PHA*: Performing A* in Unknown Physical Environments *

Ariel Felner Roni Stern Sarit Kraus
Dept. of Computer Science
Bar-Ilan University
Ramat-Gan, 52900, Israel
{felner,sternr2,sarit}@cs.biu.ac.il

ABSTRACT

We address the problem of finding the shortest path between two points in an unknown real physical environment, where a traveling agent must move around in the environment to explore unknown territories. We present the Physical-A* algorithm (PHA*) to solve such a problem. PHA* is a two-level algorithm in which the upper level is A*, which chooses the next node to expand and the lower level directs the agent to that node in order to explore it. The complexity of this algorithm is measured by the traveling effort of the moving agent and not by the number of generated nodes as in classical A*. We present a number of variations of both the upper level and lower level algorithms and compare them both experimentally and theoretically. We then generalize our algorithm to the multi-agent case where a number of cooperative agents are designed to solve this problem and show experimental implementation for such a system.

Keywords

Search, Shortest path, Mobile agents, A*

1. INTRODUCTION

In this paper we address the problem of finding the shortest path between two points in an unknown real physical environment in which a traveling agent must travel around in the environment to explore unknown territories. Problem spaces of path-finding problems are commonly represented as graphs, in which each state is a node and edges represent the possibilities of moving between the nodes. Graphs can represent different environments, such as roadmaps, games and communication networks. Moving from one node to another in graphs can be either logical operators manipulating the current state or an actual agent moving from one node to the other. The sliding tile puzzle and Rubik's cube are examples of the first class while a roadmap is an example of the second class. Graphs in search problems can be divided into the following three classes:

Fully known graphs - If all the nodes and edges of a graph are stored in the computer then the graph is fully known. These graphs are usually stored in adjacency matrices or adjacency lists.

Very large graphs - Graphs that due to storage and time limitations cannot be completely known and fully stored in any storage device. Many graphs for search problems have exponential number of nodes. For example, the 24-tile puzzle problem has 10^{25} states and cannot be completely stored on current machines.

Small, partially known graphs - Graphs that represent a partially known environment. For example, a traveling agent in an unknown area without a map does not have full knowledge of the environment, but given enough time he can fully explore the environment since it is not very large.

For the first class of graphs - the fully-known graphs - classic algorithms such as Dijkstra's single source shortest path algorithm [6] and the Bellman-Ford algorithm [2] can be used to find the optimal path between any two nodes. These algorithms assume that each node of the graph can be accessed by the algorithm in a constant time. This assumption is valid since all the nodes and edges of the graph are known in advance and are stored in the computer's memory. Thus, the time complexity of these algorithms is measured by the number of nodes and edges that they process during the course of the search.

For the second class of graphs these algorithm are usually not efficient since the number of the nodes of the graph is very large and is usually exponential. Also, only a very small portion of the graph is stored in memory at any given time. The A* algorithm [7] is the common algorithm for finding the shortest paths in large graphs. A* keeps an *open list* of nodes that have been generated but not yet expanded and chooses the most promising node (the *best*) from it for expansion. When a node is expanded it is removed from the open-list and its neighbors are generated and added to the open list. The search terminates when a goal node is chosen for expansion or when the open list is empty. The cost function for A* is $f(n) = g(n) + h(n)$, where $g(n)$ is the distance traveled from the initial state to n and $h(n)$ is a heuristic estimation of the cost from node n to the goal. If $h(n)$ never overestimates the actual cost from node n to a goal then we say that $h(n)$ is *admissible*,

When using an admissible heuristic $h(n)$, A* was proved to be admissible, complete and optimally effective [5]. In other words, with such a heuristic, A* is guaranteed to al-

*This work was partially supported by NSF grant IIS-9907482. The third author is also affiliated with UMIACS.

AAMAS'02, July 15-19, 2002, Bologna, Italy.

ways return the shortest path. Furthermore, any other algorithm claiming to return the optimal path must expand at least all the nodes that A* expands. We assume that any expansion cycle is done in a constant time. This is because it takes a constant time to retrieve a node from the open list, as well as generating all its neighbors, which is applying some operators of the environments on the expanded node. Thus, the time complexity of A* is also measured by the number of generated nodes.

In this paper we deal with finding the shortest path in the third type of graph, i.e., small, partially known graphs which correspond to a real physical environment. With this type of graphs we can not assume that visiting a node can be done in a constant time. Many of the nodes and edges of this graph are not known in advance. Therefore, if we want to expand a node that was not known in advance, then a traveling agent must travel to that node in order to explore it and learn about its neighbors. The cost of the search in this case would be the cost of moving an agent in a physical environment and is proportional to the distance traverssed by the agent. An efficient algorithm would therefore minimize the distance traveled by the agent until the optimal path is found. Note that since we deal here with small graphs, we can omit the computation time that is done by the computer and only focus on the traveling time of the agent.

