University of Alberta

Multi-Agent Pathfinding with Direction Maps

by

Maaike Renata Jansen 

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2008

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-47269-9
Our file Notre référence
ISBN: 978-0-494-47269-9

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Pathfinding is a task that is used in many applications from robotics to video games. The single-agent case is well-understood, but the multi-agent case is more difficult. To achieve cooperative behaviour among a group of agents, the agents need to share information with one another. One current approach stores static data about other agents, which is easy to maintain and plan with, but the agents may still collide frequently. Another approach stores dynamic data about other agents, which is complex to plan with, but allows agents to avoid collisions. Instead, we propose the use of a *direction map*, a shared data structure that provides information about how agents have been moving in the world, which is cheaper than planning with fully dynamic information and leads to implicit cooperative behaviour among the agents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Research advancements in areas such as computer hardware and artificial intelligence (AI) technology have contributed to the creation of more and more realistic virtual worlds. This can be used to create immersive environments for users and it gives rise to applications that were not possible in the past. Virtual simulations require that the environment is life-like and immersive, and a large part of this is realistic, intelligent behaviour of the AI characters. There are many aspects to achieving intelligent behaviour in a character. One important component is the ability to plan where the character should move in the virtual world. This problem is called pathfinding or path planning, and it is the subject of this thesis.

Realistic pathfinding is important in many video games, for example. Gamers are always looking for better hardware and software, and they want to see believable behaviour in visually life-like characters. For example, when two AI characters bump into each other in an otherwise empty room, they look unintelligent, which in turn reduces the player's enjoyment of the game.

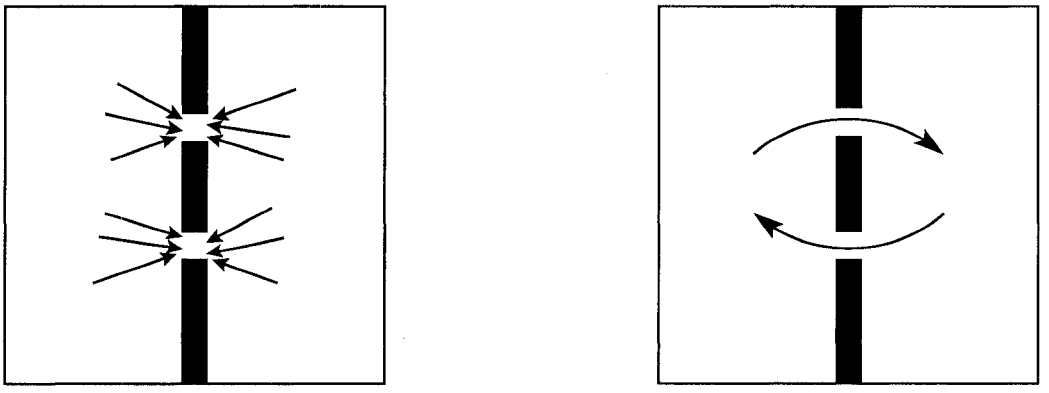
Another example is military simulation systems, such as Virtual Iraq, which was developed at the University of Southern California [19]. The goal of this system is to lessen post-traumatic stress disorder for veterans who have fought in Iraq. This is done by triggering memories of traumatic experiences as part of their therapy, which is a task that is more easily accomplished if the simulation is realistic.

However, path planning is not only used in virtual worlds. It is also utilized extensively in robotics. An example is the Defense Advanced Research Projects Agency (DARPA) Grand Challenge, in which fully autonomous vehicles must navigate an environment [11]. The first two competitions were held in the desert, where the vehicles only needed to be able to follow the road and pass other vehicles, but the 2007 competition was held in an urban environment. In the urban challenge, the vehicles needed to obey traffic laws, navigate intersections, and interact with other vehicles, both human-driven and autonomous. The path planning system is an important part of an autonomous vehicle because it needs to navigate its environment intelligently to reach its destination and avoid collisions.

Another example of path planning in robotics is the use of robots for search and rescue missions, like those used after the World Trade Center disaster on September 11, 2001 [20]. The robots were used for tasks such as searching for victims because the robots are small and can crawl into narrow spaces where humans cannot go. In addition, they can go places that are dangerous for humans, for example because of fire or because a building is in danger of collapsing. Pathfinding is important for search and rescue robots because efficient navigation through the rubble may mean that more lives can be saved.

