Learning to Navigate on a Graph*

Azriel Rosenfeld,[1] Ehud Rivlin,[1,2] and Samir Khuller[3]
[1]Center for Automation Research, University of Maryland
College Park, MD 20742-3275
[2]Department of Computer Science, Technion
Haifa 32000, Israel
[3]Computer Science Department and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742

Abstract

If a robot has to repeatedly perform navigational tasks in a complex environment, it can improve its performance by "learning" the layout of the environment. This paper is the first of a planned series on navigational applications of learning. It deals with a "street network" environment, modeled by a graph embedded in the plane; and with a "taxi-driver" agent which initially finds a path to its destination, and afterwards tries to find shorter paths to it. For a class of graphs that correspond to planar mazes, we compare several basic search strategies for finding the destination; we show that once it has been reached, a much shorter return path can usually be found; and we also define a simple path-shortening strategy that searches for shortcuts in given paths. Finally, we discuss types of partial information about the environment that might be especially useful to store if the agent has limited memory.

1 Introduction

Imagine that you suddenly find yourself driving a radio-dispatched taxicab in a totally unfamiliar city. The city has no street signs or other distinctive landmarks, and you do not have a map, but fortunately your taxi is equipped with a radio link to the Global Positioning System, so you always know the coordinates of your position.[1] Furthermore, the dispatcher always identifies passenger pickup spots, and the passengers identify their destinations, by specifying their coordinates. Unfortunately, the street pattern in the city is very complex—not at all like a regular grid; thus even though you know your current coordinates and the coordinates of your destination, it isn't at all obvious how to get to the destination. The first time you are given a destination, you can try to drive toward it, but you may soon discover that the city streets run at odd angles, are curved, and have many dead ends, so you are often forced to backtrack.

Luckily, you have an excellent memory, and can recognize (by their coordinates) places where you have already been; this allows you to avoid repeating the same mistakes while searching for destinations. In fact, if you wish, you can memorize a complete description of the parts of the street network on which you have driven. Evidently, you may then be able to use the memorized information to shorten later searches that involve previously visited parts of the network. If you are sent to many different destinations, you will eventually have visited a large fraction of the network; this will often allow you to "plan" (and then take) a short path to a destination without having to search for it at all.

If you have perfect, unlimited memory (where retrieval of information from memory, and computations performed on this information, take negligible time and effort), and if your job will require you to go to many different destinations, it evidently makes sense for you to systematically build up, as you drive, a "complete" representation of the part of the network that you have visited so far. [For example, you could construct a matrix whose rows and columns correspond to street intersections and dead ends ("nodes"), identified by their coordinates, and where if two nodes are joined by a street, the matrix specifies the distance between them along the street; this represents the street network as a graph with labeled nodes and weighted edges.] On the other hand, if you will usually need to go only to a few des-

*The support of the Advanced Research Projects Agency (ARPA Order No. 8855) and the Air Force Office of Scientific Research under Contract F49620-92-J-0549 to George Mason University is gratefully acknowledged. The second author also thanks Prof. S. Markowitz of the Technion for helpful comments.

[1]After writing this introduction, we discovered that a very similar scenario is described in [Blum and Chalasani, 1993].

tinations (hotels, stations, ...), or if your memory capacity and memory search time are limited, it may make more sense to memorize the most frequently needed paths (e.g., stored as sequences of turns and street lengths), and to use search (as you did when you were a new driver) only when you have to find unusual destinations. Note, however, that even if you have only a limited set of destinations (e.g., your taxi is a "shuttle" which only needs to go back and forth between two points), you still need to search for destinations initially, and even after you have found them, you can still use search to look for shorter paths.