Unlike ordinary navigation tasks, the purpose of our agent is not to get to the goal node as soon as possible, but rather to explore the graph in such a manner that the shortest path will be retrieved for future usage. On the other hand, our problem is not an ordinary exploration problem where the entire graph should be explored in order to map it. Following are two examples that demonstrate real world applications of such a problem:

Example 1: A division of soldiers is ordered to reach a specific location. The coordinates of the location are known. Navigating with the entire division through unknown enemy territory until the goal is reached is unreasonable and inefficient. It is common in these cases to send scouts to learn and evaluate the best path for the division to pass through. The scouts explore the terrain, and return with the best path for the division to go through in order to reach the destination.

Example 2: Computers on networks can be online or offline at different times, and communication lines may be busy, degrading their throughput. Therefore, many networks cannot be represented as constan, fully known graphs. Transferring large data like multimedia files between two computers on a network can often be time consuming since the data may pass through many communication lines and computers before reaching its destination. Finding the optimal path between these computers will improve the transfer time of large files. Since the network may not be fully known, finding an optimal path between two nodes requires some exploration of the network. An efficient and elegant solution could be to send small packets (operating as scouts) to explore the graph and return the optimal path. Assuming that a computer in the network is recognized only by its neighboring computers, then we are faced with the problem of finding an optimal path in a real physical environment.[1]

2. PROBLEM DESCRIPTION

The problem is to find the shortest path in an unknown undirected graph. The search agent can visit nodes and move from node to node only through existing edges. Each node is assigned a 2-dimensional coordinate, representing its position in the world. Each edge has a weight, which is the Euclidean distance between the two nodes that are connected by it. The input to the problem is the coordinates of the initial and goal nodes. Other nodes are assumed not to be known in advance. The agent is assumed to be located at the initial node. The task find the shortest path in the graph between the initial node and the goal node for future usage. In order to accomplish that, the agent should travel in the graph and explore relevant parts of it.

In this work we assume that when a node is visited by the search agent, the neighboring nodes are discovered as well as the edges that connect them. This assumption is reasonable, because in real life, many intersections have signs indicating the length of the roads that begin there and where they lead to. Even without road signs, when a scout reaches a location he can look around and see the neighboring locations and how far they are from its current location. Generally, assuming that the neighboring nodes are discovered instantly is a common assumption in search problems and algorithms.

Since the goal of the search is to find the best path to the goal, it is clear that the agent must expand all nodes expanded by A* because A* is optimally effective [5]. Therefore, the task of the agent is to visit all the nodes that are expanded by A* as efficiently as possible, i.e., with minimum traveling distance. Below we present the PHA* algorithm for efficient exploration of a graph in order to find the shortest path by single and multipe traveling agent.

3. RELATED WORK

Much research has been done to guide a mobile agent that wishes to explore new and unknown environments in order to learn and map them. Our work is completely different, as we only want to explore the necessary regions of the graph in order to retrieve the shortest path between two nodes. Most of the litreture deal with a physical mobile robot that moves in a real environment. These works usually focus on helping the robot to recognize physical objects in its environment. Instead of mentioning all of these works, we refer the reader to [3], which has a nice survey of these approaches and has some of the most state of the art techniques.

Another class of algorithms is navigation algorithms. A navigation algorithm halts when a path to the goal has been found. Our problem, on the other hand, is to find the *shortest* path to the goal node for future usage. The search continues until the best path to the goal node has been found. Next, we describe briefly some of the work done on navigation in partially known graphs.

Cucka et al. [4] have introduced navigation algorithms for sensory based environments such as automated robots moving in a room. They have used Depth First Search (DFS) based navigation algorithms, choosing the next node that

[1]Our research is on high level abstract graphs and we do not presume in this work to provide a new applicable routing algorithm. Current routing technologies keep large databases that store the best paths from node to node, broadcasting changes in the network and updating the paths if necessary, thus making the graph of the network practically fully known. Our algorithm may be relevant to future network architectures and routing technologies, where routers will not use these databases. This is not far-fetched since the Internet for example, is rapidly growing. There might be a time in the future where storing all the paths will not be feasible.

the agent should go to according to a heuristic function.

Real-Time-A* (RTA*) [10] and its more sophisticated version, LRTA*, are also algorithms for finding paths between two nodes in a graph. In RTA* it is assumed that there is a constraint on the computation time and thus a small search is performed and a preferred next node is chosen and traveled to. In RTA* and LRTA* the merit of a node n is $f(n) = g(n) + h(n)$ as in A*. However, unlike A*, the interpretation of g(n) in RTA* is the actual distance of node n from the current state of the problem solver, rather than from the original initial state. These algorithms are different than ours since they assume that a node can be expanded in the computer's memory even without an actual agent physically visiting this node. Also, these algorithms are designed for large graphs.

Knight [9] has presented a multi-agent version of RTA* called MARTA*. In MARTA* every agent runs RTA* [10] independently. Kitamura et. al. [8] have modified MARTA* by using coordination strategies based on attraction and repulsion for MARTA*. These strategies are employed only in tie breaking situations. When using repulsion strategy, the idea is to spread agents such that each agent intends to maximize its distance from the others. This work has inspired the algorithms presented in this research in respect to the way that we handled our multi-agent mutual decision.