The problem of pathfinding for a single agent has been well studied. This thesis tackles the more difficult problem of multi-agent pathfinding, in which multiple agents need to navigate the environment simultaneously. This problem is challenging because agents need to take other agents into account in order to avoid collisions and exhibit natural-looking behaviour.

Consider an environment like the one in Figure 1.1 that contains narrow passageways. Assume that there are agents on both sides of the map that need to move through the corridors in the center to the other side of the map. When the number of agents is small, it is relatively easy for the agents to navigate between the two open areas. However, when the number of agents increases, the environment becomes more congested. If the agents do not pay attention to other agents, a situation like the one in Figure 1.1(a) may occur, which does not look intelligent. In this situation, humans are likely to use one of the passageways for left-to-right movement, and the other for right-to-left movement, as shown in Figure 1.1(b). It is desirable that AI characters exhibit behaviour like this as well.



(a) (b)

Figure 1.1: An example of a scenario in which navigation with large numbers of agents is difficult.

When a group of agents travels in the same environment, collision-free, coordinated movement can only occur if the agents have information about other agents. Humans, for example, use visual observations of where other people are and where they are going to avoid bumping into other people. In addition, they use social constructs such as preferring the

right-hand side of a street or sidewalk. Ants, who are nearly blind, avoid imminent collisions by using their antennae to sense other ants that are near, but they also coordinate their movement on a larger scale. Ants leave traces of chemical substances called pheromones behind as a way of sharing information about where they have been with the other ants. For example, ants will lay a pheromone trail between the nest and a food source, and other ants will be attracted to this path and follow it. The use of pheromones also causes them to form lanes of incoming and outgoing ants, which produces efficient behaviour when the ants perform tasks such as gathering food [4].

A number of approaches for the multi-agent path planning problem have been developed. However, the existing methods are sometimes complicated and they do not always lead to realistic-looking behaviour. The agents may collide or take paths that do not look natural.

The method presented in this thesis uses an approach to the multi-agent path planning problem that is different from currently existing methods. Agents are able to mark the map with information, similar to the pheromones left behind by ants. Each agent in the environment marks the states it visits with information about the direction in which it was moving when it passed through this location. We call the data structure that stores this information a *direction map*. Other agents are encouraged to follow the same paths as agents who previously passed by. We will show that this causes the agents to form lanes, which helps them avoid one another as they move through the environment.

Ideas that are similar to direction maps have been used in the video game industry, but they have not been formalized. In addition, the directions are generally hand-drawn rather than learned as they are in our approach, and they are used for greedy one-step movement rather than for planning an entire path.

The remainder of this thesis is organized as follows. Chapter 2 discusses related work. It provides background information on uninformed and heuristic search. This chapter also discusses single- and multi-agent path planning on a map, as well as the use of abstraction to speed up search. The chapter finishes with an overview of navigation and ant-based approaches. Chapter 3 describes the new approach, path planning with direction maps, in detail. It discusses how direction maps are learned and how they are used for planning. Furthermore, this chapter describes some variations on the basic approach. Chapter 4 contains experimental results, and shows that performance is relatively insensitive to variations in the parameters. In addition, the new method is compared to existing multi-agent path planning algorithms, and experimental results for variations on the basic approach that were discussed in Chapter 3 are presented. The last chapter, Chapter 5, consists of conclusions and further directions in which this work can be taken.

Chapter 2

Background

The focus of this thesis is multi-agent pathfinding, which is a special case of the AI field of heuristic search, in which multiple agents perform search at the same time. This chapter will provide background information on both informed and uninformed search algorithms, as well as pathfinding algorithms, abstraction for search, and multi-agent search. It will also discuss ant-inspired algorithms and other non-pathfinding approaches that have been used for agents traversing a map.

In artificial intelligence, an *agent* is defined as any entity that perceives its environment and can perform actions in this environment. A problem-solving agent wants to take actions that move it from its current state to some desirable state, or *goal state* [28]. Examples of problems are solving a puzzle, finding a path between two locations, or moving a robot arm into the correct position to pick up a ball.