Our taxi driver scenario is a paradigm for a general class of problems dealing with a mobile agent that operates in a complex environment, and needs to (repeatedly) go from its current position to a specified destination; this is evidently a basic navigational task. We gave our agent an absolute position sense, so it would not have to use other types of sensory data (e.g., vision) to identify goals, landmarks, etc. (Obviously, we do assume that the agent uses sensory data to follow the streets, detect intersections, make turns, avoid collisions, etc.; but these aspects of navigation are not of interest to us at the moment.) Our basic problem is: How can the agent "learn" the structure of the environment, in order to simplify the task of finding future destinations? In this paper, the agent is a "taxicab" which always knows its position, and the environment is a network of "city streets" (a planar graph, which we will further restrict in our experiments to a class of mazes with walls piecewise parallel to the coordinate axes). Evidently, however, we could have allowed other types of networks (a road network that has over- and underpasses, or a network of caves or sewer pipes, which can correspond to an arbitrary graph, or a network in which streets can be one-way, corresponding to a directed graph); or we could have posed essentially the same problem in other types of environments (e.g., a region of two- or three-dimensional space containing polygonal or polyhedral obstacles) or for other types of agents (e.g., agents that can climb or jump, or that can use sensors of various types as aids to navigation).

In this paper we consider the following questions about an agent that has to find destinations in a network (specifically, a maze):

a) How can the agent find its first destination?

b) If the agent can remember every part of the network that it has already visited, how can it use this knowledge to find new destinations, or shorter paths to old destinations?

c) If the agent has limited memory, what information about the network should it memorize?

Graphs (and mazes) have been used by many authors as a framework for theoretical studies of navigation. Algorithms for systematic traversal of a graph (which can be regarded as another basic navigational task: "patrolling") are well known (e.g., [Even, 1979]; see [Deng and Papadimitriou, 1990; Betke, 1992] for recent work on economical traversal of all the edges in a directed graph). Traversal of mazes is a classical problem [Lucas, 1882] which continues to be studied (e.g., [Blum and Kozen, 1978; Fraenkel, 1970]). Work on destination finding has dealt primarily with mazes (or planar regions that contain obstacles of simple shapes); for a review of this class of search problems see [Huang and Ahuja, 1992], and for other recent examples see [Papadimitriou and Yannakakis, 1991; Blum et al., 1991; Zelinsky, 1992; Balch and Arkin, 1993; Lumelsky and Tiwari, 1994]. Very little work seems to have been done, however, on learning to improve performance in destination finding; an exception is a recent paper by Blum and Chalasani [1993], which deals only with axis-parallel rectangular obstacles in the plane.

In Section 2 of this paper we describe search algorithms that the agent can use to find destinations, and we illustrate the performance of these algorithms on a class of planar mazes. In Section 3 we describe methods that can be used to find short paths, or shortcuts in a given path. Section 4 discusses planned extensions of our work; in particular, it suggests types of partial information about the environment that could be useful to an agent that has limited memory.

2 Robotic graph (and maze) search

Since the agent initially knows nothing about the layout of the graph, to find its destination for the first time it must use a search algorithm that is capable of visiting every node of the graph. Evidently, standard graph traversal algorithms [Even, 1979] can be used for this purpose. However, such algorithms take no advantage of the fact that our street-network graph is embedded in the plane, and that the agent always knows its own coordinates and the coordinates of its destination ("goal"). This knowledge allows the agent to use search algorithms that attempt to reduce its (apparent) distance from the goal, defined in terms of coordinate differences. In our experiments, the graph represents a maze whose walls are everywhere parallel to the coordinate axes, so that the "city block" (or "Manhattan", or L_1) distance $|x-a|+|y-b|$ is a lower bound on the distance in the maze between the position (x, y) of the agent and the position (a, b) of the goal.

"Heuristic" graph search algorithms that try to find a goal node in a graph by attempting to reduce an estimate of the distance to the goal have been extensively studied in the Artificial Intelligence literature [Pearl, 1984]. However, these algorithms allow the search to "jump" from one part of the graph to another, e.g. to shift the search to whichever (known) node has minimum estimated distance to the goal, even if this node is not adjacent to the current search position. A "robotic" agent, on the other hand, operating on a "real" graph, can only make moves between neighboring nodes. In this section we consider only graph search algorithms that take this limitation into account.

As a "baseline", we use a standard method for searching graphs called Depth First Search [Even, 1979], referred to below as DFS. Starting from an initial node s of the graph, DFS searches for a desired goal node t in accordance with the following pseudocode:

```
proc DFS($v$)
    mark $v$ visited
    If $v \neq t$ then
        for each $w$ adjacent to $v$ do
            if $w$ is not marked visited
            then DFS($w$)
        end for
        return
    else break
end proc
```

The initial call is DFS(s). Note that we must examine the nodes adjacent to v in some order; we choose as the "next" node w the first unmarked neighbor of v that we find.

