use A* Algorithm

How to use A* Algorithm in computing the k shortest path in 2-terminal problem is discussed in this section.

If a dual feasible estimator is given, the replace of the length of an edge $l(u,v)$ with $l'(u,v)$ described at (2) in Theorem 1 does not change the k shortest paths:

Theorem 3: The k shortest paths from s to t on a graph in which the length of the edge (u,v), or $l(u,v)$ is replaced by $l'(u,v)$ as in (2) are same as those on the original graph.

Proof: Let p be a path from s to t, and h_s be a dual feasible estimator for s. Then the length of p, or $length'(p)$ in new graph is described by the length of p, or $length(p)$ in the original graph as follows:

$$\begin{aligned} length'(p) &= \sum_{(u,v)\in p} l'(u,v) \\ &= \sum_{(u,v)\in p} l(u,v) + h_s(t) - h_s(s) \\ &= length(p) + h_s(t) - h_s(s) \end{aligned} \tag{5}$$

According to this, all the paths on the new graph from s to t are $h_s(s) - h_s(t)$, which is constant, shorter than those on the original graph. This means the k shortest paths on the new graph are same as those on the original graph. □

The same is the case with $l'(u,v)$ of (3) in Theorem 2:

Theorem 4: The k shortest paths from s to t in a graph, in which the edge length $l(u,v)$ is changed to $l'(u,v)$ as in (3), are same as those in the original graph.

Proof: All the paths from s to t in the modified graph are only a constant shorter than in the original:

$$\begin{aligned} length'(p) &= \sum_{(u,v)\in p} l'(u,v) \\ &= \sum_{(u,v)\in p} l(u,v) + h \\ &= length(p) + h \end{aligned} \tag{6}$$

h is constant, that is $(h_s(t) + h_t(s) - h_s(s) - h_t(t))/2$. This means the k shortest paths are same in the two graphs. □

Thus we can use either unidirectional or bidirectional A* algorithm implicitly by changing the length of the edges.

3.2 Unidirectional Method

Letting p_{opt} be the shortest s-t path, we define $\Delta(p)$ for a s-t path p as $length(p) - length(p_{opt})$. If we can use the upper bound of this $\Delta(p)$, or $\bar{\Delta}$, where p is within the k shortest paths, we can easily reduce the searched vertices.

When we do not need paths a constant longer than the shortest one or we only want to list the paths a constant longer than the shortest one at most, we should let $\bar{\Delta}$ be this constant. If we can know approximate value of $\bar{\Delta}$ by experience or other methods or really know the value, we can use it.

The algorithm is very simple, and the outline of this is as follows:

1. If a dual feasible estimator for t is given, replace length of each edges as in Theorem 3.
2. Search from t by the Dijkstra method until the shortest path from s to t is discovered.
3. Search successively until a vertex v from which the shortest path to t is more than $\bar{\Delta}$ longer than that from s.
4. Find the k shortest paths in the searched region, by Eppstein's algorithm.

If a path passes a vertex v not in the searched region, this path is longer than $length(p_{opt}) + \bar{\Delta}$ because the shortest path from v to t is longer than it. So, all the obtained k paths are shorter than $length(p_{opt}) + \bar{\Delta}$, which is supposed to be longer than the actual kth shortest path. Thus, the obtained k paths are the actual k shortest paths.

3.3 Bidirectional Method

The Eppstein's algorithm can be used both on the shortest path tree from the source and on that to the destination. Thus it is a natural idea to use a bidirectional method for finding the k shortest paths. But it needs some modification to the Eppstein's algorithm.

The heaps obtained by the Eppstein's algorithm are not enough if we use a bidirectional method. To solve this problem, we propose a new path heap graph involving another heap called H_{mid}. We also use $\bar{\Delta}$ defined at 3.2.

