An Hierarchical Terrain Representation for Approximately Shortest Paths

David Mould and Michael C. Horsch

Department of Computer Science,
University of Saskatchewan,
Saskatoon, SK, Canada S7N 5A9
mould@cs.usask.ca,horsch@cs.usask.ca

Abstract. We propose a fast algorithm for on-line path search in grid-like undirected planar graphs with real edge costs (aka terrains). Our algorithm depends on an off-line analysis of the graph, requiring poly-logarithmic time and space. The off-line preprocessing constructs a hierarchical representation which allows detection of features specific to the terrain. While our algorithm is not guaranteed in general to find an optimal path, we demonstrate empirically that it is very fast, and that the difference from optimal is almost always small.

1 Introduction

Path planning through terrains is a problem often seen in areas including robotics and computer games. By *terrain*, we mean a planar graph whose nodes are evenly distributed across a portion of the plane, and in which each node is connected to its nearby neighbours and only those. In terrains, edges have non-negative weights representing the cost of traversing the edge (not necessarily distance). The cost of a path is the sum of the weights on all edges along the path. We are specifi cally interested in applications that require frequent path planning.

Applications requiring computation of shortest paths in general graphs (not necessarily terrains) are common; algorithms for this task are well-known, and we discuss some of them below. However, general graph search algorithms do not consider the terrain's properties, such as regular connectivity and a tendency to contain regions of similar edge costs, both of which can be exploited to improve search performance.

In this paper, we propose a fast algorithm for on-line path search specifi cally designed for applications in which path planning within a terrain is a very common task. Our technique, called HTAP, uses precomputation and multiresolution. Precomputing a modifi ed representation of the graph can help us speed up path queries.

The HTAP representation is a multiscale one: a "pyramid" of graphs, with the original graph at the bottom and each higher level being a decimated version of the one immediately beneath. The construction of the pyramid extracts features from the terrain that allow important decisions about path planning to be made at a higher level of abstraction. When a pathing query is made, we iteratively perform queries at each level of the pyramid beginning at the top, using the results from higher levels to restrict the region of the graph in which we search at the current level. When we reach the lowest level, corresponding to the original graph, the search space is highly restricted: a narrow corridor of constant width. In Fig. 1 (right) we show an example terrain, and the corridor constructed using HTAP; the number of nodes in the terrain is $243^2 = 59049$, but only 1284 nodes are in the corridor.

The HTAP technique is not guaranteed to find an optimal path. It is well-suited to applications in which real-time path-planning is required, and in which the penalty for slight deviations from optimality is not high. Our empirical results (obtained by a prototype implementation which was not highly optimized) indicate that HTAP can be used in real time for a wide range of terrain sizes. The empirical results suggest linear-time on-line complexity for path planning, although a proof of that claim has not yet been obtained. Empirically, the paths returned by HTAP are rarely worse than 1.3 times the cost of the optimal path, and usually much better than 1.1 times optimal.

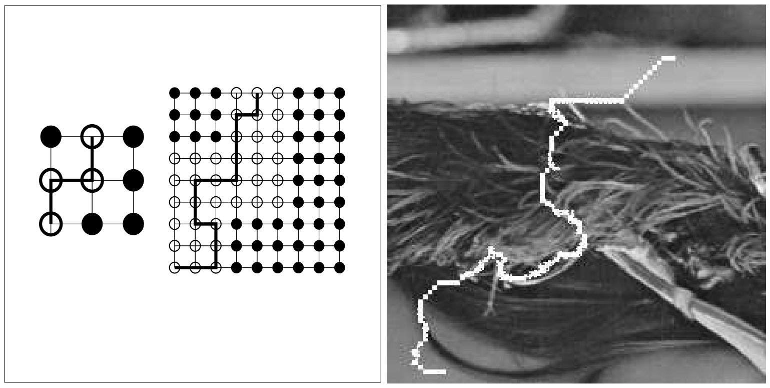


Fig. 1. (left) Two levels of a pyramid: the path, marked in bold, and the corridor derived from the path above, denoted with unfilled circles. (right) A sample terrain with edge costs visualized in greyscale, with the corridor (marked in white) superimposed. Darker regions are cheaper.