DFS does not specify the order in which we should examine the neighbors of v after making the call DFS(v). Our next algorithm, called D-DFS (for "Directional DFS"), uses coordinate information in choosing this order. Specifically, it gives priority to the neighbors of v whose L_1 distances from t are smallest. Note that in a Cartesian grid, a node v can have two neighbors that are closer than v (in L_1 distance) to t; one of these neighbors has a smaller x-coordinate difference $|x-a|$, and the other has a smaller y-coordinate difference $|y-b|$. D-DFS chooses the neighbor which reduces the larger of the two differences; ties are resolved arbitrarily.

An alternative method of assigning an ordering to the neighbors of v takes advantage of the fact that we know the directions to the neighbors. This allows us to use the classical maze-solving strategy in which, as you explore the maze, you keep your right hand on the wall. [This strategy works well when we know that the start and goal nodes are both adjacent to a common wall (such as the boundary wall of the maze). If they are not, we can build an "imaginary" boundary wall that has the start and goal nodes (say) in opposite corners, and explore the maze in such a way that we never cross this imaginary wall. This strategy will work, provided the start and goal nodes belong to the same connected component of the maze even after the wall is built. It has the advantage that we do not need to memorize the coordinates of the nodes that we have already visited; it is a memoryless strategy. It also allows us to avoid exploring certain types of "dead ends", which can be identified when the path approaches the wall, as illustrated in Figure 1.] We can use a similar strategy to explore a graph whose nodes are embedded in the plane, by always examining the neighbors of v in counterclockwise order, starting from the edge along which we reached v. We shall call this algorithm RHW.

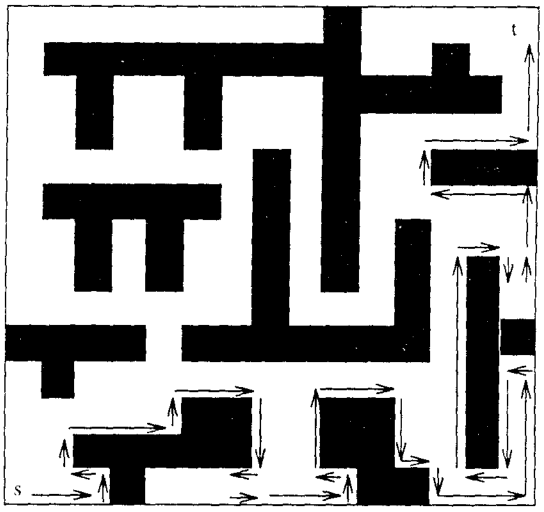
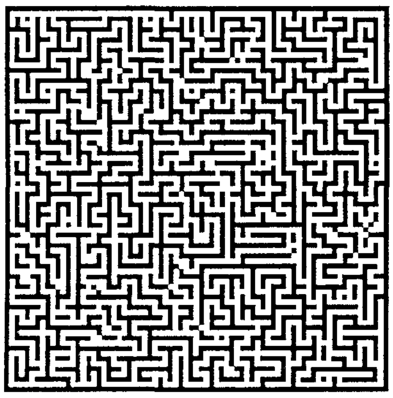


Figure 1: Shortcutting an RHW path at a dead end.

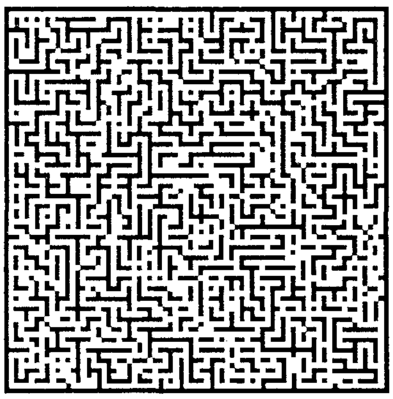
In D-DFS and RHW, the only possible choices for the "next" node are the neighbors of the "current" node. We next describe a generalization of D-DFS which we call RA* (for "Robotic A*"); we have not yet implemented this generalization because an efficient implementation would require an efficient algorithm for updating all the shortest (known) distances between pairs of nodes each time a new node of the graph G is visited. Specifically, let the current node be v, and let G' be the graph consisting of all the nodes and edges of G that have been visited so far. For every node u of G' we compute the shortest distance $g(v,u)$ in G' between u and v, and we also compute the L_1 distance $h(u,t)$ between u and the goal node. RA* then chooses as next node a node u which minimizes $g(v,u)+h(u,t)$, or perhaps a linear combination of them. [Note that since we are dealing with a "real" robot on a "real" graph, we cannot simply "jump to" the node u that minimizes $h(u,t)$; we must take into account the cost $g(v,u)$ of getting to u from v.]