In the algorithm, let S and T be sets of searched vertices from s and t by bidirectional Dijkstra method, p_s and p_t be potentials, d_s be the length of the shortest path from s to the last vertex added to S, and d_t be the length of the shortest path from the last vertex added to T to t. The following is the outline of the algorithm:

1. If a dual feasible estimator for s and that for t are given, change the length of the edges as in Theorem 4.

2. Search by the bidirectional Dijkstra method both from s and t, until the shortest path (p_{opt}) is discovered.

3. Continue searching until $d_s + d_t$ is longer than $length(p_{opt}) + \bar{\Delta}$.

4. Let set of edges F be $\{(u,v)|(u,v) \in E, u \in S, v \in T - S\}$, and set of vertices U be $\{v|(u,v) \in F\}$. (*see* Figure 1) Construct heap H_{mid} of vertices in U ordered by the value $q(v) = p_s(v) + p_t(v)$. Note that v is not in S but $p_s(v)$ is not $+\infty$. In this step, if $q(v)$ is larger than $length(p_{opt}) + \bar{\Delta}$, we can ignore v.

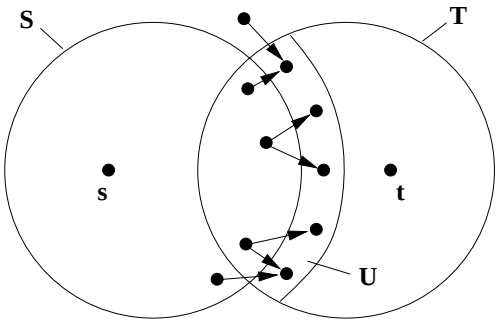


Fig. 1 Situation of U

5. Construct Eppstein's path heap graph on the shortest path tree from s, only in $G_s = (S + U, \{(u,v)|u \in S, v \in S + U\})$ and on that to t only in $G_t = (T, \{(u,v)|u,v \in T\})$. Then let the heap found in the graph be $H_s(v)$ and $H_t(v)$.

6. For each v in U, make a new edge from a node in H_{mid} which represents v, to the roots of $H_s(v)$ and $H_t(v)$, whose length is the value δ of the roots. Then think as if there is an edge from each node in $H_s(v)$ to the root of $H_t(v)$. (This pointer is decided when the path heap graph is searched.) The length of it is also δ of the root of $H_t(v)$. Note that the completed virtual heap is 5-heap.

7. Find the k shortest paths from the root of the entire heap by the same way as in Eppstein's algorithm.

Notice that, in step 5, we have to construct H_t only in T. It is because once a s-t path has gone out from S, the rest of the path cannot pass any vertices out of T, because $d_s + d_t$ is guaranteed to be larger than any path within the k shortest paths.

In step 4, H_{mid} can be constructed by the modification of the heap used for searching from s in step 2 and 3, which can be done in $O(n)$ time. Add to this, step 6 can also be done in $O(n)$ time.

4. Detour Problem

'Detour' is a path which is short but overlaps little with the shortest path. To find it is very important in route navigation systems, ATM network, and so on. We discuss how to gain this detour based on the above algorithms.

4.1 How to Compute Overlapping Length

In searching a detour, the overlapping length of a detour with the shortest path is an important factor. Let this value for a path p be $overlap(p)$. This length can be computed very fast for every path encountered in searching in the path heap graph. The following is the outline of this procedure:

1. For each vertex v in V, compute the length of the part of the v-t shortest path which overlaps with the s-t shortest path. This can be done in $O(n)$ time, by the depth first search on the shortest path tree to t for example. Then, let this value be $ov(v)$.
2. For each sidetrack (u,v), compute following $\delta'(u,v)$:

$$\delta'(u,v) = ov(v) - ov(u) \tag{7}$$

3. When we search in the path heap and obtained a path p, we compute the $overlap(p)$ as follows. Let q be $parent(p)$ and e be $sidetrack(p)$.

$$overlap(p) = overlap(q) + \delta'(e) \tag{8}$$

Note that the same technique can be easily done in the bidirectional method described in 3.3.

4.2 Definition of 'Detour'

'Detour' is not so clear concept. Thus we must define it precisely. In easiest way, we can define it as follows for example:
Definition 2: 'Detour' is the shortest path which has overlap, which is shorter than the half of the shortest path length, with the shortest path length.
But this definition requires searching the path heap in order until a desired path will be found, which means it takes $O(k \log k)$ time in checking k paths, and setting $\bar{\Delta}$ is difficult. Add to this, the obtained detour by this definition may branch off and join the shortest path many times. In the car navigation system, such a detour is not desirable.

Taking these things into consideration, we define 'detour' as following:
Definition 3: 'Detour' is $\bar{\Delta}$ longer than the shortest path at most, branch off and join the shortest path only once, and has the smallest overlap with the shortest path among such paths. If several paths satisfies these constraints, choose the shortest one.