2 Previous work

Single source shortest path algorithms such as Dijkstra's algorithm [4] can be too expensive to use repeatedly for on-line or real-time queries. All-pairs algorithms such as Johnson's algorithm [7] have suitable on-line time complexity for repeated path queries, but require quadratic space to store all the paths.

When heuristic information is available, the A* algorithm [5] is often used to solve path planning problems. Its variants include ε-admissible heuristics [11] which attempt to speed up search by relaxing the optimality requirement, iterative deepening A* [8], which improves the memory requirement of A* search, and real-time A* [9]. For terrains, the performance of heuristic search methods can be very slow, because good heuristics are difficult to find. The usual "air-distance" heuristic does not always give a reasonable estimate of the cost-to-goal in terrains with weighted edges, especially when edge costs and distances are measured in different scales. The phenomenon of "flooding," i.e., exploring all vertices in an area near an obstacle to find a way around it, can increase the computational costs dramatically. If a terrain is maze-like, A* guided by a distance heuristic often has to flood large fractions of the terrain to find a path.

Variants of heuristic search, including the use of way-points and multi-level representations, are common in computer game applications [12]. Waypoints are chosen by methods such as "line-of-sight" and "generalized cylinders." Multi-level representations often stop with two levels. The computational costs of heuristic search (using A* or some variant) seem to be accepted as unavoidable.

Techniques for path-finding in robotics include the use of potential fields, Voronoi regions in continuous domains, quad-tree representations of continuous space and wavefront propagation (flood-fill) [10], in addition to techniques mentioned above.

Repeated path planning is central to the area of Intelligent Transportation Systems, and researchers have proposed hierarchical representations to speed up on-line processing [6,3]. The connectivity in ITS graphs can be quite different from terrains, so these methods for building hierarchies cannot be applied directly and usefully to terrains.

Shortest path problems are also important in graph theory and computational geometry. For example, the problem of computing shortest paths in a continuous plane containing a set of polygonal obstacles has received much attention. Chiang and Mitchell [2], for example, give algorithms for this problem that require more than quadratic time and space for precomputation, while allowing sublinear time on-line processing. Arikati *et al.* [1] describe a preprocessing algorithm and hierarchical representation of a planar graph that allows fast on-line shortest path computation. This algorithm requires $O(N^2/r)$ time and space for preprocessing, and the on-line process to find shortest paths requires $O(r)$ time, where N is the number of vertices in the graph, and r is the number preprocessed regions of the graph. The approach above can be extended to approximate shortest paths to within a factor of at most two, requiring $O(N^{3/2})$ time and space preprocessing, and $O(\log N)$ time on-line processing.

3 Algorithm

Our pyramid is a multiresolution representation of the graph: the base of the pyramid is the original graph, and each level above the base is a graph with constant-factor fewer nodes. The nodes at a given level which are also present at the level above, we call *survivors*; the process of selecting survivors we call *decimation*. Each node at the base level has a pointer up to its *representative* at each higher level. We use *immediate representative* to refer to the nearest survivor to an arbitrary-level node.

We have the notion of *blocs*, which are purely geometrically determined entities, and *regions*, which are the Voronoi regions for a given subset of survivors. (The Voronoi region for a survivor is the set of nodes closer to that survivor than to any other survivor, where "closer" is with respect to optimal path length in the original graph. Ties are broken arbitrarily.) In our implementation, a bloc is a 3×3 grouping of nodes, but any constant grouping could have been used instead. Each node at a level above the bottom has a collection of *children* – nodes at the level below which are nearer to it than to any other node of its level. See Fig. 2 (right) for a picture of the survivor-representative relationships, and Fig. 2 for the difference between blocs and regions.