D* [14] is another algorithm that is designed to find the optimal path for a mobile robot. However, this algorithm assumes that the entire graph is known in advance but might change dynamically. D* also starts the search from the goal state and assumes that neighbors of states are known in advance. The task of the agent in D* is actually to find changes in the graph and recompute the path to the goal.

Roadmap-A* [13] is a more sophisticated single agent navigation algorithm, using local-A* algorithm to navigate to nodes chosen by a higher-level algorithm called A^*_ε [12]. Instead of always choosing the best node from the open list, A^*_ε allows the search agent to choose from a set of good nodes. This set is called the *focal* set. The *focal* is a set of nodes from the open list that have f-value larger than the value of the best node by no more than ε. Once the *focal* nodes are determined, a local search is performed to navigate the agent to one of these nodes. In Roadmap-A*, ε is a constant, chosen before the search, which determines the tradeoff between the local search and A*. For example, A_0 is A* while A_∞ is just a local search, choosing at each iteration any node from the open list.

Note that most of the algorithms listed above are navigation algorithms and thus do not guarantee that the path they find is the optimal one.

4. PHA* FOR A SINGLE AGENT

Nodes in our environments can be divided into *explored* and *unexplored* nodes. Exploring a node means to physically visiting that node by the agent and learn about its location and the location of its neighbors. Our new algorithm PHA* actually activates A* on this environment. However, in order to expand a node by A*, this node must be first explored by the agent in order to have the relevant data (edges and neighbors) about that node. PHA* is a 2-level algorithm. The upper level algorithm is a regular A*, which chooses at each cycle which node from the open list to expand. The heuristic function $h(n)$ which is used here is the Euclidean distance of a straight line between the two nodes in question. If the node chosen by the upper level algorithm has not yet been explored by the agent, the lower level algorithm, which is a navigation algorithm, is then activated to navigate the agent to that node and explore it. After a node has been explored by the lower level it is expandable by the upper level. If the chosen node has already been explored, or if its neighbors are already known, then it can be expanded immediately by the upper level. Following is a pseudo-code for the upper level:

```
upper-level(open-list){
.  while(open list is not empty){
.          target = best node from open-list.
.          if unexplored(target){
.                explore(target) by lower level
.          }
.          expand(target)
.  }
}
```

4.1 Lower level algorithms

The upper level algorithm (A*) chooses to expand the node with the smallest f-value in the open list, regardless of whether the agent has visited that node before or not. If the chosen node has not yet been visited by the agent, the lower level instructs the agent to visit that node. We call this node the *target* node for the lower level. In order to reach the target node we must use some sort of a navigation algorithm. We have implemented a number of navigation algorithms for the lower level. We first describe simple algorithms which only use known information on the graph. Then, we present more efficient algorithms, which also explore the graph during the navigation and add new information for the upper level. We assume that the agent is in the *current* node and that it should navigate to the *target* node.

4.1.1 Simple navigation algorithms

Shortest Known Path: Like every best-first search, A* spans the nodes which it generates in a tree which is called the *search tree*. Some of these nodes have already been explored by the agent and all of the edges incident with them are known. This tree, plus the additional edges of the explored nodes, can be seen as a subgraph that is fully known. All nodes of this subgraph are connected because they are all part of the search tree. Using this subgraph, we can calculate the shortest path via known nodes and edges. As mentioned above, finding the best path in a known graph can be easily done, so the agent using the shortest known path simply calculates the best path in the known graph to the target node and travels along this path.

Air Path: Assuming that the agent is able to move freely in the environments and is not restricted to the edges of the graph, we can simply move the agent from the current node to the target node via the straight line connecting these nodes. This method may be relevant when the search agents are highly mobile, but are exploring the environment for a less mobile agent that is confined to travel only along the edges. Note that air-path will never be longer than the shortest known path.

4.1.2 DFS-based Navigation algorithms

In the simple navigation algorithms described above, the exploration of new nodes is done only by the upper level algorithm. Thus, the lower level does not add any new knowledge of the graph and in that sense is inefficient. Here we propose more intelligent approaches that try to find a path to the target node by also looking at unexplored nodes. Two advantages are added by these approaches. The first one is that the paths that are currently known to the agent may be much longer than other paths that have not yet been explored. It may be more efficient to navigate through unknown parts of the graph if they seem to lead to a better path to the target node than the paths currently known in the graph. A more important advantage is that while navigating through unknown parts of the graph, the agent might visit new nodes that have not yet been explored and explore them on the fly. This may save traveling back to those nodes in the future if they would ever be chosen for expansion by the upper level algorithm.

These reasons encourage the usage of a DFS-based navigation for the lower level. In a DFS-based navigation algorithm, the search agent moves to a neighboring node that has not been visited yet in a classical DFS manner. A DFS search backtracks when reaching a dead end and it keeps on searching until it reaches the target node. When there is more than one neighbor, we have to use some sort of heuristic function to evaluate which neighbor is more likely to lead to the target node faster and visit that node first. We have experimented with a couple of such DFS-based navigations which are presented below.