A maze can be represented by a binary digital image in which the walls are composed of black pixels (unit squares). To represent a maze by a graph, we take the white pixels to be the nodes of the graph. Two nodes have an edge between them if the corresponding pixels share a common side. Note that in this graph, each node has degree at most 4; the possible directions to the neighbors of a node are east, west, north, and south, corresponding to differences of ± 1 in the x- and y-coordinates of the pixels.

To construct mazes for our experiments, we first created "dense" mazes using the public-domain PC program MAZE. Such mazes typically allow only one path from the start to the goal. We then made these mazes less dense by randomly changing pixels from black to white. Examples of such mazes of size 75×75 are shown in Figure 2. In Figures 2a and 2b, 5% and 15% of the original black pixels were changed to white, respectively.



(a)



(b)

Figure 2: Examples of mazes having two different "densities".

We tested DFS, D-DFS, and RHW on a set of mazes of size 200×200. We first created five dense mazes, and then randomly changed 5%, 10%, and 15% of the black pixels to white; this gave us five examples each of mazes that had three different "densities". The start and goal nodes were always the cells in the southwest and northeast corners of the maze. Figure 3 shows examples (displayed as dark gray) of the sets of nodes visited by the three algorithms while searching for the goal in mazes of size 75×75, for two of the maze densities. (The lighter-gray paths in Figure 3 will be explained in Section 3.) The efficiency of a search algorithm is measured by the number of search steps that were needed to find the goal; averages of these numbers for the five mazes of size 200×200 for each search algorithm and each maze density, are shown in Table 1. As the table shows, RHW performs much better for dense mazes, but D-DFS catches up as the maze becomes less dense.

Table 1: Average number of search steps to reach the goal in mazes having three different densities, using three search algorithms.

	Percentage of black pixels changed to white		
Algorithm	5%	10%	15%
DFS	13036	7142	6554
D-DFS	10077	1800	813
RHW	1733	1221	856

moves

3 Path shortening

As soon as the agent reaches the goal node, it can use the information about the graph that it acquired while searching for the goal to find a much shorter path back to the start node. Let G' be the part of the graph that the agent has visited; then the agent can—"in its head", without moving—apply a breadth-first search (BFS) algorithm to find a shortest path in G' between the start and goal nodes. This "return" path, which is shown in light gray in Figure 3, will usually be much shorter than the path taken by the agent in searching for the goal. Table 2 gives the average lengths of these paths for the sets of examples treated in Table 1, as well as the average lengths of the true shortest paths from start to goal in these mazes, obtained by applying BFS to G. It should be pointed out that the bigger G' is (i.e., the more exploration was needed to find the goal), the shorter will be the return path in G'.

We now address the issue of still further shortening the path when we next actually traverse it, by exploring in its vicinity for possible