In the remainder of this section, we describe the pyramid representation of the graph; we discuss our algorithm for constructing the pyramid; we describe how the pyramid is used to constrain the search space in resolving paths; and we close with some remarks on complexity.

3.1 Pyramid Representation

The base of the pyramid is the original graph and each upper level is a decimated version of the level beneath. With each original node, we store pointers up to all of its representatives, one per pyramid level. With each upper node, we store two pointers: one down to the node in the lower graph which occupies the same position in the terrain, and one up to the node's representative in the level above. A sketch of the up pointers is in Fig. 2 (right); the marked node demonstrates the necessity of maintaining a list of all representatives at the pyramid base, since the sequence of representatives may differ from the sequence of immediate representatives.

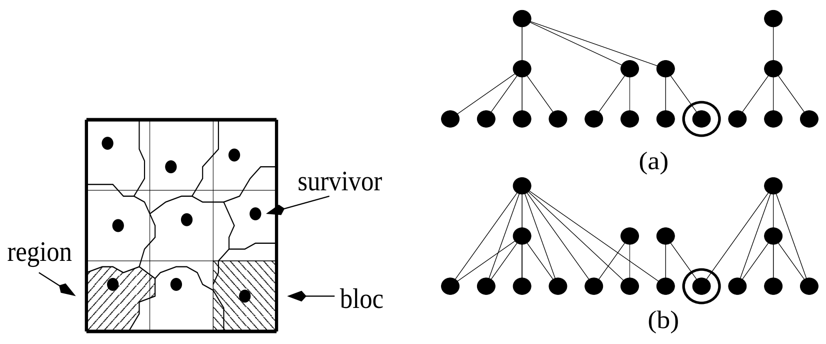


Fig. 2. (left) A sketch of a pyramid level: blocs, regions, and a single survivor per bloc. Two levels of a pyramid. (right) In (a), links between nodes and their immediate representatives; (b), links between bottom nodes and their representatives at all levels. All links are directed upward.

In addition to the information in the pyramid structure, each level of the pyramid is a proper graph, which is to say that it contains weighted edges between nodes. Details on how the graph connectivity and edge weights are computed appear below.

3.2 Pyramid Construction

We repeatedly perform the following process, stopping when the newly created level is suffi ciently small. "Suffi ciently small" is with respect to the maximum graph size we are willing to risk searching exhaustively; when we come to do path queries, we will fi rst fi nd a path at the top level of the pyramid.

Suppose we are constructing a new level i, $i > 0$, where $i = 0$ is the bottom of the pyramid. We perform the following steps, explained in greater detail below:

1. Divide the level into blocs.
2. Choose from each bloc a single node to survive to the level above (decimation).
3. Find the Voronoi regions for the survivors, in the original graph.
4. Determine which pairs of survivors should be linked by edges.
5. Determine costs for the new edges from the previous step.

Division into blocs For each node in the current level, we assign a bloc identifier based on its location in the terrain. In our implementation, 3×3 groups of nodes were assigned to a single bloc.

Decimation From each bloc there will be a single survivor. The challenge is to decimate the graph so as to best preserve its structure, from the point of view of finding short paths. The nodes most worth preserving are those which lie on the greatest number of optimal paths among the entire ensemble of paths; unfortunately, computing all paths is an extremely expensive task. We choose instead to compute a proxy, inspired by the notion of parallel resistance in electrical circuits. The resistance of a node R is given by

$$1/R = 1/c_1 + 1/c_2 + \ldots + 1/c_n \quad (1)$$

where c_j is the cost of the jth edge to the node. Within a given bloc, the node with the lowest resistance is the one used by the most paths, and hence the one which survives to the level above. We use resistance because it is a natural measure of the difficulty of traversing a node. Parallel resistance behaves gracefully when new edges are added, decreasing monotonically as more paths become available. Having some low-cost edges will give a node low resistance, but even high-cost edges might be used by some paths.

