Distributed Navigation in an Unknown Physical Environment *

Arnon Gilboa, Amnon Meisels
Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
{gilboaar,am}@cs.bgu.ac.il

Ariel Felner
Department of Information Systems Engineering
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
felner@bgu.ac.il

ABSTRACT

We address the problem of navigating from an initial node to a goal node by a group of agents in an unknown physical environment. In such environments mobile agents must physically move around to discover the existence of nodes and edges. We assume that agents communicate by exchanging messages about their discoveries, their current locations and their intended plans. We also assume that an agent can only communicate with a predefined set of *neighboring* agents. A distributed algorithm, which is run independently by each agent, is presented. Given the current knowledge of the agent about the environment and the positions and intentions of other agents, the algorithm instructs the agent where to go next. An experimental evaluation of the algorithm is presented, with constrained and liberal neighborhood schemes. Results show that it is more beneficial to have a constrained neighborhood scheme because with this scheme the distributed intelligent behavior of agents generates a spread of knowledge throughout the environment more efficiently. Agents reach the goal node fast and the length of the path that they find is very close to that of the optimal path.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed AI—*Intelligent Agents*

General Terms

Algorithms, Experimentation

Keywords

Mobile agents, Distributed navigation

*Supported by the Lynn and William Frankel center for Computer Sciences and the Paul Ivanier Center for Robotics and Production Management.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS'06 May 8–12 2006, Hakodate, Hokkaido, Japan.
Copyright 2006 ACM 1-59593-303-4/06/0005 ...$5.00.

1. INTRODUCTION

We address the problem of navigation from an initial node to a goal node in an unknown physical environment, by a group of agents. To abstract away from problems of real robots and obstacle avoidance [2, 3] and to concentrate on navigation, we use the notion of a partially known physical graph [4]. The environment is represented by a planer graph where nodes are physical points and a weight of an edge is the Euclidean distance between the two nodes it connects. In such environments, only when an agent physically visits a node, the weights of outgoing edges and the adjacent nodes are revealed. Therefore, agents must move around the graph in order to find information about other nodes and about weights of edges.

1.1 Navigation versus optimal path finding

Let us start by making a distinction between navigating to a node and finding the optimal path from an initial node to a goal node. When navigating, the task is to reach the goal node as soon as possible. At every step the task of the navigator is to find a path from its current node to the goal node. Once the navigator reaches the goal, the task is accomplished. Many times, the navigator did not travel via the shortest path between the initial and goal node and does not even know this shortest path upon arriving to the goal node. Much of the literature on navigation deals with physical mobile robots that move in a real environment. The published research usually focuses on the issue of assisting the robot to recognize physical objects in its environment. We refer the reader to [2, 3] which contain extensive surveys of various approaches and state of the art techniques of robot navigation . The present work is different, as it represents the physical world by a discrete abstract graph which is initially unknown.

Finding an optimal (shortest) path is different than navigation. Here, even if we have some path from the initial node to the goal node, we must perform a systematic search on the graph in order to guarantee the optimality of the path. A common method for finding the shortest path in graphs is to use the A* algorithm [5]. A* keeps an *Open-list* of generated nodes and expands them in a best-first order according to a cost function defined by $f(n) = g(n) + h(n)$. In standard A*, $g(n)$ is the distance from the initial node to node n, and $h(n)$ is an *admissible* heuristic estimate of the cost of the path from node n to the goal node. $h(n)$ is *admissible* if it is always a lower bound on actual cost from node n to the goal (e.g. Euclidean distance between the two nodes in a physical environment).

Usually, A* (or its variants, e.g., IDA*) are applied to exponential combinatoric problems. In such domains, an A* expansion cycle is carried out in constant time as it takes a constant amount of time to retrieve a node from the open-list and to generate all its neighbors by applying domain-specific operators to the expanded node. Therefore, the time complexity of A* is usually measured in terms of the number of generated nodes. This is not the case in physical graphs where an agent needs to physically travel to a node in order to explore it. The Physical A* algorithm (PHA*) and its multi-agent version MAPHA* [4] find the optimal path between two points in a partially known real physical environment with either one or many mobile agents. These algorithms modify the A* algorithm to work on graphs with physical characteristics where agents need to physically travel to a node in order to learn about its neighbors. The focus of these algorithms is to minimize the travel effort of the agents. In MAPHA*, a complete sharing of knowledge between agents is assumed. This could be achieved by a supervisor agent coordinating all agents or by a global database which holds the entire knowledge. A reminiscent approach is the D* algorithm family (e.g., D*-lite) [6] where a mobile robot needs to find the shortest path in an environment that was completely known but changes dynamically over time.