Consider an agent who is travelling across Europe. Suppose the agent is currently in Rome and wants to travel to London by air in as few flights as possible. Figure 2.1 shows a simplified map of Europe with a number of European capital cities. A line connecting two cities means that a flight is available between them, and the number by the edge indicates the distance in kilometres.

The agent's objective is to find a series of actions that lead from its current state to its goal state. In this example, the state of the agent is the city it is in, and the actions are flights that take the agent from one city to another. A search problem like this can be represented as a graph, where each node in the graph represents a state, and an edge represents an action that leads the agent from one state to the next. When the agent searches this graph for a solution, or a path between its start and goal states, it generates a *search tree*. This search tree starts with the root node, which is the starting state. Next, the agent will *expand* this node, meaning that it generates the successors of the current state. In the travel example, expanding a node means finding all the cities that are a single flight away from the current city. In the search tree, these become the children of the state that is being expanded. For example, the search tree after expanding Rome in the example above

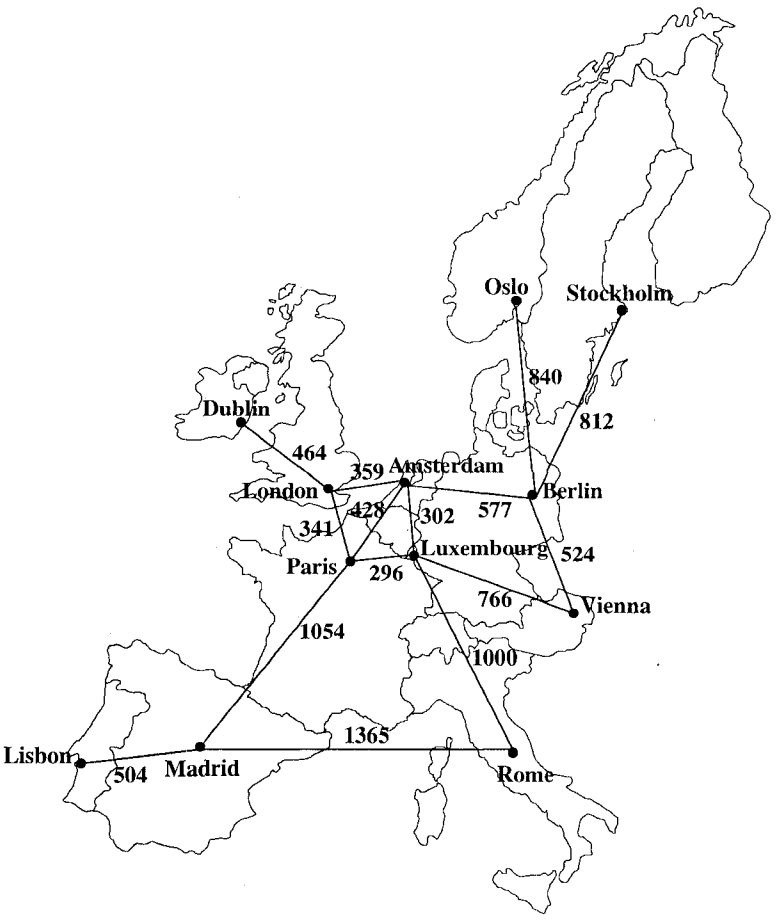


Figure 2.1: A simplified map of Western Europe with distances between cities in kilometres. Map of Europe designed by Brian V. Smith and used with permission. Source for distances: http://www.convertunits.com/distance

is shown in Figure 2.2(a). When an agent searches for a solution, it repeatedly chooses a node to be expanded and checks if it is a goal state. If it is not a goal state, the node is expanded and its successors are added to the search tree. The order in which nodes are expanded is determined by the search algorithm.

One problem that can arise when a search tree is being built is that nodes may appear more than once because there are multiple paths to the same node, *i.e.* if there are cycles in the graph. To avoid expanding the same node twice, a *closed list* is often used. The closed list is a list of nodes that have previously been expanded, and each time a node is generated it is first checked against the nodes already in the closed list. If it has previously been expanded, it is not added to the search tree. This increases the efficiency of the search algorithm and avoids infinite loops.

A search algorithm is said to be *complete* if it will always find a solution if one exists. An algorithm is considered *optimal* if it always finds an optimal solution. In our context,

an optimal solution is a path that has the lowest cost. It is possible that there exists more than one optimal solution.