Notice that the detour defined in either Definition 2 or 3 has no cycles unless there exists a zero-length cycle in the graph.

4.3 How to Obtain Detour

We discuss how to get the detour defined in Definition 3, in this section.

4.3.1 Unidirectional method

The method to search the paths which branch off and join the shortest path only once in the path heap is as follows: If u of $(u, v) = sidetrack(p)$ is on the shortest path tree to t and $parent(p)$ is not the shortest path, or p_{opt}, we only have to search children of p in H_T. This technique can be also applied in the bidirectional method, and note that listing paths which branch off and join the shortest path i times can be done with a similar method.

Add to this, notice that, if there is longer overlap from s to $sidetrack(p)$ along p than the temporary shortest overlap, we also have to search children of p only in H_T. To make this technique more efficient, we should not search the children of p_{opt} in $H_T(s)$ from the root of it, but search $H_{out}(v)$ and its children from s to t along p_{opt}. The same technique can be used in finding the detour defined in Definition 2. But this technique is difficult to use with the bidirectional method.

4.3.2 Bidirectional method

If the length of the edges are all integers, and $\bar{\Delta}$ is not very large, we can compute the detour efficiently based on the bidirectional method in 3.3. The outline of this algorithm is as follows:

1. Construct H_s and H_t as in 3.3.
2. Let v_{mid} be the nearest vertex to s along p_{opt} in U. For each vertex v in U except for v_{mid}, do the following:
   a. If $q(v)$ is larger than $length(p_{opt}) + \bar{\Delta}$, skip (b) and (c).
   b. Let $\bar{\Delta}'_v$ be $\bar{\Delta} - q(v)$. Search $H_s(v)$ for s-v paths which branch off p_{opt} only once as in *4.3.1*, and not more than $\bar{\Delta}'_v$ longer than the shortest s-v path in G_s. At the same time, make a table T_v whose size is $\bar{\Delta}'_v + 1$ and fill in $T_v[i]$ the path whose overlapping length with p_opt is the shortest among the paths *s.t.* $\Delta(p) \leq i$.
   c. Search $H_t(v)$. Note that, for some i, if there is no path in $T_v[i]$ or the overlapping length of the path in $T_v[i]$ is longer than the temporary shortest overlapping length, or l, we only have to search paths less than $l - i$ longer than the v-t shortest path.
3. If the length of the v_{mid}-t shortest path in G_t, or l, is shorter than the temporary shortest overlapping length, find the detour between s and v_{mid} in G_s using $\bar{\Delta}'_v - l$ as $\bar{\Delta}$.
4. If the length of the s-v_{mid} shortest path in G_s, or l, is shorter than the temporary shortest overlapping length, find the detour between v_{mid} and t in G_t using $\bar{\Delta}'_v - l$ as $\bar{\Delta}$.

In step 2, we can search either $H_s(v)$ or $H_t(v)$ first. To make this technique more efficient, we should sort v in U ordered by $q(v)$, and search from the vertex of which $q(v)$ is large. This is because if $q(v)$ is larger, the overlapping length tends to be shorter. In step 3 and 4, use the techniques in 4.3.1.

4.3.3 Method for the planar graph

A planar graph has a good feature. If $overlap(p)$ is larger than $overlap(parent(p))$, there are three cases. If p_{opt} and $parent(p)$ are like in Figure 2, p must take the form of (a), (b) or (c). If p takes the form of (a), we have to search the children of p only in H_T.

Determining which form p takes is a little difficult. But, fortunately, in most cases p takes the form of (a). It means, if we do not require the exact detour defined in Definition 3, we can ignore the cases (b) and (c).

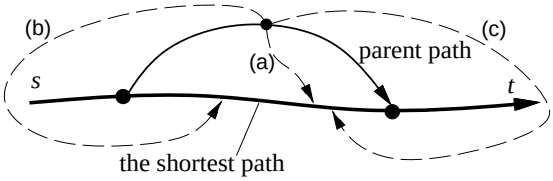


Fig. 2 Searching paths in a planar graph

The road network is not a planar graph, but property of it is similar to it. Hence, this technique may have some effect on the road network.