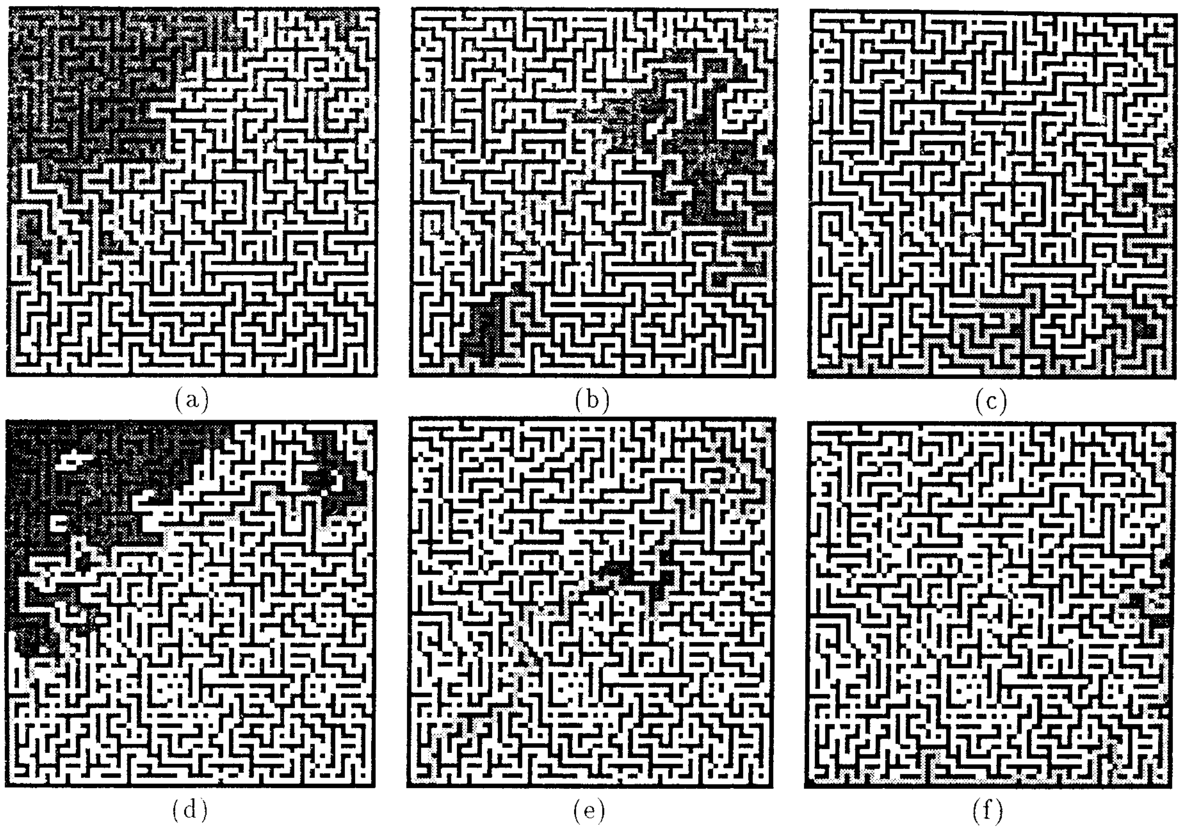


Figure 3: Nodes visited while searching for the goal in mazes of size 75 × 75 having two densities, using three search algorithms. (a,d) DFS; (b,e) D-DFS; (c,f) RHW. The upper (lower) maze in each column was generated by randomly changing 5% (15%) of the black pixels to white.

Table 2: Average "return" path lengths for the six cases in Table 1, and average true shortest path lengths for the three sets of mazes.

	Percentage of black pixels changed to white		
Algorithm	5%	10%	15%
DFS	908	775	628
D-DFS	1005	761	642
RHW	1232	975	746
BFS (in G) (True shortest path)	642	470	418

shortcuts. Natural places at which to look for shortcuts are segments $(v_i, \ldots, v_j)$ of the path $(v_1, \ldots, v_n)$ for which the arc length $j-i$ is high and the L_1 "chord length" $h(v_i, v_j)$ is low, since such segments can potentially be significantly shortened (provided the endpoints of the segment are not on opposite sides of a long piece of wall).

We applied this method of path shortening to the return path found after RHW was used to find the goal in one of the "15%" mazes of size 200 × 200. RHW generated relatively small G's in these sparse mazes, so that the return paths were relatively long (see Table 2), and could thus especially benefit from further shortening. We considered all path segments $(v_i, \ldots, v_j)$ such that $h(v_i, v_j) \leq 5$ and $j-i \geq 15$ and such that the pixel adjacent to v_i in the direction toward v_j, and vice versa, was not blocked by a wall.

Figure 4 shows "closeups" of two such segments of the return path that were judged to be the best candidates for shortening; they have arc lengths 30 and 20, respectively. As indicated in Figure 4, both of these segments can indeed be shortened considerably—in fact, they can be replaced by arcs of lengths 4 and 8, respectively. [It is not hard to see that these shortcuts are the same as the paths from v_i to v_j that would have been found by D-DFS; other search algorithms could also be used to find the shortcuts.]