The present paper addresses the navigation problem in the partially known physical environments by a distributed group of agents. Agents are autonomous entities, processing information received from other agents and making their own navigational decisions. The task is to reach the goal node as soon as possible. When the first agent reaches the goal the task is achieved and the agents terminate. Agents explore the graph by moving on it, sharing their discovered nodes and edges as well as their current position and running a distributed navigation algorithm (presented below) to arrive at the goal node. Each agent runs the same algorithm but uses its own partial knowledge of the environment. A distributed algorithm is in general more fault tolerant and is expected to reduce communication, compared to a centralized algorithm [8]. A distributed navigation algorithm is naturally scalable to a large number of navigators.

1.2 Communication models

There are many models of communication in multi-agent systems that allow agents to cooperate and coordinate their decision making. The most trivial model is a complete knowledge sharing where any new discovery of an agent is immediately shared with all the other agents. Other models restrict the level of communication.

The present study focuses on a natural and realistic model for the communication among agents where each agent can only communicate with a subset of the agents - its *neighboring agents* (or *neighbors* in short). The *neighbor* relation is symmetric. We assume that agents can communicate with one another only if the distance between them is below a given radius R. We call this the*radius constraint*. To satisfy these two constraints, we enforce neighboring agents to stay close to one another at all times in order to be able to communicate. Neighboring agents exchange relevant information, such as their current location and the length of the outgoing edges. In each step, agents receive messages from other agents, decide whether and where to move, and send updates to their neighboring agents.

We use two performance measures - the path traveled by the first agent arriving at the goal node and the quality of the best path found compared to the optimal path. Our agents are designed to reach the goal as fast as possible and to minimize their travel effort. This corresponds to the first measure. On the other hand, during navigation they also explore many nodes and the shortest path that will be known to them at the end might improve. This corresponds to the second measure. These two measures might be in conflict with each other. This is known as the exploration versus exploitation conflict. In this paper we show that even though we are mostly concerned with the travel effort measure, a good balance between the two measures can be achieved. Indeed, experimental results show that when the algorithm terminates the quality of the best known path is very close to the optimal path.

The DisNAV algorithm was implemented on a distributed asynchronous simulator. Experiments were run on different graphs and different number of agents and neighborhood schemes. Results show that the proposed distributed navigation algorithm generates an efficient spread of knowledge throughout the environment. Agents reach the goal node very fast and the length of the path that they find is close to that of the optimal path. We have considered three different communication schemes and show that using a constrained scheme on the defined neighborhood produces better results than using a more liberal scheme.

2. NEIGHBORHOOD SCHEMES

The main motivation of the present study is to look at multi-agent navigation as a distributed problem, where agents communicate only with their neighboring agents. Given an agent A, its *neighborhood* N is a subset of the agents. A must coordinate its moves with all the agents in N. We have defined three different neighborhood schemes: *static neighborhood*, *unconstrained dynamic neighborhood*, and *constrained dynamic neighborhood*. In all these schemes, we have defined a radius parameter R such that agents can only receive messages sent by an agent distanced away no more than R units. We call this the *radius constraint*. However, the schemes differ in the definition of neighborhood. Extreme values for the radius R significantly influence the entire behavior of the multi agent system. With large R, the radius constraint becomes meaningless and the communication scheme is similar to the full knowledge sharing. With small R, moves are limited and we get close to the behavior of a single agent. Thus, the value for R should be carefully chosen.

2.1 Static neighborhood

In order to simplify the definition of the distributed navigation problem, we have defined the notion of static neighborhood which enable an easy definition of a static connected communication network of agents. In a static neighborhood each agent is assigned a set of neighbors and this set is fixed throughout the entire navigation. An agent has to coordinate its moves with all its neighbors therefore must keep the radius constraint with all them. The rationale behind this restriction is that new discoveries of an agent will always be propagated to its neighbors. Static neighborhoods can cause some restriction on agent moves since the *radius constraint* may prevent agents from making some of their possible moves, if they move too far from their neighbors. Note that in the static neighborhood scheme an agent does

not communicate with an agent which is not its neighbor even if it is located nearby.

2.2 Constrained dynamic neighborhood

In this scheme, an agent needs to coordinate its moves with K agents within the radius R. However, unlike the *static neighborhood* version where the set of neighboring agents is predefined, here, any K agents suffice. The only requirement is to keep the *radius constraint* with any K agents whose identities can change over time. Agents broadcast their move proposal and wait for any K approvals from agents within the radius R.

2.3 Unconstrained dynamic neighborhood

This is the most liberal scheme. Here, when an agent chooses a move, it broadcasts its decision and performs that move without the need to coordinate it with anyone. Of course, only agents within the radius R actually receive its move decision and other messages. The pitfall of this scheme is that an agent can travel by itself to a distant location where it cannot communicate with any other agent. This will weaken the knowledge sharing of the entire multi-agent system as this agent becomes isolated.

3. DISTRIBUTED NAVIGATION

We now turn to describe the DisNAV algorithm, focusing first on the main case of *static neighborhood*. We then present the modifications that are needed for the *constrained dynamic neighborhood* and the *unconstrained dynamic neighborhood* cases.