Voronoi regions We next find the Voronoi regions for all survivors, making use of breadth-first search from multiple initial points. Our distance metric is path cost within the original graph. All bottom-level nodes fall within the Voronoi region of some survivor; for each bottom-level node, we record which survivor is closest. Also, if the new level $i > 1$, then for every node at level $i - 1$ we record which survivor at level i is closest (the immediate representatives) using the already-determined Voronoi regions.

Placing edges Initially level i has no edges. We place a new edge between every pair of nodes at level i whose Voronoi regions at the pyramid base are linked by at least one edge.

Finding new edge costs The cost of an edge between two nodes is the path cost of travelling between these two nodes in a subset of the original graph, where the path is restricted to lie within the Voronoi regions of the two nodes in question.

3.3 Query Processing

Each shortest-path query consists of a start and end node. The overall pathfinding exercise is a cascade of searches; at each level below the top, we find a path by searching in a tightly restricted subset of the original graph.

We begin by finding the representatives of both nodes at the top level and finding the optimal path through the entire top-level graph using A*. Having found a path at a given level, we then mark all children of the nodes on the path as eligible, and find the shortest path one level down, searching only in the eligible corridor. The algorithm ends when a path is found at the pyramid base. Fig. 1 suggests how the corridor is derived from the path one level up and used to constrain the search space.

A subtask of the path query resolution process involves marking the children of a given node, so that we can add the marked nodes to the corridor. However, the nodes do not explicitly store their children. To mark a node's children, we perform the following: we first find a single child, then we flood to find all nodes at the child's level who share its representative. The nodes form a contiguous region — recall that they are the Voronoi region for their representative — and therefore flood-fill can efficiently mark the region. The initial child is found by going to the node's location in the original graph (where pointers to all levels are stored) then taking the pointer up to the proper level. Fig. 2 shows sketches of the pyramid structure.

3.4 Complexity

Construction of the pyramid requires $O(N \log N)$ time, where N is the number of nodes in the original graph. There are $O(\log N)$ levels in the pyramid, and constructing each requires an investigation of every node at the pyramid base. The memory footprint of the pyramid is $O(N \log N)$ because at the base level, every node has a pointer up to every level above, and there are $O(\log N)$ levels. There are $O(N)$ nodes in total in the pyramid, from equation 2 below.

If our survival policy does a good job of making regions of approximately equal size, then the run-time complexity of the algorithm is $O(n)$, shown as follows. At level i, we are seeking a path of length $p^i n$, where n is the length of the bottom-level path, and the linear dimension of the graph was reduced by a factor $p < 1$ at each level of the pyramid. Then we have a total computational cost of

$$n + pn + p^2 n + p^3 n + ... + p^k n \leq n(\sum_{i=0}^{\infty} p^i) = n/(1-p). \tag{2}$$

In general, our algorithm is not guaranteed to find the shortest path. Our empirical results are presented below. Here, following Pearl [11], we consider an abbreviated analysis of the algorithm on a regular 4-connected lattice with each edge having unit cost. In this kind of grid, an optimal path has the property that each edge traversed on the path reduces the Manhattan distance to the goal. By construction (assuming ties are broken in a deterministic manner), each level in the pyramid is a regular lattice with uniform edge costs. An optimal path at level $k+1$ in the pyramid defines a corridor in the kth level of the pyramid which contains an optimal path at level k.

4 Results

Next we report results from our experiments. Each experiment consisted of a single path planning exercise. Endpoints were chosen at random within the map; the optimal path was determined using $A^*$and compared to the path reported by HTAP. We compare the computational costs of finding paths, in terms of the opened node count, and the path costs themselves, expressed as the ratio of the optimal path to the HTAP path.

In reporting the computational costs of using the pyramid to answer path queries, we sum all nodes in all corridors in the pyramid. Even though we might not open a given node when searching the corridor, we had to visit the node to mark it eligible.