It should be pointed out that if the agent expects to make very few trips, finding shorter paths may be of little benefit to it, since the effort spent in searching for a shorter path may be greater than the effort required to take a longer

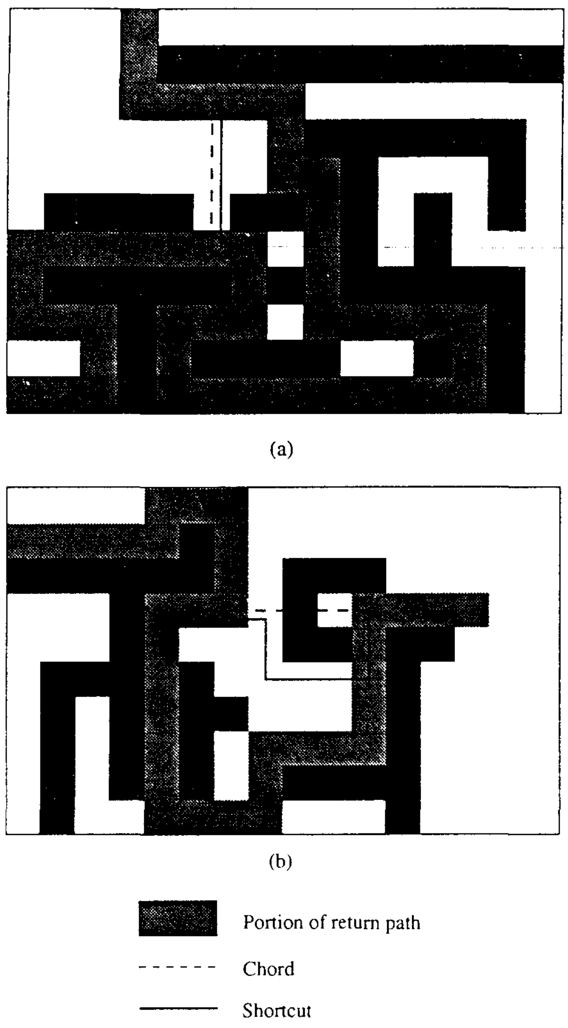


Figure 4: Shortcuts found in two segments of a return path.

path. On the other hand, if the agent expects to make very many trips, even a small amount of path shortening will pay off; in fact, it would be worth the agent's while to systematically traverse the entire graph and determine shortest paths to its destination(s), since these paths will be used many times, whereas the "overhead" of graph traversal and shortest path determination is incurred only once. Thus searching for shorter paths is of interest primarily when the expected number of trips is not very large, and this in turn limits the amount of effort that should be spent in the search.

4 Future directions

Evidently, the ideas discussed in this paper can be extended in many directions. As pointed out in Section 1, the problems considered here can also be posed in many other types of domains, involving various classes of agents that have to perform various types of navigational tasks in various types of environments. Even for the specific choices of agent, task, and environment considered here, other types of search algorithms (e.g., RA*) could be investigated. It would be of interest to characterize the expected performance of all these algorithms on various classes of mazes or graphs.

Path shortening heuristics should be systematically investigated; here too, it would be of interest to characterize their expected performance on various classes of graphs. It would also be desirable to extend the theoretical results of [Betke et al., 1993] and [Blum and Chalasani, 1993] to our more general class of environments.

An especially interesting direction for future investigation is the study of how agents that have limited memory capacity can learn to navigate. Memory limitations may account for the navigational strategies used by various kinds of animals, particularly insects (e.g., [Wehner and Wehner, 1990]); and they may also be helpful in explaining human performance on complex navigational tasks in strange environments (e.g., [Elliott and Lesk, 1982]). We have not yet attempted to formulate models for animal or human navigational performance based on such considerations, but in the following paragraphs we suggest some tools that a limited-memory agent might use in finding destinations.

If the environment is large, and the agent has a limited amount of memory, it may not have the memory capacity to store information about the entire environment, or even about the portion it has traversed. In such a situation, a possible strategy for the agent is to store a "spanner" of an appropriate size. A *c-spanner* of a graph G is a sparse spanning subgraph H with the property that the distance between any pair of nodes in H is no more than c times the distance between them in G. Thus H is a "sparse" representation of G that encodes "approximate" shortest paths between every pair of nodes of G. For recent results on graph spanners see [Althöfer et al., 1993; Chandra et al., 1992]. The main result is that if $c = 1 + t$, any graph G has a spanner with at most $O(n^{1+\frac{2}{t}})$ edges.