There are two broad categories of search algorithms. The first is uninformed search, in which the agent only has access to the information given in the problem definition. The second is informed search, where the agent has access to additional problem-specific information, such as a heuristic function, as well. These two types of search algorithms are discussed in the next sections.

2.1 Uninformed Search

In uninformed search, the only information available to the agent is the information that is given in the problem. Formally, the problem is defined as the tuple $\{G = \{V, E\},\ s,\ T\}$, where G is the search graph composed of the set of nodes V and set of edges E, s is the start state, and T is the set of goal states. In other words, the agent only knows the successors of each state, the cost of each action, and whether or not each state is a goal state. Two well-known examples of uninformed search algorithms are *breadth-first search* and *depth-first search*.

2.1.1 Breadth-First Search

In breadth-first search (BFS), the search is done in layers radiating from the start state. First, the immediate neighbours of the start state are expanded, then states that are two steps away from the start state, and so on until the goal state has been found.

In the example, the agent is looking for a path between Rome and London that takes as few flights as possible. Rome will be expanded first, which generates Madrid and Luxembourg (Figure 2.2(a)). Next, the neighbours of Rome are expanded, starting with Madrid. Madrid's successors are Lisbon, Paris, and Rome, but since Rome has already been expanded it is not added to the search tree again. The search tree after this step is shown in Figure 2.2(b). The algorithm then expands the other child of Rome, which is Luxembourg, as shown in Figure 2.2(c). Again, since Rome has already been expanded it is not added to the search tree again. At this point, the immediate successors of the start state have all been expanded, and the algorithm moves to the next depth layer of the tree. Lisbon is the first node at this depth, and expanding it generates only Madrid, which has already been expanded. After Lisbon, Paris is expanded, as shown in Figure 2.2(d), and search continues in this manner until the goal, London, has been found.

The algorithm is called breadth-first search because the algorithm finishes its search of each level of the search tree before it moves to the next level down. BFS is both optimal and complete, and the time and space complexities for BFS are $O(b^d)$, where b is the branching factor, or the average number of successors of a node, and d is the depth needed to be