We tested HTAP on a number of different maps. Specific maps employed were the *noise map*, where every edge had a cost chosen at random with uniform distribution over $\{1..255\}$; two *maze maps*, hand-drawn mazes whose edge costs were either 1 (hallway) or 255 (wall); and various image maps, where standard test images were converted to greyscale and edge costs were derived from pixel intensities. Image maps produced edge costs as follows: for two nodes (pixels) p_1 and p_2, having intensities i_1 and i_2 respectively, the cost was $\max(1,(i_1+i_2)/2)$. Pictures of our maps are shown in Fig. 5.

We chose to use images because they share some characteristics with real terrains. They have a wide range of edge costs, but pixel intensities (and the derived edge costs) tend to be correlated, and in some cases it is possible to divide the terrain into subregions within which costs are roughly uniform. The presence of such subregions is a feature of real terrains. The images we used are standard test images in the computer vision community, in the absence of a standard set of terrains.

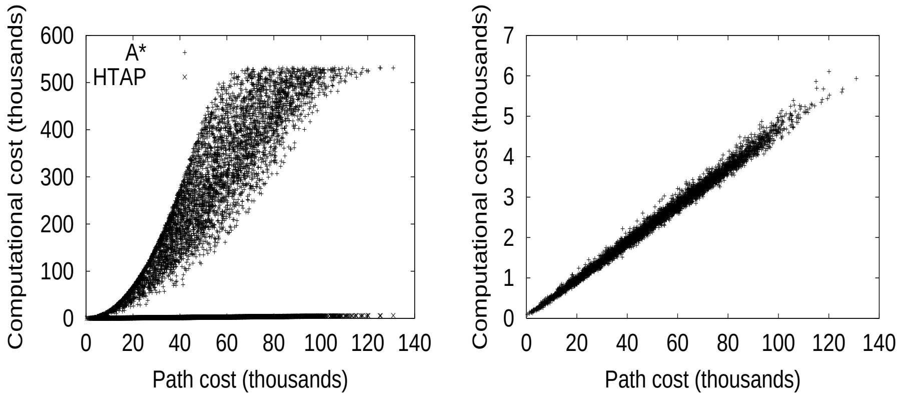
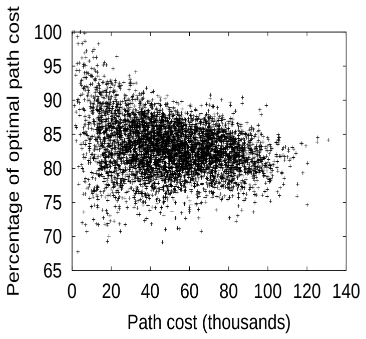


Fig. 3. (left) Comparison of computational costs for HTAP and for A*. The HTAP data points lie along the x-axis and are difficult to see. (right) The HTAP computational costs alone. Note the change in range on the *y*-axis between the two figures.

The multiresolution representation allows us to find even long paths very quickly; see Fig. 3 (left) for a time comparison between A* and HTAP. Strikingly, the data points for HTAP are scarcely visible. On the scale of the graph, our computational cost is near zero. The difference illustrated by the graph is the difference between O(n^2) and O(n). Our path costs are also shown by themselves, where the O(n) behaviour is more apparent. There is some variation owing to the slight differences among different region sizes. Each of these graphs shows 5000 random paths on the 729×729 noise map. Also of interest is the comparison between our paths' costs and the optimal paths' costs, shown in Fig. 4; we show an example to give the flavor, with detailed data in the table. From the graph we see that the cost of short paths is very likely to be near the optimal path cost, and that while the ratio drops off somewhat as paths become very long, it never drops off too far. Again, the results in the graph are for 5000 random paths on the 729×729 noise map. The results for the noise map are representative of results on other maps.

In the table, we report cost differences in terms of the cumulative distribution: what proportion of paths were within 1% of optimal, what were within 5%, and so on. Percentage values represent the ratio of the optimal path to the reported path (so 50% is the spot where the optimal path had half the cost of the reported path). We had virtually no cases where the reported path was worse than 50%. Our results are better for more structured images, which would be more susceptible to compression; even for the largest noise map, however, 95% of our paths were within 25% of optimal. Each table row summarizes the results from 5000 paths; in practice, we found that the results were stable to one decimal point after a few thousand trials.