3.1 DisNAV for static neighborhood

The DisNAV algorithm is run by each agent independently. Agents communicate with their neighbors, exchanging messages which contain information such as their current position, node discoveries and move proposals. Agent navigation is based on its known information which is gathered by its own discoveries as well by information received in messages from other agents. Agents keep their local information in three main data structures: *KnownGraph*, *AgentView* and *Open-list*, all are described below.

KnownGraph - The *KnownGraph* is the part of the world currently known to the agent. *KnownGraph* is a subgraph of the entire environment modeled as a weighted graph. We assume that all agents start from the same initial node. Since agents send each node discovery to their neighbors (as will be described below) the *KnownGraph* will always be a connected graph. Each agent uses its own *KnownGraph* to calculate distances between nodes during the execution of the algorithm. In addition, when the algorithm terminates, *KnownGraph* is used for calculating the best path between the initial node and the goal node.

AgentView - The *AgentView* data structure, contains important information about each of the neighbors of the agent.
AgentView is updated by messages received from the neighbors. In particular, *AgentView* of an agent contains the current position and the current target or the move proposals of each of its neighbors.

Open-list - Each agent maintains a local *Open-list* of nodes which are in the frontier of its *KnownGraph*. Nodes are inserted to the *Open-list* when they are first known to the agent either by its own discovery or by a message from one of its neighbors. Similarly, a node is being deleted from the *Open-list* when the agent (or one of its neighboring agents that has just sent him a message) physically visits it and discovers all its neighbors (which are themselves inserted to the *Open-list*). The *Open-list* is maintained as a priority queue according to an intelligent cost function described below. Agents choose the best node on the *Open-list* and propose to physically travel to it.

We now turn to present the details of the DisNAV algorithm. The pseudo-code for the algorithm is given in Figure 1. The navigation is done as follows. First, an agent picks a *target* node from its *Open-list* and proposes a *move* to that target. Each proposed move should be approved by all neighbors before its actual performance. When the move is approved by the neighbors, the agent physically navigates from its current node to the target node.

The main procedure run by each agent is ***DisNAV*** which is a message driven distributed algorithm. In each *cycle* of DisNAV an agent reads all the received messages (lines 5-6) and decides about a proper resulting operation. The resulting operation can be a message sending (e.g. NODE, APPROVAL etc.) and can also include a performance of an approved move. An agent running the algorithm can be in one of several possible logical states. Different events can change the state of an agent. The initial state of an agent is READY. Each agent runs a loop, which ends when the agent enters the TERMINATED state.

The basic state is the READY state. In this state the agent selects a *target* node from the *Open-list*. The *target* is chosen to be the best node on it its *Open-list* according to the cost function described below. Then, the agent sends an *OK?* message to its neighbors in which it proposes to move to this target and asks for their approval (***propose-Move***, lines 51-54). The state of the agent is now changed to WAIT_FOR_APPROVAL.

Note that we ensure that targets that break the *radius constraint* between an agent and the current location of its neighbors will not be selected by giving them a cost of infinity. Therefore, sometimes agents choose targets not in the front of the *Open-list* and might even decide to remain idle.

Receiving an OK? message (lines 19-23), an agent has two choices. It can either approve the proposal, replying with an APPROVAL message or reject the move using a NOGOOD message. We say that move proposals of two neighboring agents are in *conflict* if performing them both at the same time will break the *radius constraint* between them. Such a conflict is resolved in favor of the move with the lower cost. In such a case, a NOGOOD message is sent only by the agent with the better move. When that proposing agent receives a NOGOOD (lines 29-31), the state returns to READY and the move proposal is canceled.

When the proposing agent receives APPROVALs from all its neighbors (lines 25-28), its state changes to IN_TRANSIT. In this case the target node is removed from the *Open-list*, and the agent starts performing the *move* to the target. A *move* includes a sequence of navigation *steps*. At each *step* the agent chooses the neighbor w that minimizes the sum of the distances from the current node v to w and from w to the target node t. This cost function is called A*DFS in [4] since it uses a cost function which is similar to that of A*, i.e., $f(n) = g(n) + h(n)$. Note, that this cost function is used here locally to find a path from the current node to the target node.

Upon agent arrival to a node (line 37), the procedure ***uponArrivalToNode*** (lines 40-50) is called. If it is the goal node, the agent changes its state to TERMINATED and sends a TERMINATE message. Each agent receiving such a message (lines 32-34), sends it to its neighbors and changes the state to TERMINATED. Otherwise, if the node is not the goal, the agent learns about this node's neighbors and sends a NODE message, to inform its neighboring agents about its current position and about the node's neighbors. If the current node is in the *Open-list* this node is *expanded*, i.e., it is removed from the *Open-list*, and the node's neighbors which where not visited yet, are inserted to the *Open-list*. If the current node is the target node (lines 48-50) the state returns to READY and the agent considers another move proposal.