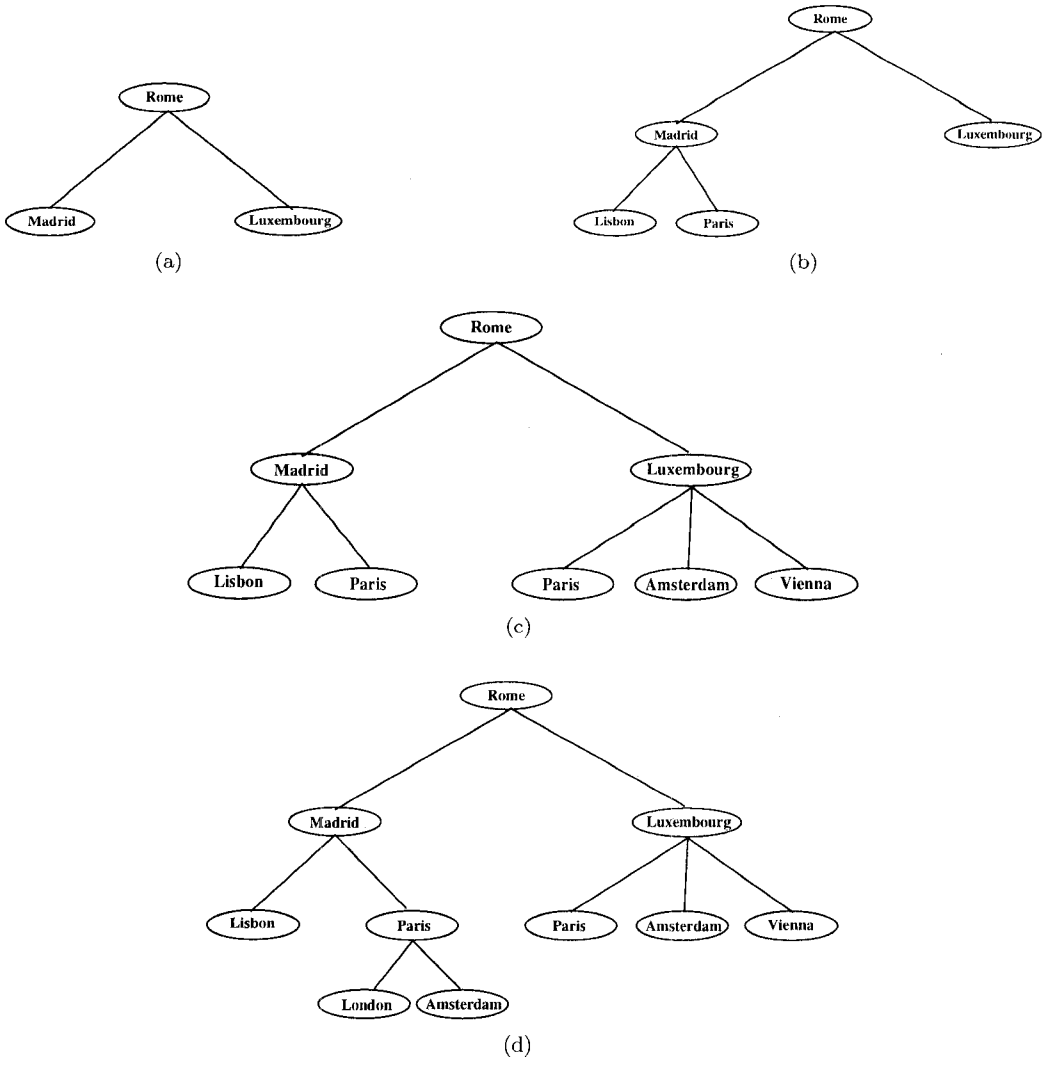


Figure 2.2: A breadth-first search example on the map from Figure 2.1

searched to find the optimal solution [28].

2.1.2 Depth-First Search

In depth-first search (DFS), the agent always expands the generated state that is farthest from the start state next. First, one of the child nodes of the start state is expanded, then one of its children, then a child of this node, and so on.

When there are cycles in the graph, it is possible that DFS gets stuck in an infinite loop. Therefore DFS is often done by searching until some pre-defined cutoff depth d, and then backtracking and expanding a sibling of the last expanded node.

In the travel example, the agent will expand Rome first (Figure 2.3(a)), and Madrid next (Figure 2.3(b)). These first two steps are the same as for breadth-first search. The

next step in DFS is to expand a child of the node that was just expanded. The algorithm expands Lisbon, generating Madrid, but since Madrid has already been expanded, it is not added to the search tree again. Since this branch of the tree starting at Lisbon has been fully explored, the algorithm now expands a sibling of Lisbon, Paris, whose child nodes are shown in Figure 2.3(c). Again, Madrid is not added to the search tree again because it has already been expanded. Next, we are ready to expand London, which is a goal node. The algorithm can either return this path, which is not necessarily optimal, or continue to look for other goal nodes.