Map	size	1%	5%	10%	25%
noise	243×243	0.8	3.4	16.2	95.2
noise	729×729	0.1	0.6	3.8	98.4
simple maze	243×243	57.1	92.1	96.3	99.2
complex maze	340×340	57.1	68.6	77.5	92.6
Lena	512×512	25.3	78.4	89.4	97.3
mandrill	512×512	15.3	67.0	84.7	96.9
peppers	512×512	17.4	71.4	87.1	97.6
terrain	512×512	1.3	30.3	72.4	98.5

Fig. 4. (left) Scatter plot of path costs. The horizontal axis is raw path cost and the vertical axis is the ratio of the costs of the optimal path and the reported path. (right) Cumulative distribution of path costs.

We compared HTAP and A^* since A^* is the standard heuristic search algorithm. The problem with applying A^* or its variants to path fi nding in terrains is the lack of a good heuristic. A heuristic based on distance will rarely give good guidance in a terrain whose costs are not commensurate with distance. For example, we performed some preliminary experiments with A^*_ε[11], and found that A^*_ε performs extremely poorly in terrains. For small ε, A^*_ε performed only marginally better than A^* in terms of computational effort. For larger ε, A^*_ε wasted enormous effort revisiting nodes for which the fi rst paths found were suboptimal.

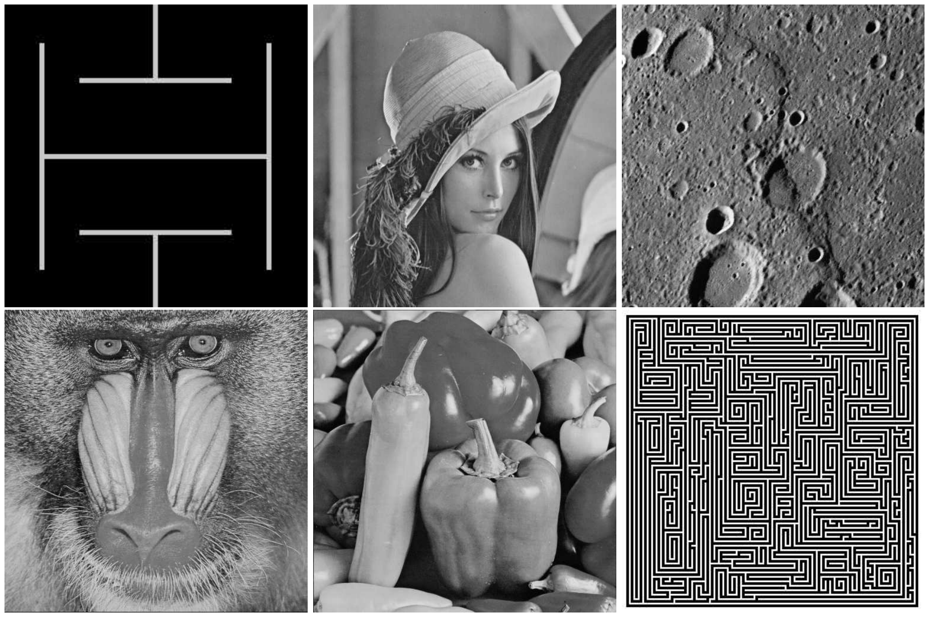


Fig. 5. Visualizations of some of the graphs we used. Above, the simple maze, the Lena image, and a terrain image; below, the mandrill image, the peppers image, and a complex maze.

5 Discussion

Empirical results for HTAP suggest that it is O(n) in the path length, rather than the typical O(n^2) for A^* in terrains. It is extremely fast, two orders of magnitude faster than A^* on the maps that we tried. It can be applied to very large maps; we have successfully used it on maps of size 729×729. It has the disadvantage that it does not always return the optimal path, rather a path whose cost is not much worse than optimal.