Agent receiving a NODE message (lines 8-18) updates its *AgentView* with the sender location and updates its *KnownGraph* with the new nodes and edges discovered. Of course, in case the node is in the *Open-list*, it is immediately expanded by the agent without the need for the agent to physically visit that node. This is the main contribution of data sharing among neighboring agents.

3.2 Unconstrained dynamic neighborhood

The unconstrained dynamic neighborhood scheme is a simplified variation of the static neighborhood algorithm. When an agent decides on its target (the best node in its *Open-list*), it broadcasts a MOVE message declaring it. The agent changes its state to IN_TRANSIT and removes the target from the *Open-list*. This step replaces the sending of an OK? move proposal and waiting for APPROVAL and NOGOOD messages as done in the static neighborhood (or the constrained dynamic neighborhood). The OK?, APPROVAL and NOGOOD messages, as well as the state WAIT_FOR_APPROVAL, become irrelevant in this scheme.

3.3 Constrained dynamic neighborhood

In this scheme, each agent has a dynamic set of *current neighbors* all of them are within the radius R. When all agents start at the same point, the set initially includes all the other agents. Otherwise, the set is modified according to agent positions as they are received in NODE messages. During navigation, agents are added and removed from the set according to the moves performed. A *neighbor* in this context is therefore a member of the *current neighbors* set. The changes that the constrained dynamic neighborhood algorithm imposes on the static neighborhood version are the following:

1. Agents propose only moves which are consistent with all their *current neighbors* locations.
2. Agents wait for k APPROVALs. When they are received, the agent broadcasts a MOVE message declaring its target, changes its state to IN_TRANSIT and removes the target from the *Open-list*.
3. If an OK? is received from an agent which is not a *neighbor* but within the radius R, send an APPROVAL and add the agent to the set of *current neighbors*.
4. If an APPROVAL is received from an agent which is not a *neighbor* and if the move proposal is preformed then add the agent to the *current neighbors*.

```
DisNAV:
1.   state ← READY; node ← initial; proposal ← null
2.   expand node
3.   update KnownGraph and Open-list
4.   while(state ≠ TERMINATED)
5.     while(not msgQueue.isEmpty)
6.       msg ← msgQueue.getFirst
7.       switch msg.type
8.         NODE:
9.           update KnownGraph, AgentView
9a.            and Open-list
10.          if(state=READY)
11.            proposeMove
12.          else if(state=WAIT_FOR_APPROVAL)
13.            if(move proposal is redundant)
14.              state ← READY
15.              ignore proposal
16.            else if(move proposal should be updated)
17.              update proposal
18.              send(OK?,neighbors)
19.        OK?:
20.          if(self already sent a better conflicting
20a.             move proposal)
21.            send(NOGOOD,sender)
22.          else
23.            send(APPROVAL,sender)
24.        APPROVAL:
25.          if(state=WAIT_FOR_APPROVAL
25a.             and all approved)
26.            state ← IN_TRANSIT
27.            target ← proposal
28.            remove target from Open-list
29.        NOGOOD:
30.          if(state=WAIT_FOR_APPROVAL)
31.            state ← READY
32.        TERMINATE:
33.          state ← TERMINATED
34.          send(TERMINATE,neighbors)
35.    if(state=IN_TRANSIT)
36.      move to a neighboring node, navigating to target
37.      uponArrivalToNode
38.    else if(state=READY and proposal=null)
39.      proposeMove

uponArrivalToNode:
40. if(node=goal)
41.   state ← TERMINATED
42.   send(TERMINATE,neighbors)
43. else
44.   discover node neighbors
45.   send(NODE,neighbors)
46.   if(node in Open-list or node=target)
47.     update KnownGraph and Open-list
48.     if(node=target)
49.       state ← READY
50.       proposeMove

proposeMove:
51. proposal ← get best node from Open-list
52. if(proposal is worthwhile)
53.   send(OK?,neighbors)
54.   state ← WAIT_FOR_APPROVAL
```

Figure 1: The DisNAV Algorithm

5. If a conflicting MOVE is received from a *neighbor*, remove the agent from the *current neighbors*.

4. COST FUNCTION HEURISTICS

We now turn to describe the cost function for choosing a target node. Each agent running the DisNAV algorithm, owns a local *Open-list* which includes nodes discovered during navigation, as well as nodes discovered by neighboring agents. The nodes are chosen as targets in a best-first order according to a cost function[1].

The ordinary cost function used by A* and other search algorithms for finding optimal paths is $f(n) = g(n) + h(n)$, where $g(n)$ is the distance traveled from the initial node to the node n, and $h(n)$ is a heuristic estimate of the cost of traveling from node n to the goal node. A reminiscent algorithm is Real-Time-A* (RTA*) [7]. RTA* selects the node with the best cost in the *Open-list*. The problem solver then moves one step along the path to that node, and the search continues from the new state of the problem solver. The cost function for an arbitrary node n in RTA* is also $f(n) = g(n) + h(n)$ but in this case $g(n)$ is an estimation of the distance of node n from the current node and not from the initial node.