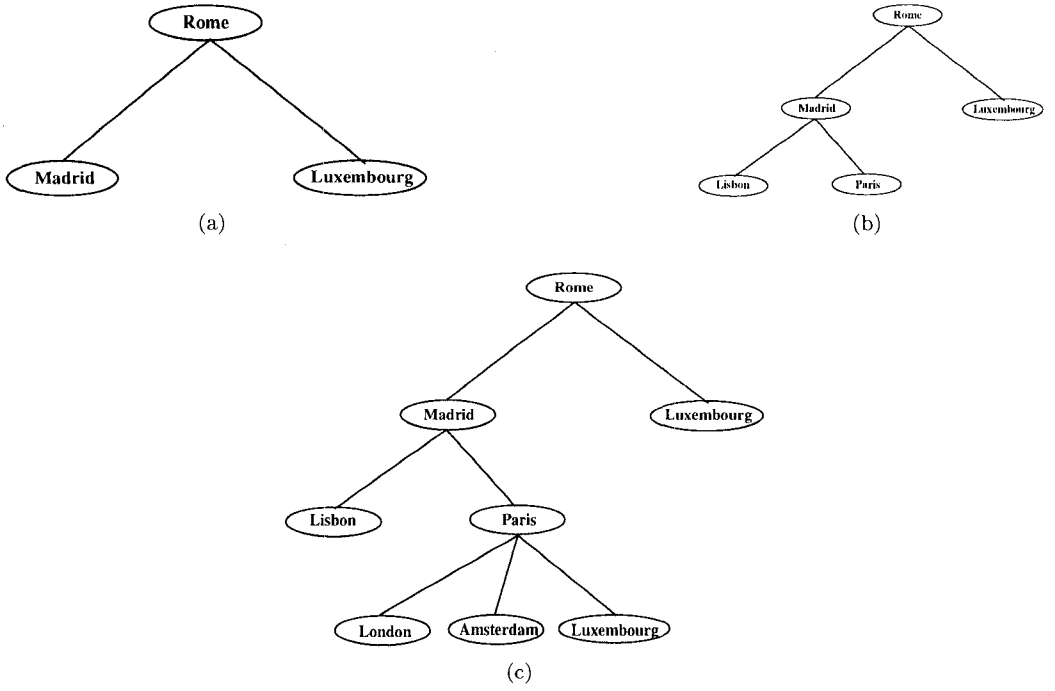


Figure 2.3: A depth-first search example on the map from Figure 2.1

This algorithm is called depth-first search because it always expands the deepest node next. It will fully explore the branches of the tree starting at one child before moving on to another subtree. DFS is not optimal; for example, the agent may find a very deep (high-cost) goal node after expanding the first child of the root node, while a different child of the root node also leads to a goal node, possibly of lesser cost. It is not complete if the graph contains cycles and duplicates are not checked, since the algorithm may get stuck on an infinite branch of the tree. It is also not complete if a cutoff depth d is used, because the goal state may be at a depth greater than d. The time complexity for DFS is $O(b^d)$ and its space complexity is $O(d)$.

Iterative deepenening is a technique that can be used with depth-first search. In iterative deepening, the algorithm first searches to depth 1, then it re-starts and searches to depth 2,

then re-starts again and searches to depth 3, and so on until the goal has been found. Korf has shown that when the ideas of depth-first search are combined with iterative deepening, the algorithm always finds an optimal solution, and has time complexity $O(b^d)$, and space complexity $O(d)$, which makes depth-first iterative deepening asymptotically optimal in time, space, and solution cost [17].

2.2 Informed Search

In an informed search, also called heuristic search, the agent has the same information as for an uninformed search problem, but it is also given a heuristic function $h(n)$, which is an estimate of the cost of a shortest path between any node n and the goal state. A heuristic is said to be *admissible* if it never overestimates the distance to the goal. It is said to be *consistent* if for every state s and every successor s', the estimated cost of the path from s to the goal is never more than the estimated cost of the path from s' to the goal plus the cost of the action between s and s' [28]. In other words, the difference in the heuristic between two adjacent states s and s' can never be more than the cost of the action that takes an agent from s to s'.

An approach to take advantage of this heuristic information is *best-first search*, in which the next state to be expanded is chosen based on an evaluation function $f(n)$, which is an estimate of how promising it is to expand this node. The node with the lowest $f(n)$ will be chosen for expansion at each step.

2.2.1 A* Search

The best known, and most commonly used, heuristic search algorithm is A* [13]. This algorithm expands nodes in order of minimal total estimated solution cost, and maintains generated nodes in an *open list*, a queue in which states are sorted by $f-$cost. Whenever a node is expanded, its children are ignored if they have already been expanded, added to the open list if they are neither in the open or closed list, and updated to reflect the cheapest

	Ams	Ber	Dub	Lis	Lon	Lux	Mad	Osl	Par	Rom	Sto	Vie
Amsterdam	–	577	759	1864	359	302	1482	916	428	1294	1128	936
Berlin	577	–	1320	2315	934	592	1871	840	880	1182	812	524
Dublin	759	1320	–	1640	464	947	1451	1269	779	1887	1633	1686
Lisbon	1864	2315	1640	–	1585	1725	504	2741	1454	1866	2992	2302
London	359	934	464	1585	–	485	1264	1157	341	1434	1436	1238
Luxembourg	302	592	947	1725	485	–	1294	1169	296	1000	1311	766
Madrid	1482	1871	1451	504	1264	1294	–	2391	1054	1365	2597	1812
Oslo	916	840	1269	2741	1157	1169	2391	–	1344	2008	417	1354
Paris	428	880	779	1454	341	296	1054	1344	–	1108	1546	1038
Rome	1294	1182	1887	1866	1434	1000	1365	2008	1108	–	1977	764
Stockholm	1128	812	1633	2992	1436	1311	2597	417	1546	1977	–	1244
Vienna	936	524	1686	2302	1238	766	1812	1354	1038	764	1244	–