Although fast for long paths, our method incurs some overhead and hence is not very fast for short paths. For extremely short paths, the corridor is more egg-shaped than ribbon-shaped, and contains many more nodes than are needed. However, for short paths it is also highly likely that the corridor contains the optimal path. The longer the corridor, the more likely it is that some part of the optimal path lies outside it. We have often observed the behaviour that the optimal path lies largely within the corridor, only departing for a short detour before returning.

Our algorithm is best able to find the optimal path when the original graph is well represented by the pyramid: hence, large regions with near-uniform edge costs lose little information when most of the edges are removed. Highly non-uniform regions suffer more, which is why our results on the noise map come so much further short of optimal. The maze maps were difficult because even small lossiness in compression can translate into serious errors in the paths. Note, however, that even in the difficult cases the corridor usually contains a very good path.

6 Conclusions and Future Work

We have presented a fast method for finding short paths in the graph. Though the method is not guaranteed to find the shortest path, with high probability it finds a path not much more expensive than the optimal path – and moreover, it finds a path of length n by searching only $O(n)$ nodes. Our method depends on precomputing a multiresolution structure of size $O(N \log N)$ in the number of nodes in the graph; the precomputation is expensive, but runtime queries are processed very quickly. On a 1.8 GHz AMD processor, it requires about 7.5 minutes to compute the pyramid for a 729×729 graph.

We have presented algorithms for building the pyramid and for exploiting it for fast path planning. Future work involves optimizing the pyramid construction and investigating the tradeoffs between performing more computation at runtime and having a higher likelihood of finding the optimal path.

We are interested in investigating the tradeoffs between a wider corridor and a better chance of finding the optimal path. We have considered only static terrains so far, while some application areas involve dynamic terrains; we are therefore interested in looking at incremental modification to the pyramid. Our memory footprint right now is $O(N \log N)$ in the size of the original graph, and we believe that we can reduce it to $O(N)$. We are interested in looking at the effects of different policies for node preservation. We want to investigate a hybrid algorithm, in which a traditional method is first used, and HTAP is used only when the traditional method does not quickly find the solution. Finally, we want to perform more detailed analysis of HTAP's complexity.

References

1. Srinivasa Rao Arikati, Danny Z. Chen, L. Paul Chew, Gautam Das, Michiel H. M. Smid, and Christos D. Zaroliagis. Planar spanners and approximate shortest path queries among obstacles in the plane. In *European Symposium on Algorithms*, pages 514–528, 1996.
2. Yi-Jen Chiang and Joseph S. B. Mitchell. Two-point euclidean shortest path queries in the plane. In *Symposium on Discrete Algorithms*, pages 215–224, 1999.
3. Y. Chou, H.E. Romeijn, and R.L. Smith. Approximating shortest paths in large-scale networks with an application to intelligent transportation systems. *INFORMS Journal on Computing*, 10:163–179, 1998.
4. E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
5. P.E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determiniation of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.
6. Yun-Wu Huang, Ning Jing, and Elke A. Rundensteiner. Hierarchical path views: A model based on fragmentation and transportation road types. In *ACM-GIS*, pages 93–100, 1995.
7. D.B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. Assoc. Comput. Mach.*, 24(1):1–13, 1977.
8. R.E. Korf. Iterative-deepening A^*: An optimal admissible tree search. In *IJCAI-85*, pages 1034–1036, 1985.
9. R.E. Korf. Real-time heuristic search. *Airtificial Intelligence*, 42(3):189–211, 1990.
10. Robin R. Murphy. *Introduction to A.I. Robotics*. MIT Press, 2000.
11. Judea Pearl. *Heuristics: Intelligent Search Strategies for Intelligent Problem Solving*. Addison-Wesley, 1984.
12. Steve Rabin, editor. *AI Game Programming Gems*. Charles River Media, Inc, 2002.