DisNAV deals with physical graphs where nodes can be expanded only after an agent physically visits them and learns about their locations and outgoing edges. This is completely different from RTA* which assumes that a node can be expanded in the computer's memory without an agent having to physically visit that node. However, the cost function used in DisNAV treats $g(n)$ similarly to RTA* and is measured according to the current location of the agent. Thus, at every step the cost values of the entire set of nodes in the *Open-list* are changed so that g will reflect the current location. The best node in the open list according to the cost function is selected and becomes the target node of that agent.

A good cost function for our purpose needs to combine both navigation and exploration. In other words, we want to get to the goal node as fast as possible but also to learn about new regions of the graph in order to provide useful information to other agents. For this purpose, one can use combinations of the following two functions:

- $f(n) = g(n) + h(n)$ where g is measured from the current node and h is the Euclidean distance from n to the goal. This is the navigation part which leads the agent to the goal as fast as possible.
- $vc(n)$ - a counter that keeps track of the number of visits to the node n done by other agents. $vc(n)$ is initially set to 0 and is incremented every time the agent is informed of a visit in n.

When combining both cost functions then nodes with small f-value are preferred, but on the other hand, so are nodes with small vc-value. Such nodes are on a promising path to the goal and are distanced away from paths chosen by other agents. A number of combinations between f and vc were tried. The best results were obtained by using

$$c(n) = f(n) \cdot [1 + p \cdot vc(n)]$$

for choosing a target node from the open list. The constant p defines the weight of the exploration in the cost function. Our extensive empirical studies have shown that $p \approx 0.3$ produces the best performance.

[1]When an agent knows that one of its neighbors is navigating to a target node, this node can be removed from the *Open-list*, so the agent will not choose the same target

5. CORRECTNESS

$DisNAV$ terminates when the first agent reaches the goal node. An important role of the algorithm is to delete nodes from the open list and to add new nodes to it. Also, it is easy to see that a node cannot be inserted to the *Open-list* more than once. In other words, the frontier of the *KnownGraph* is always expanding. Therefore, as long as we keep doing this, more and more new nodes will be discovered and since the graph is finite we will eventually reach the goal node. There are, however, some "pathological" cases where the entire multi-agent system enters a *deadlock* and cannot continue because there is not even one agent with a legal target node in its *Open-list*. This happens when for each agent all the nodes in the *Open-lists* break the radius constraint with at least one of its neighboring agents. Of course, deadlocks are not possible in the unconstrained neighborhood scheme since the radius constraint is not kept.

5.1 Deadlock elimination

We want to eliminate such deadlocks and ensure the termination of the algorithm even when keeping the *radius constraint*. We provide a solution to the static neighborhood scheme. A solution to the dynamic constrained neighborhood scheme can be easily modified since the basic principle is similar. The basic idea for deadlock elimination is to choose a *leader* agent. When a deadlock occurs, the leader chooses a move from its open-list and perform it. The other agents are forced to move in such a way that the radius constraint is kept given that leader performs its chosen move. We choose the leader that will benefit the most by its chosen move. The deadlock elimination steps are the following:

1. **Initialization -** An agent with no legal move initializes the deadlock elimination process by sending an *INVITE* message to all its neighbors declaring its wish to start a deadlock elimination process.

2. **Joining -** Agents receiving the *INVITE* can *join* this process and invite their neighbors as well or continue their navigation as usual, as long as they have legal moves in their *Open-list*.

3. **Leader Election -** All the agents which joined the deadlock elimination process pick a leader. The agent with the move of the best cost (of all the *Open-lists*) is elected and allowed to perform its move. This is done using a leader election algorithm [1].

4. **Distributed Search -** The agents solve the distributed constraint satisfaction problem (DisCSP) [10, 12] of finding a legal move for each of them. They run a DisCSP search algorithm (e.g. ABT [11, 12]) where moves of an agent are sorted according to their cost.

5. **Moves are performed -** The agents perform their moves found by the distributed search.

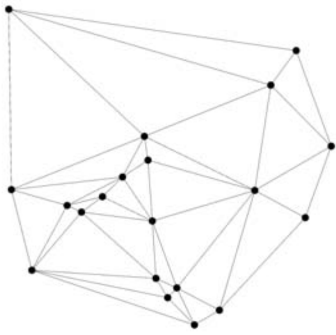

Figure 2: Delaunay graph of 20 nodes.