5. Experiments on Road Network

In this section, we investigate the efficiency of algorithms described above through experiments on the actual road network of 160 kilometers times 80 kilometers region in Tokyo metropolitan area. As the length of each edge, we used the necessary time (seconds) to pass the edge, which is rounded off to an integer, rather than the distance along the edge. Accordingly, we used Euclid distance to t and from s divided by the maximum speed, that is 105 kilometers per hour, as h_s and h_t.

5.1 Property of the Road Network

Figure 3 shows Δ of the kth shortest path in case of Hamadayama to Hongo. According to this, Δ increases

in proportion to $\log k$. This may help the estimation of $\bar{\Delta}$ used in Section 3.

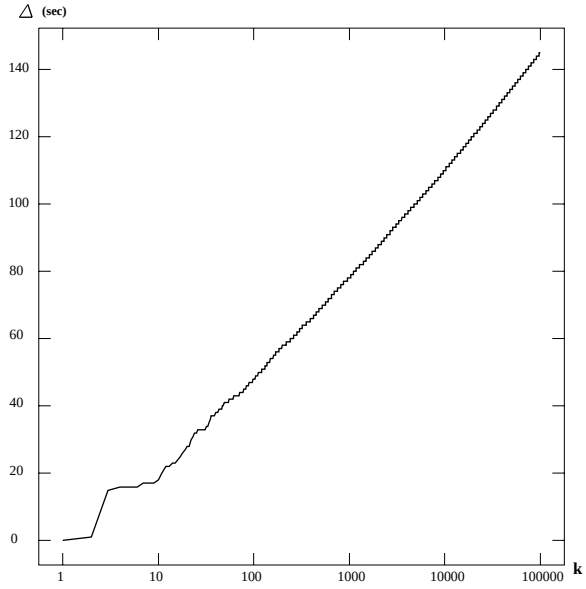


Fig. 3 Relation between the k and Δ

In the algorithms in Section 4, we search paths regardless of cycles. If there are many paths with cycles among suboptimal paths, the algorithms may not work well, because the detour defined by Definition 3 has no cycle. Table 1 shows the ratio of paths with cycles in the suboptimal paths whose Δ's are smaller than 100, in cases (a) Yumenoshima - Hongo (b) Narita - Ichihara and (c) Sayama - Chosi.

Table 1 The number of paths within $\Delta < 100$

s-t	d(s,t)	♯ paths	♯ paths with cycles (%)
(a)	1010	780299	14828 (1.9%)
(b)	2851	202531	17857 (8.8%)
(c)	8769	674630	111644 (16.5%)

According to this table, the paths with cycles are relatively rare among the suboptimal paths. This table also shows that the number of suboptimal paths increases in cases such that the paths passes the urban areas or the distance between the terminals is large. The example (a) is the former case and (c) is the latter case.

5.2 Efficiency of the Extended Eppstein's Algorithm

Table 2 shows the number of the searched nodes in finding the k shortest path in case of Hamadayama to Matsudo. Note that (f) means forward search and (b) means the backward search. In this experiment, we used the actual Δ of the kth shortest path as $\bar{\Delta}$, so that we can evaluate the best case of these algorithms.

Table 2 Searched nodes by various algorithms

k		1	10	100	1000	10000
length		2165	2170	2184	2203	2224
Dij.	(b)	11162	11203	11305	11422	11545
A*	(b)	5816	5871	6034	6252	6542
Bi-Dij.	(f)	2812	2812	2882	3011	3135
	(b)	2794	2794	2874	3012	3150
Bi-A*	(f)	1325	1325	1388	1557	1752
	(b)	1352	1352	1402	1555	1743

According to this table, the number of the searched nodes by the bidirectional method is about half of that by the unidirectional method in both of the cases using A* algorithm and Dijkstra method. On the other hand, the number of them using A* algorithm is also about half of that not using A* algorithm in both of the cases unidirectional and bidirectional. Accordingly, the algorithm using the bidirectional A* algorithm is far better than those of the others. But, as k increase, the effect of the bidirectional A* algorithm seems to decrease, so it may not so efficient if k is enormously large. To sum up, using the bidirectional A* algorithm is best, as long as k is not much larger than the size examined in this experiment.