Another possible strategy is to partition G into a union of "neighborhoods" $N_1 \cup \ldots \cup N_k$ such that each neighborhood has limited size and limited radius (with respect to some central vertex v_i); the size and radius might be functions of the size of G. Such a neighborhood decomposition of G can be used as an approximate "map" of G. We memorize all the shortest paths (or build a spanner) between the centers of the neighborhoods. We can then find a path from s to t by first finding a path from s to the center of its neighborhood; then taking a shortest path to the center of t's neighborhood; and finally finding a path from that center to t.

The "learning" studied in this paper consists primarily of discovering short(er) paths to destinations using search techniques. A more interesting objective would be to discover general characteristics of the environment that could be useful in guiding the search for short paths. An agent that could accomplish this could more correctly be described as "learning to navigate".

References

[Althöfer et al., 1993] I. Althöfer, G. Das, D. Dobkin, D. Joseph, and J. Soares, On sparse spanners of weighted graphs, *Discrete and Computational Geometry* **9**, 1993, 81–100.

[Balch and Arkin, 1993] T. Balch and R. Arkin, Avoiding the past: A simple but effective strategy for reactive navigation, Proc. IEEE ICRA, 1993, 678–685.

[Betke, 1992] M. Betke, Algorithms for exploring an unknown graph, Master's thesis, MIT Department of Electrical Engineering and Computer Science, February 1992 (also MIT/LCS/TR-536, March 1992).

[Betke et al., 1993] M. Betke, R. Rivest, and M. Singh, Piecemeal learning of an unknown environment, Proc. ACM COLT, 1993, 277–286.

[Blum and Chalasani, 1993] A. Blum and P. Chalasani, An on-line algorithm for improving performance in navigation, Proc. IEEE FOCS, 1993, 2–11.

[Blum and Kozen, 1978] M. Blum and D. Kozen, On the power of the compass (or, why mazes are easier to search than graphs), Proc. IEEE FOCS, 1978, 132–142.

[Blum et al., 1991] A. Blum, P. Raghavan, and B. Schieber, Navigating in unfamiliar geometric terrain (preliminary version), Proc. ACM STOC, 1991, 494–504.

[Chandra et al., 1992] B. Chandra, G. Das, G. Narasimhan and J. Soares, New sparseness results on graph spanners, Proc. ACM Symp. on Computational Geometry, 1992, 192–201.

[Deng and Papadimitriou, 1990] X. Deng and C.H. Papadimitriou, Exploring an unknown graph (extended abstract), Proc. IEEE FOCS, 1990, 355–361.

[Elliott and Lesk, 1982] R.J. Elliott and M.E. Lesk, Route finding in street maps by computers and people, Proc. AAAI NCAI, 1982, 258–261.

[Even, 1979] S. Even, *Graph Algorithms*, Computer Science Press, Potomac, MD, 1979, Ch. 3.

[Fraenkel, 1970] A.S. Fraenkel, Economic traversal of labyrinths, *Math. Magazine* **43**, 1970, 125–130 (and correction, ibid. **44**(1)).

[Huang and Ahuja, 1992] Y.K. Huang and N. Ahuja, Cross-motion planning—a survey, *ACM Computing Surveys* **24**, 1992, 219–291.

[Lucas, 1882] E. Lucas, Récreations Mathématiques, Paris, 1882.

[Lumelsky and Tiwari, 1994] V. Lumelsky and S. Tiwari, An algorithm for maze searching with azimuth input, Proc. IEEE ICRA, 1994, 111–116.

[Papadimitriou and Yannakakis, 1991] C. Papadimitriou and S. Yannakakis, Shortest paths without a map, *Theoretical Computer Science* **84**, 1991, 127–150.

[Pearl, 1984] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, MA, 1984.

[Wehner and Wehner, 1990] R. Wehner and S. Wehner, Insect navigation: Use of maps or Ariadne's thread, *Ethology, Ecology and Evolution* **2**, 1990, 27–48.

[Zelinsky, 1992] A. Zelinsky, A mobile robot exploration algorithm, *IEEE Trans. on Robotics and Automation* **8**, 1992, 707–717.