6. EXPERIMENTAL EVALUATION

DisNAV was evaluated experimentally on random Delaunay graphs [9] which are common testbeds (e.g., [4]) for simulating physical graphs. They are derived from Delaunay triangulations computed over a set of planar point, generated by a *Poisson point process* [9]. Points are distributed at random over a square, using a uniform probability density function. In regular Delaunay graphs, each node is connected by edges to its nearest neighbors as illustrated in figure 2. This property may not always apply to a real road map. For example, nearby geographic locations may not always be connected by a road segment, due to the existence of obstacles like a mountain or a river. To capture this additional characteristic, sparse Delaunay graphs [4] were considered. Instances of these variants can be easily obtained from regular Delaunay graphs by random deletion of edges. We define the *density* as the fraction of Delaunay edges left in the graph. The graphs were generated to be connected, by first adding a spanning tree edges and then adding random Delaunay edges up to the desired *density.*

In all our experiments we used Delaunay graphs which were randomly generated on a square with 100x100 units. For each experiment we generated 100 different graphs and selected a pair of nodes with maximal distance as the initial and goal nodes. Every data point in the figures below is the average over these 100 cases.

6.1 Comparing the different neighborhoods

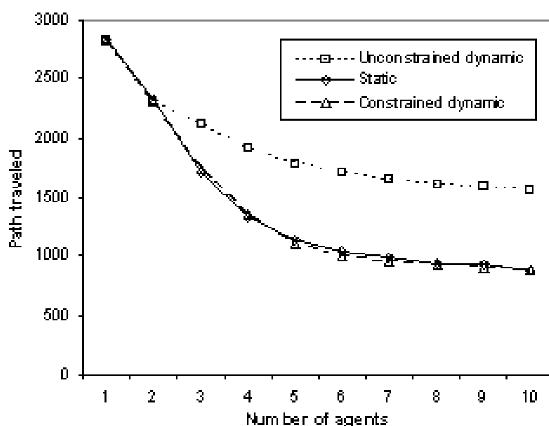

Figure 3: Different neighborhood schemes

Figure 3 presents the length of the path traveled by the first agent reaching the goal node, as a function of the number of participating agents on a Delaunay graph with 500 nodes. The radius R was set to 25 units, which is much smaller than the diameter of the problem and information is only shared locally. There are three curves in the figure, one for each neighborhood schemes. The neighborhood topology for the static neighborhood scheme was a chain where each agent was defined to be a neighbor of its predecessor and successor in the chain. Thus, each agent (except for the agents at the two ends of the chain) has two neighbors, but no 3-agent cliques exist. Similarly, the size of the dynamic constrained neighborhood was set to 2.

For all neighborhood schemes, as more agents are added the first agent reaches the goal node after traveling a shorter path. However, the figure clearly shows that the two constrained neighborhood schemes significantly outperform the unconstrained neighborhood scheme and that the path to the goal is found much faster with the constrained schemes. The two constrained schemes produced results which are not significantly different. We have performed an extensive number of experiments with all three neighborhood schemes and obtained results with the same tendency.

The unconstrained scheme performs rather poorly. If an agent is not constrained to actually stay close to other agents it might work on its own task and run away from the others. With this behavior, a valuable knowledge sharing is not guaranteed and the performance of the multi-agent system derogates. This teaches us the following valuable lesson in multi-agent communication - in order to have an efficient multi-agent system we should build a strong and constrained communication scheme to guarantee the knowledge sharing and data spread.

The two constrained schemes (static and dynamic) behave similarly. However, there are a number of advantages in the static neighborhood scheme. First, this scheme is simpler as an agent always has the same set of neighbors and thus we have a fixed and connected communication graph. This allows a simpler and faster termination of the entire task. Furthermore, the number of messages sent in this scheme is much smaller than the dynamic scheme. In the static scheme an agent only sends messages to the specific set of neighbors while in the dynamic neighborhood scheme the agents actually broadcast their discoveries and proposals to the entire set of agents. Therefore, for the remaining part of the experiments we focus on the static neighborhood results.

6.2 Static neighborhood

The next set of experiments compares different densities on sparse Delaunay graphs with 1000 nodes. The radius R was set to 75 units. Note that since agents communicate only with a predefined (static) set of agents, the relatively large radius (75) does not cause global information sharing. The number of agents in the experiments was varied from one to ten and the neighborhood topology was again a chain where each agent has two neighbors.

Figure 4 shows the length of the path traveled by the first agent reaching the goal node, as a function of the number of participating agents. The three curves correspond to graph densities of 0.3, 0.35 and 0.4. Again, adding more agents improves the results. It is also clear that as the density of the graph grows, the path traveled by the first agent is shorter. In graphs with high densities, agents have more alternative