P-DFS - Positional DFS: This DFS-based navigation algorithm sorts the neighbors according to their Euclidean distance from the target node, choosing to try the node with minimum distance to the target node first. This variation was first introduced in [4].

D-DFS - Directional DFS: This DFS-based navigation algorithm sorts the neighbors according to the direction of the edge between them and the current node. It first chooses the node with the smallest difference in angle between the line from that node to the current node, and the line from the current node to the target. In other words, the nodes are prioritized by the directional difference between them and the target node, giving priority to nodes that differ the least. This algorithm was also first introduced by [4].

A*DFS: A*DFS is an improved version of P-DFS. At each step the agent chooses the neighbor that minimizes the sum of the distances from the current node to that neighbor and from that neighbor to the target node. We call it A*DFS since it uses a cost function which is similar to A*.

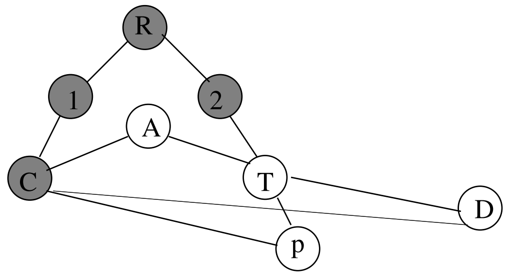

Figure 1: The different navigation algorithms.

Figure 1 illustrates the navigation algorithms listed above. The search agent is currently at node C. The upper level algorithm chooses to expand node T. Dark nodes are the nodes that have already been visited by the agent (explored nodes). These nodes and the edges connecting them are the tree spanned by the upper level A* search. Since T has not yet been unexplored, the lower level algorithm will now navigate to node T. The shortest known path will follow the path of C, 1, R, 2 and finally T. Note that since node A was not explored yet, the path from node C to node R through node A is not known at this point. When the agent uses one of the DFS-based navigations, it will move to node T through nodes P,D or A according to the heuristic used: P for P-DFS, D for D-DFS and A for A*DFS. The benefit of the DFS algorithms is that they explore new nodes during the navigation (nodes P, D or L) and they will not need to go there again if the upper level chooses to expand them.

4.1.3 *Improved A*DFS*

The DFS-based navigation algorithms explore new nodes as they traverse the graph, saving future navigation if these nodes will ever be chosen for expansion by the upper level. While this behavior is very useful as can be seen in the results of the experiments below, we can take these approaches much further.

Suppose that the agent is navigating to a target node. On its way, it may pass near nodes that have small f-value without visiting them because they are not on the way to the target node according to the navigation algorithm. This behavior is counter-productive, since nodes with small f-values are likely to be chosen for expansion by the upper level in the near future. If the agent visits them now, when they are nearby it may save a lot of future traveling effort.

To incorporate this notion, we introduce Improved A*DFS (I-A*DFS). The basic concept is that when navigating to the target node by the lower level and choosing the next node to visit, the heuristic will consider a node's f-value in addition to its approximate distance to the target node. On its way to the target node, I-A*DFS should tend to visit nodes with small f-value on one hand, but on the other hand avoid nodes that are completely off the track of the target node. I-A*DFS therefore chooses to go to the neighboring node of the current node which minimizes the following heuristic function:

$$h(n) = \begin{cases} A^*DFS(n) * \left(1 - c_1 \left(\frac{f(T)}{f(n)}\right)^{c_2}\right) & if n \in OPEN \\ A^*DFS(n) & otherwise \end{cases}$$

T is the target node while n is the agent's neighbor which is being evaluated by the heuristic function. $f(x)$ is the f-value of a node x given by the upper level A*. c_1 and c_2 are constants. If a neighbor n is not in the open list then I-A*DFS treats it just the same as A*DFS and $h(n)$ is the same as in A*DFS . However, if it is on the open list then we also consider the goodness of its f-value. The target node has the smallest f-value on the open list and thus has been chosen for expansion. Therefore, $\frac{f(T)}{f(n)} < 1$. If $f(n)$ is very close to $f(T)$ then the fraction $\frac{f(T)}{f(n)}$ is close to 1 and the overall h-value will be decreased compared to the h-value given to it by simple A*DFS. If however, $f(n)$ is very large then $\frac{f(T)}{f(n)}$ is very close to 0 and the overall h-value will not be decreased. Thus, the agent might choose to visit nodes that are in the open list and have good f-values even if their A*DFS value is not the best. We have searched for the constants that will yield the best results, and our comprehensive experiments have shown that using this formula with c_1=0.25 and c_2=2.5 produces the best results. Our experiments have shown that using I-A*DFS instead of the other navigation algorithm listed above yields better results. More details on generating this formula can be found in [15].

4.2 Experimental results

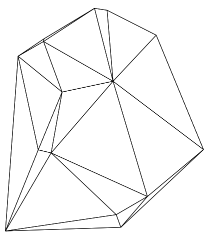

Figure 2: Delaunay graph, 15 vertices.