5.3 Efficiency of the algorithms for the Detour Problem

Table 3 shows the efficiency of algorithms in cases (1) Hamadayama-Hongo and (2) Sayama-Matsudo. In the table, (a) is the number of paths at most $\bar{\Delta}$ longer than p_{opt}, (b) is that of paths who branch off and join p_{opt} among them, (c) is that of searched paths by the algorithm in *4.3.1*, (d) and (e) are those of searched paths by the algorithm in *4.3.2*. In (d), we searched H_s first, and in (e), H_t first. (f) is the overlapping length with p_{opt} of the obtained detour and its ratio to the length of p_{opt}.

Table 3 Searched paths by various algorithms

(1) Hamadayama - Hongo ($length(p_{opt}) = 891$)

$\bar{\Delta}$	20	40	60	80	100	120
(a)	11	48	261	1180	5115	20246
(b)	9	35	185	869	3880	15348
(c)	1	6	67	375	1979	8109
(d)	2	14	97	362	1542	5883
(e)	2	28	94	237	891	3115
(f)	157 (17.6%)	157 (17.6%)	58 (6.5%)	58 (6.5%)	58 (6.5%)	58 (6.5%)

(2) Sayama - Matsudo ($length(p_{opt}) = 3256$)

$\bar{\Delta}$	20	40	60	80	100	120
(a)	466	7174	61544	418016	2406750	12474190
(b)	108	324	1078	3259	12638	63811
(c)	107	273	954	3004	11973	37466
(d)	69	136	451	1290	5395	8689
(e)	69	136	451	1295	5379	8670
(f)	3159 (97.0%)	1837 (56.4%)	1837 (56.4%)	1837 (56.4%)	1753 (53.8%)	1201 (36.9%)

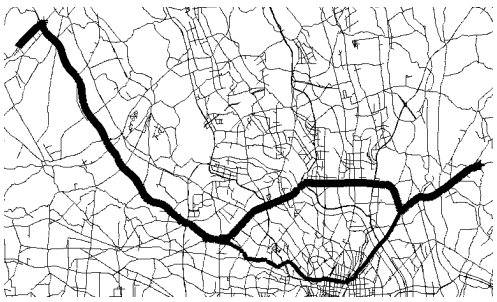
According to 5.1, the farther the 2 terminals are, the more suboptimal paths exists. In algorithm in 4.3.2, if we search H_s first, the searched paths in H_t is reduced, and vice versa. Add to this, if $\bar{\Delta}$ is large in compared with the distance of two terminals, the vertices in U may be nearer to t than to s. It means that we should search H_t first in such a case. This can be seen in the case (1) especially when $\bar{\Delta}$ is larger than 80 seconds. But in case of (2), such phenomenon is not seen because the distance of the 2 terminals is much larger.

According to the table, in general, the best method is to use the algorithm in *4.3.2* and search H_t first, but there are some cases that another algorithm is better.

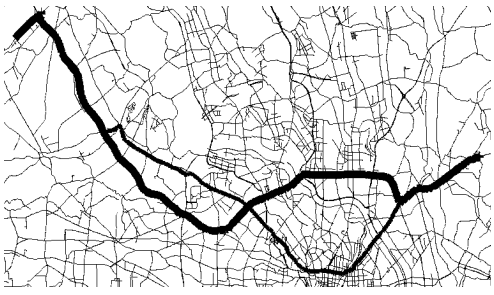
Figure 4 shows the obtained detour from Sayama to Matsudo when $\bar{\Delta}$ is 100 seconds and 120 seconds. In the figure, the thickest line is the shortest path, and the relatively thinner line which branch off it is the obtained detour. Note that the left terminal is Sayama and the other Matsudo.

In a road network, intersections cannot be nodes in the graph, for the reasons that costs of turning left or right or going straight in intersections differ, we cannot make U-turn in most intersections, and so on. Accordingly, we let a node in the graph be one side of a road segment between intersections. Hence, in a road network, the detour defined in Definition 3 can cross the shortest path at intersections.

The road network is not a planar graph, but we must examine the effect of the technique in *4.3.3* because the road network may have a feature similar to a planar graph.



(1) $\bar{\Delta} = 100$



(2) $\bar{\Delta} = 120$

Fig. 4 Detour between Sayama and Matsudo

Table 4 shows the effect of this technique. This table shows the number of searched paths in cases of (1) Hamadayama to Hongo and (2) Sayama - Matsudo (same cases as in Table 3) by the algorithms (a) unidirectional method and (b) bidirectional method searching H_t first.