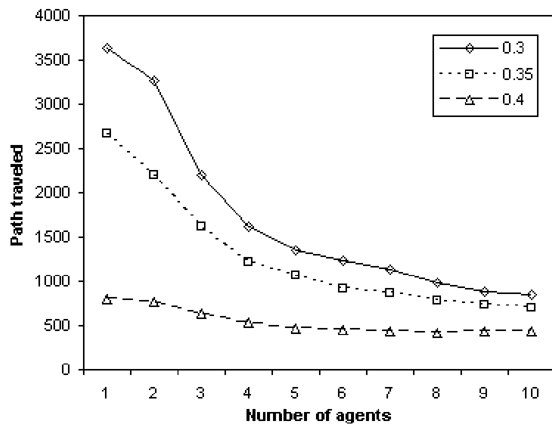

Figure 4: Path traveled (1000 nodes)

paths to the goal and there are only a few dead-ends. Such graphs are easy for navigating and agents can find their way by simple directional navigation. They do not need a lot of information from their neighbors. Thus, the improvement for adding more agents is small and a single agent is almost as good as many agents. When the graph is sparse and there are many dead-ends, the distributed algorithm outperforms a single agent. In other words, the first agent from the group reaches the goal much earlier than a single agent.

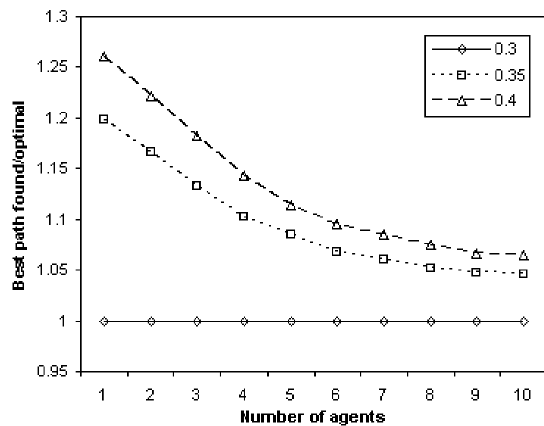

Figure 5: Best path found (1000 nodes)

Figure 5 presents the ratio between the length of the best path known when the first agent reaching the goal node and the optimal path. It is clear from figure 5 that as the number of participating agents grows, the first agent to reach the goal node knows a better path, compared to the optimal. In graphs with low densities (0.3 and below) the best known path is actually the optimal path. This is an interesting result. In such graphs agents cover a much larger portion of the graph due to dead-ends, so on arriving to the goal node, their *KnownGraph* covers a larger portion of the complete graph. In other words, the higher cost needed to reach the goal node (figure 4) reveals a path that is closer to the optimal. The combined result of figures 4 and 5 can be stated as follows. Using a minimal number of neighbors and using a large number of agents, the harder problems (low density graphs) are solved close to optimality.

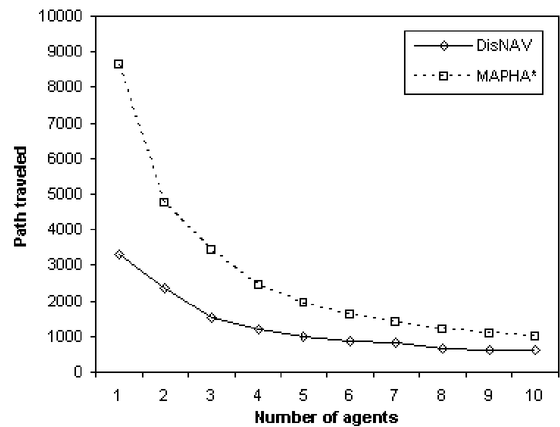

Figure 6: DisNAV vs. MAPHA* (500 nodes)

Figure 6 presents a comparison between DisNAV and MAPHA*, the multi-agent physical A* algorithm of [4], on a graph with 500 nodes and a density of 0.3. Note again, that MAPHA* actually activates a systematic global search and is guaranteed to return an optimal solution. The two curves correspond to the length of the path traveled by the first agent running DisNAV that reached the goal node, and the path traveled by the agents performing MAPHA*. As shown in figure 6, the first DisNAV agent that reaches the goal node, travels a shorter path than the MAPHA* agent. Recall from figure 5 that for a density of 0.3 the best path found by DisNAV is in practice the optimal path. This means that in graphs with low densities it is more beneficial to use DisNAV agents than MAPHA* agents as they both find the optimal path in practice.

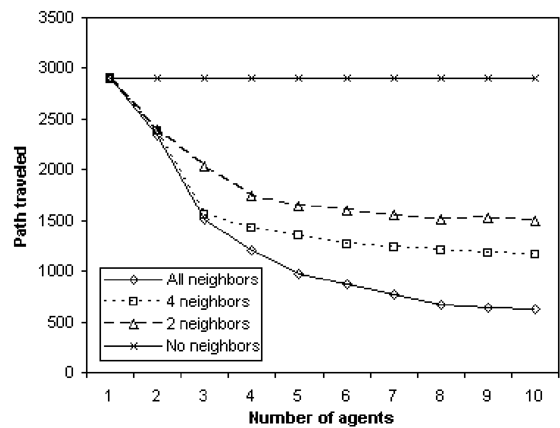

Figure 7: Different number of neighbors (500 nodes)

In another experiment the number of neighbors connected to each agent was varied. Figure 7 shows the length of the path traveled by the first agent reaching the goal node, as a function of the number of agents. The four curves correspond to cases where there are no neighbors, when the number of neighboring agents is 2 and 4 and when all agents

are connected to each other. The results show that adding neighbors improves the total knowledge sharing and reduce the travel effort. We see a diminishing return when adding more neighbors to each agent and the biggest jump is between the case of no neighbors and the case where each agent has two neighboring agents.