Since we focus on graphs that represent roadmaps, we have decided to experiment with Delaunay graphs [11]. Delaunay graphs comprise Delaunay triangulations of planar point patterns that are generated by a *Poisson point process* [11] that distributes points at random over a unit square using a uniform probability density function. Delaunay triangulation of a planar point pattern is constructed by creating a line segment between each pair of points *(u,v)*, such that there exists a circle passing through u and v that encloses no other point. This characteristic simulates roadmaps. To construct Delaunay graphs for our experiments, we used the Qhull software package [1], which generates Delaunay tringulations on a square frame with unit size of 1. Figure 2 illustrates Delaunay graph with 15 nodes.

In graphs built by Delaunay triangulation nodes are connected to the nodes that are near them. However, in roadmaps which are the object of this research, sometimes nearby locations do not have roads between them, perhaps due to some sort of an obstacle like a mountain or a river. To imitate this characteristic to our graphs we have also experimented with Delaunay graphs that had some of their edges deleted. Another possible characteristic of roadmaps, is the existence of highways, that connect distant locations. To add this effect to our graphs as well, we have added additional random edges to the Delaunay graph in some of the experiments.

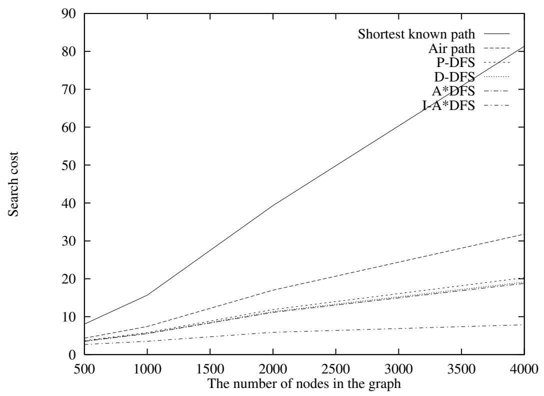

Figure 3: Lower level searches.

Figure 3 shows the traveling distance of the agent, using PHA* with the different lower level algorithms on Delaunay graphs with 500, 1000, 2000 and 4000 nodes. Every data point (here and in all of the experiments below) is an average of 250 different pairs of initial and goal nodes. Note that the entire graph was generated on a square of size 1X1 and that the average optimal path is about 0.55 units. The figure clearly shows that it is efficient to use a more complex algorithm. The I-A*DFS is consistently better than all of the other algorithms for all sizes of graphs. For a graph of size 4000, for example, it outperforms the most simple algorithm by more than a factor of 10. By deleting edges from the graph it becomes more sparse and therefore the agent will run into dead ends more often. When we have deleted edges, all the algorithms needed more traveling effort to find the optimal path. However, the difference between any two algorithms tends to remain the same and I-A*DFS was the best for all these graphs. This behavior was stable for all our experiments in both the single agent and the multi agent environments.

4.3 Upper level algorithm: A* with window

A* expands the nodes from the open list in a best-first order according their f-value. This order is the optimal order when the complexity of expanding a node is $O(1)$. However, in real physical environments in which expanding a node may require an agent to perform costly tasks, it is not always better to expand the current best node. There may be nodes that are near the agent, and while they are not the best nodes in the open list, they are in a high position in the open list, and therefore will be probably chosen for expansion by A* in the next few steps. An intelligent agent might choose to explore them first even though they are not the best nodes in the open list.

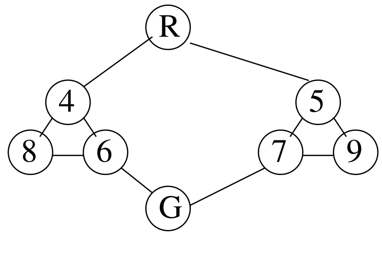

Figure 4: Disadvantage of A*.

Consider Figure 4 where there are two clusters of nodes. The number inside a node is its f-value. An agent that wants to visit the nodes in a best-first order (as A* expands them) will have to travel back and forth from one cluster to the other. A much better approach would be to explore all the nodes in one cluster and then to move to the other cluster thus only traveling once from one cluster to the other.

In order to incorporate this into our algorithm, we generalized A* to what we call Window-A* (WinA*). While A* always chooses to expand the node with the lowest f-value, WinA* creates a set (window) of n nodes with the smallest f-value and chooses one node from that set for expansion. Our window uses the same principal as A^*_ε of [12] which was explained above. After constructing the window we have to choose a node from it for expansion. Our intention is to minimize the traveling effort of the agent and not necessarily to reduce the number of expanded nodes. Thus, rather than choosing only these nodes that have a small f-value, we want to choose nodes that are also close to the location of the agent. While we tried many combinations, we have found that the best combination between these two aspects is to simply multiply them. Therefore, we order the nodes of the window by the following cost function:

$$c(n) = f(n) * d(Curr, n)$$

where $f(n)$ is their f-value and $d(Curr, n)$ is the distance between n to the current location of the agent. Note that if a node with a small f-value is not chosen for expansion then its f-value relative to other nodes in the open list will tend to decrease as time passes. This is because the f-value of new generated nodes is monotonically increasing because we are using a consistent admissible heuristic. Because of that we do not face the problem of starvation.