Table 4 Effect of the technique in 4.3.3.

	$\bar{\Delta}$	20	40	60	80	100	120
(1)	(a)	1	6	67	374	1952	8000
	(b)	2	26	81	190	580	1658
(2)	(a)	107	261	808	2443	10290	31316
	(b)	68	133	385	1018	4472	24321*

* Failed finding the optimal solution

According to this, in most cases, the optimal detour is obtained, and the number of the searched paths are decreased. But in computing the detour from Sayama to Matsudo when $\bar{\Delta}$ is 120 by the algorithm (b), we failed finding the optimal one but the same one as when $\bar{\Delta} = 100$. Figure 4 shows the reason of this. The detour crosses the shortest path when $\bar{\Delta}$ is 120. Such paths are difficult to obtain with this technique. Add to this, we searched with this technique much more paths than without it in this case. It is because we failed to find the path which has little overlap with the shortest path, and as a result of it, we could not cut away the unnecessary paths in searching.

Thus, it is not strongly recommended to apply this technique to the road network. But if we deal with a planar graph, we should take this technique into consideration.

6. Conclusions

In this paper, we first reviewed algorithms for the shortest path problem such as the Dijkstra method, A* algorithm, bidirectional Dijkstra method, and bidirectional A* algorithm, and also discussed about Eppstein's algorithm which is for the k shortest paths problem.

Based on these algorithms, we proposed how to improve Eppstein's algorithm for the 2-terminal k shortest paths problem, using the upper bound of the length of the kth shortest path. We first showed we can use the A* algorithm in finding the k shortest paths both in the unidirectional method and in the bidirectional one. Then, we extended the Eppstein's algorithm to use with the bidirectional methods.

For the application of these algorithms, we discussed on the detour problem. This problem is a problem of finding a short path which overlaps little with the shortest path. First, we defined what the 'detour' is, and then proposed the algorithms to find it. We described a technique when we use the unidirectional method, then another algorithm using the bidirectional method which can be used under certain constraints, and an approximate method for the problem in a planar graph.

These algorithms are examined in the experiments in the actual road network. For the k shortest path problem, if a proper upper bound of the kth shortest path length are given, the extended Eppstein's algorithm using bidirectional A* algorithm showed the best performance. For the detour problem, the method using the bidirectional method is more efficient than the other algorithms in most cases. In this problem, the approximate technique for a planar graph is not so effective for the road network which is not a planar graph.

Constructing an efficient algorithm for the 2-terminal k shortest path problem without using the value $\bar{\Delta}$, and algorithms fit for the road network, which is constructed on hierarchical structures, for various problems like described above are remained as a future work.

The algorithmic results in this paper surely enhance the advanced use of geographical databases.

References

[1] D. Eppstein. "Finding the k Shortest Paths", *SIAM J. Computing*, 1998, to appear.

[2] T. Ikeda, M.-Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku and K. Mitoh. "A Fast Algorithm For Finding Better Routes By AI Search Techniques", *IEEE VNIS'94*, 1994, pp.291–296.

[3] P. E. Hart, N. J. Nillson and B. Rafael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", *IEEE Trans. Sys. Sci. and Cyb. SSC-4,* 1968, pp.100–107.

[4] T. Hiraga, Y. Koseki, Y. Kajitani and A. Takahashi, "An Improved Bidirectional Search Algorithm for the 2 Terminal Shortest Path" (in Japanese), *The 6th Karuizawa Workshop on Circuits and Systems,* 1993, pp.249–254.

[5] M. Luby, and P. Ragde, "A Bidirectional Shortest Path Algorithm With Good Average-Case Behavior", *Proc. 12th International Colloquium on Automata, Languages and Programming,* LNCS 194, 1985, pp.394–403.

[6] G. N. Frederickson. "An Optimal Algorithm for Selection in a Min-Heap", *Information and Computation, 104*, 1993, pp.197–214.

[7] N. Katoh, T. Ibaraki and H. Mine, "An Efficient Algorithm for K Shortest Simple Paths", *Networks*, vol.12, 1982, pp.411–427.

[8] T. Shibuya and H. Imai, "New Flexible Approaches for Multiple Sequence Alignment", *J. Computational Biology*, vol.4, no.3, 1997, pp.385–413.

[9] P. H. Winston, "Artificial Intelligence", Addison-Wesley, 1977.