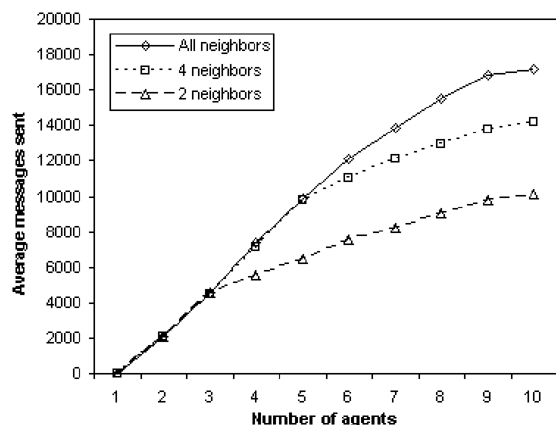

Figure 8: Communication load (500 nodes)

Figure 8 presents the average overall number of messages sent by the agents. The three curves correspond to different number of neighboring agents. It is clear from the results that as the number of neighbors increases, the average number of messages sent by an agent increases as well.

7. DISCUSSION

A multi-agent distributed algorithm for navigation in an unknown physical environment was presented. Our approach assumes that agents have limited communication abilities, and each of them is able to communicate only with a predefined set of agents. In our algorithm each agent is only navigating to the goal and we do not activate a systematic global search to find the optimal path. Former studies (MAPHA* [4]) activate a systematic global search and use complete knowledge sharing between the agents. Therefore, the travel effort and communication overhead of our agents are much smaller than MAPHA*. Even with the above limits, the experiments have demonstrated that our agents, using a simple distributed knowledge sharing, work together very efficiently. The path found by them in practice, is very close (and in many cases equal) to the optimal path. Thus, in practice, unless the optimal path is a hard requirement, our distributed algorithm is preferable. Furthermore, in many cases an agent that arrives at the goal node can prove that the best path known to it is actually the optimal path. This can be done by activating A* from the initial node to the goal. If all nodes generated by A* are included in the *KnownGraph* then the agent can prove that it has the optimal path. Our experimental results show that in small, sparse graphs (e.g., 200 nodes and a density of 0.3) the agents always found the optimal path and in 40% of the cases they could even prove that it is optimal.

Our experiments show that as the number of participating agents grows, the path traveled by the first agent arriving at the goal node is shorter and the best known path is also shorter. This benefit is more pronounced as the graph density decreases. In dense graphs even single agent would travel a relatively short path and will find a path which is a good approximation of the optimal path and adding more agents does not improve the results. However, in sparse graphs, the best path known to a single agent is far from the optimal path and adding more agents improves the results dramatically.

In addition, our experiments with the different neighborhood schemes teach us an important lesson about the influence of the communication constraints on the behavior of the entire multi-agent systems. Indeed, such constraints add strength to the knowledge sharing of the system and the entire multi-agent system performs much better.

Future work will examine how do the communication network topology, the neighborhood radius and the connectivity affect the overall performance. Another interesting question is to examine the cases where agents are starting from different initial nodes and when all agents are required to arrive at the goal node in order to complete the task.

8. REFERENCES

[1] H. Attiya and J. Welch. *Distributed Computing.* McGraw-Hill, 1998.

[2] M. A. Bender, A. Fernandez, D. Ron, A. Sahai, and S. P. Vadhan. The power of a pebble: Exploring and mapping directed graphs. In *Proc. ACM Symposium on the Theory of Computing*, pages 269–278, Dallas, Texas, May 1998.

[3] W. Burgard, M. Moors, C. Stachniss, and F. Schneider. Coordinated multi-robot exploration. *IEEE Transactions on Robotics*, 2005.

[4] A. Felner, R. Stern, A. Ben-Yair, S. Kraus, and N. Netanyahu. PHA*: Finding the shortest path with a* in unknown physical environments. *JAIR*, 21:631–679, 2004.

[5] 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.

[6] S. Koenig and M. Likhachev. D* lite. In *Proc. AAAI-02*, pages 476–483, Edmonton, Canada, July–August 2002.

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

[8] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Series, 1997.

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

[10] G. Solotorevsky, E. Gudes, and A. Meisels. Modeling and solving distributed constraint satisfaction problems (dcsps). In *Constraint Processing-96*, October 1996.

[11] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Data and Knowledge Engineering*, 10(5):673–685, September 1998.

[12] M. Yokoo and K. Hirayama. Algorithms for distributed constraint satisfaction: A review. 3:189–212, 2000.