The way to combine this modified upper level algorithm with the lower level algorithm and then to determine the shortest path is not straight forward and some difficulties arise due to the fact that we do not expand nodes from the open-list in a best-first order. A more comprehensive description of how we have solved these problems can be found in [15] and is omitted here due to lack of space.

4.3.1 Experimental results

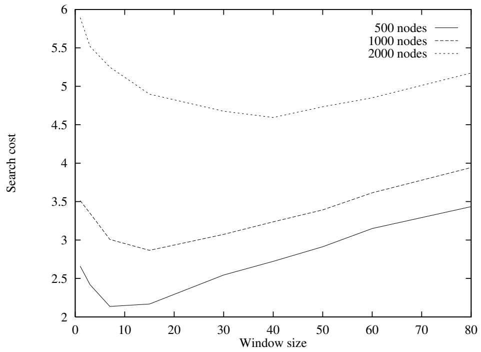

Figure 5: Different windows.

Our experiments have shown that using WinA* as the upper level phase of PHA* has led to a significant improvement in the efficiency of our algorithm. Figure 5 presents the average distance traveled by the search agent until the optimal path is found, as a function of the window size. Here we only used I-A*DFS as the lower level algorithm. The results in Figure 5 show that using a window of size larger than one (which is the trivial case of A*) significantly improves the algorithm performance on all of the different sizes of graphs we have experimented with. We have also found that the optimal size of the window tends to be a function of the size of the graph. The best approximation that we found is that the optimal window size should be 1/50 of the number of nodes in the graph. Thus, the best window for graphs of size 500 was 10 while for graphs of size 2000 it was 40.

At first glance, the improvement of WinA* over simple A* for the upper level seems somewhat modest and is never higher than 30%. This is due to the fact that I-A*DFS is very strong and already explores many nearby nodes. When we use the other navigating algorithms, then the improvement of WinA* over simple A* is much greater. However, when we deal with time of real agents and people even this improvement of 30% is significant and worthwhile.

5. MAPHA*: MULTI-AGENT PHA*

In the following section we generalize the above techniques to the multi-agent case where a number of such agents cooperate in order to find the shortest path. We call the resulting algorithm Multi-agent physical A* (MAPHA*).

We want to divide the traveling effort between the agents in the most efficient way possible. We can measure this efficiency for the multi-agent case using two different measurements. The first is the overall global time needed to solve the problem. The second is the total amount of fuel that is consumed by all agents during the search. Sometimes we might want to minimize the cost of moving the agents, perhaps considering the fuel cost of mobilizing the agents. In that case, it may be wise to move some agents while other agents remain idle. However, if the goal is to find the best path to the goal as soon as possible, idle agents seem wasteful, as they can better utilize their time by further exploration of the graph. In that case, all available agents should be moving at all times. Note that in the single agent case these two measurements converge. Below, we introduce two algorithms for these two aspects namely a *fuel-efficient algorithm* and a *time-efficient algorithm*. We assume that each agent can communicate freely with all of the others agents and share data at any time. Thus, any knowledge obtained by one agent is known to all other agents. We therefore use the model of a centralized supervisor that moves the agents according to the complete knowledge that was gathered by all the agents. This is a reasonable assumption since in many cases there is a dispatcher or some sort of centralized controller that gathers information from the agents and instructs them accordingly. Future research may address a more restrictive communication model, perhaps limiting the communication range or inducing communication errors.

The main idea of the MAPHA* algorithm that we present below is very similar to the PHA* for the single agent case. The upper level chooses which nodes to expand while the lower level navigates the agents to the required nodes. In the multi-agent case we only tried our most powerful techniques for both levels, namely WinA* for the upper level and I-A*DFS for the lower level. The problem that we deal with below for MAPHA* is how to assign the different agents to explore different nodes.

5.1 MAPHA*: Fuel-efficient algorithm

For simplicity, we assume that the amount of fuel consumed by an agent is equal to its traveling distance during the search. Since the purpose of the algorithm is to minimize the amount of fuel spent by the agents, regardless of the overall search time, there is no benefit to moving more than one agent at a time. This is because by moving only one agent, that agent might gain new knowledge of the graph that would allow the other agents to make more intelligent and beneficial moves.

At the beginning, all agents are situated at the initial state. Then, as in the case of the single agent, the upper level defines a window of unexplored nodes from the open list that are potential candidates for expansion. For each pair (a, n), where a is an agent and n is a node from the window, we calculate an allocation cost function of $c(a, n) = f(n) \cdot dist(a, n)$, where $f(n)$ is the f-value of node n and $dist(a, n)$ is the distance from the location of agent a to node n. We then choose an agent and a target node that minimizes that allocation function. In the case of tie-breaking (such as the beginning of the search where all agents are at the initial state), we randomly pick one agent from the relevant candidates. At this stage, the lower level algorithm navigates the chosen agent to the chosen target node from the window to explore this node. Here again, during the navigation, more knowledge about the graph is being learned as many unexplored nodes are visited by the traveling agent. Only when the chosen agent reaches its target is a new cycle for the upper and the lower level activated.[2]

5.2 MAPHA*: Time-efficient algorithm

The time-efficient algorithm is very similar to the fuel-efficient algorithm that was presented above, with only one basic modification. Instead of moving only one agent for every upper level cycle, we now move all of the available agents since we only care about the time spent by the agents and not the fuel. We cannot use the same allocation function here as we did for the fuel efficient algorithm. This is because this allocation formula will cause all agents to always go to the same node, since all the agents begin at the same node and move together. Thus, we would like to distribute the agents to many candidate nodes from the window. Suppose that we have p available agents and q nodes in the window. We want to distribute these p agents to these q nodes as efficiently as possible. Since the f-values of neighbouring nodes are somewhat corolated with each other then nodes with small f-value are more likely to generate new nodes with small f-values than nodes with larger f-value. Therefore, the distribution should be biased in favor of nodes with small f-value. We have tried many variations for the distribution function and found that they all perform well as long as they are biased in favor of nodes with small f-values i.e., nodes from the window that have a small f-value will tend to have more agents assigned to them than nodes with a large f-value. Once the number of agents for each node has been determined, each agent is assigned to one of the nodes in such a manner that minimizes the expected travel distance, i.e., preferably, an agent is assigned to a node with a small distance from it. Further discussion of these distribution functions can be found in [15]. Once again, each agent navigates to its target node with the help of our best lower level algorithm - I-A*DFS. Another upper level cycle begins as soon the the first agent reaches its target node. Note again that computation time of the window and the distribution and allocation functions can be omitted since we only care about the traveling time of the agents.

5.3 Experimental results

The experiments performed here were again on Delaunay graphs of sizes of 500, 1000, 2000, 4000 and 8000 nodes where in some experiments some edges were randomly deleted and new edges were randomly added.

5.3.1 MAPHA: Fuel-efficient algorithm results*

The fuel consumption that we report is the total fuel consumed by all the agents. Note again that the entire graph was generated on a square of size 1X1. Thus, the average optimal path is about 0.55 units.

[2]We have also implemented a more complex algorithm such that whenever a new unexplored node is reached a new upper level cycle is activated. Results were not significantly different and we omit the details of this variation for simplicity. A comprehensive description can be found in [15].

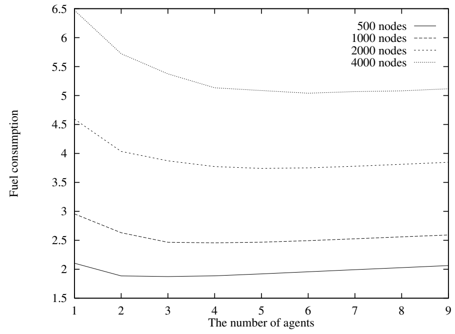

Figure 6: Fuel consumption, simple Delaunay graph.

Figure 6 presents the results of the fuel-efficient algorithm on simple Delaunay graphs as a function of the number of agents used in the search. The figure clearly shows that as more agents are added, the overall fuel consumption decreases until a certain point at which adding more agents tends to increase the overall consumption. Thus an optimal number of agents exists for each of the graphs. This phenomena is due to the fact that A* search is usually characterized by a small number of search regions. Therefore, a small number of agents suffices to cover those regions. Adding more agents will only waste more fuel. Support for this explanation can be obtained by the fact that the optimal number of agents increases as the number of nodes in the graph increases. While the optimal number of agents for a graph of 500 nodes was 2, this number increases up to 7 when searching in a graph of size 4000. Larger graphs have more search regions and thus more agents are needed to explore them.

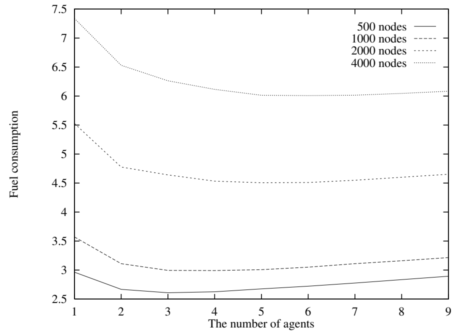

Figure 7: Fuel consumption, sparse graphs.

Figure 7 presents a similar test on Delaunay graphs in which 60% percent of their edges have been randomly deleted. Sparse graphs have fewer paths between the nodes, thus causing the agents to backtrack more often. The overall cost of the search is increased, as can be seen from a comparison of the results from Figures 6 and 7. The overall fuel consumption in the sparse graphs tends to be larger by a factor of 1.5 than the consumption on a simple Delaunay graph. Note that the optimal number of agents also increases on the sparse graph. Here, agents need to backtrack more often and thus more agents will help. As expected, adding random edges to the graphs causes the opposite effect; that is, less fuel was consumed and the optimal number of agents was reduced. Since there are new edges that may connect between nodes, many "shortcuts" are generated and the search can be done at a faster speed and with a smaller number of agents.

5.3.2 MAPHA*: Time-efficient algorithm results

Here we report the results of the Time-efficient algorithm. The overall search time is actually the distance that any one of the agents has traveled until the best path to the goal node is found since all agents are always moving.

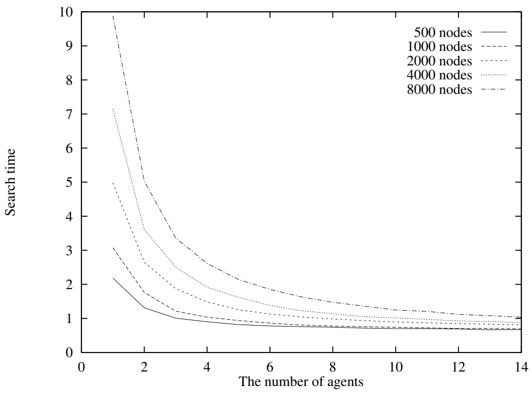

Figure 8: Time consumption, Delaunay graphs.

Figure 8 presents the results of the time-efficient algorithm on Delaunay graphs as a function of the number of agents used in the search. Note that the search time can never be shorter than the time that it takes to travel through the best path to the goal. As the results show, adding more agents is always efficient since we only measure the overall time that has elapsed until the goal is found. What makes our algorithm interesting and efficient is the fact that as we add more agents, the search time asymptotically converges to the size of the shortest path. This means that with many agents, one of them actually travels in that shortest path and almost never deviates from it. For example, the average size of the best path in graphs of size 500 is approximately 0.55 and indeed with many agents the overall time tends to converge to that number. With 14 agents the overall search time was 0.7.

6. CONCLUSION AND FUTURE WORK

We have addressed the problem of finding the best path to a goal node in unknown graphs that represents physical environments. We have presented a two-level algorithm, PHA*, for these environments for a single search agent, as well as MAPHA* for multi-agents. We have experimented on several variations of Delaunay graphs, with up to 8000 nodes. The most complex single agent algorithm yielded much better results than other trivial implementations of A*. The results on the fuel-efficient algorithm have shown that using more agents is beneficial only to some extent. This is because all of the agents are initially positioned at the same location and they all consume fuel for all their moves. For the same reason the benefit of using the optimal number of agents as apposed to only one agent is modest. The results of the time-efficient algorithm are very encouraging since the search time quickly converges to the optimum as the number of search agents used in the search increases.

This work can be taken further in the following directions:

1. We have assumed that when an agent reaches a node, it can learn the locations of all of its neighbors. In many domains this model is not valid and the location of a node is known only when an agent actually visits it. Further research should be done in order to implement our algorithms in such a model.

2. We have assumed a centralized model where all the agents share their knowledge at all times. Future work can assume other communications paradigms.

3. In many cases, both time and fuel are important. Further work may find a way to combine the cost of both time and fuel. **4.** We have used traveling agents to solve the shortest path problem. A similar mechanism might be used to solve other known graph problems.

7. REFERENCES

[1] C. B. Barber, D. P. Dobkin, and H. Huhdanpaa. The quickhull algorithm for convex hull. *ACM Trans. on Mathematical Software*, 1996.

[2] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[3] M. Bender, A. Fernandez, D. Ron, A. Sahai, and S. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proc. of STOC-98*, pages 269–278, 1998.

[4] P. Cucka, N. Netanyahu, and A. Rosenfeld. Learning in navigation: Goal finding in graphs. *IJPRAI*, 10:429–446, 1996.

[5] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *JACM*, 32(3):505–536, 1985.

[6] E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[7] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SCC-4(2):100–107, 1968.

[8] Y. Kitamura, K. Teranishi, and S. Tatsumi. Organizatonal strategies for multiagent real-time search. *In Proc. of ICMAS*, pages 150–156, 1996.

[9] K. Knight. Are many reactive agents better than a few deliberative ones. In *Proc. of IJCAI-93*, pages 432–437, 1993.

[10] R. E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(3):189–211, 1990.

[11] A. Okabe, B. Boots, and K. Sugihara. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams.* Wiley, Chichester,England, 1992.

[12] J. Pearl and J. E. Kim. Studies in semi-admissible heursitics. *IEEE Trans. on PAMI*, 4(201):392–400, 1982.

[13] L. Shmoulian and E. Rimon. Roadmap-A*: an algorithm for minimizing travel effort in sensor based mobile robot navigation. In *In Proc. of ICRA*, 98.

[14] A. Stentz. Optimal and efficient path planning for partially known environments. *In Proc. of ICRA-94*, pages 3310–3317, 1994.

[15] R. Stern. Optimal path search in unknown physical enviroments. *M.Sc thesis, CS Dept., Bar-Ilan University, Israel, Available at http://www.cs.biu.ac.il/ strenr2*, 2001.