Table 2.1: Table of distances between some European capital cities, in kilometres. Source: http://www.convertunits.com/distance

path to this node if they are already in the open list. For a node n, this cost is computed as

$$f(n) = g(n) + h(n),$$

where $g(n)$ is the cost of an optimal path from the start to n, and $h(n)$ is the heuristic cost of the path between n and the goal. Therefore, $f(n)$ is an estimate of the cost of a cheapest path from the start to the goal that passes through n. A* search is optimal if the heuristic is consistent.

A heuristic for the travel example is given in Table 2.1. The heuristic used is the straight-line distance between two cities, which is both admissible and consistent.

Now, we will assume that the agent wants to find a shortest path between Rome and

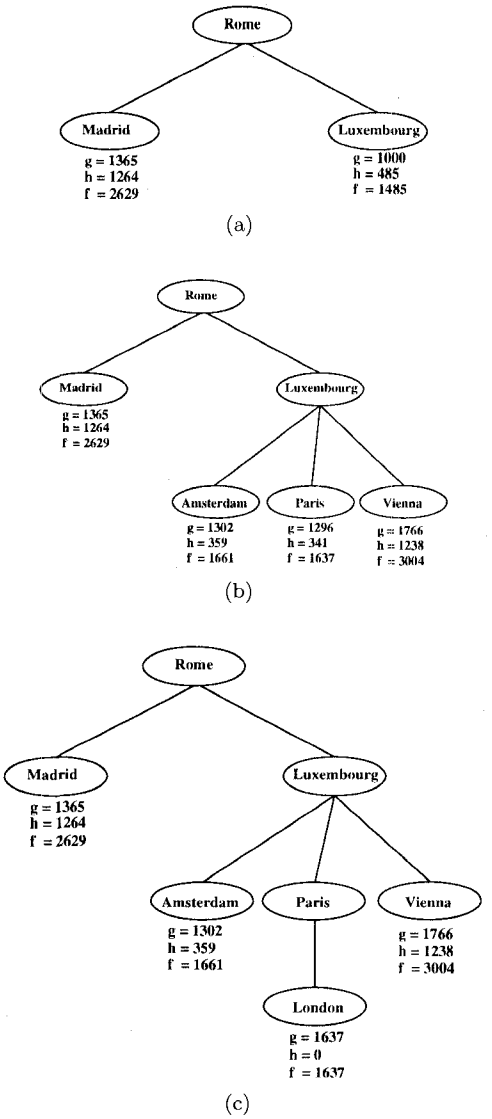


Figure 2.4: An A* search example

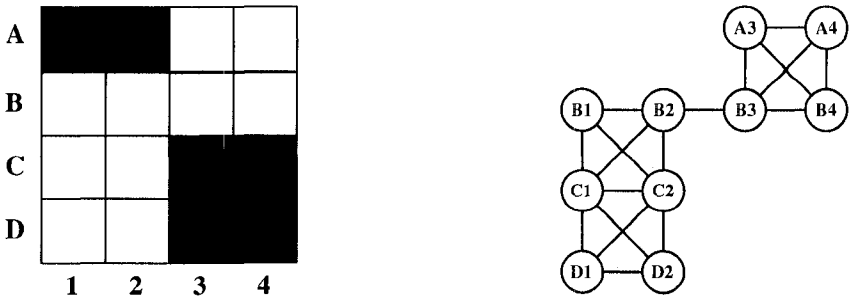
London in terms of kilometres flown rather than in terms of the number of flights required. Actions now have a cost associated with them, equal to the distance in kilometres between the two cities. In the agent's search for a path between Rome and London, Rome is again expanded first. The nodes that are generated are Madrid and Luxembourg. The g-, h- and f-costs for the child nodes are shown in Figure 2.4(a), where the g-cost is the actual distance from Figure 2.1, the $h-$cost is the heuristic distance from Table 2.1, and the $f-$cost is the sum of the $g-$ and $h-$costs. Next, the algorithm expands the node with the lowest f-cost, which is Luxembourg. This node's children are Rome, Paris, Amsterdam and Vienna, but Rome has already been expanded so it is not added to the search tree (Figure 2.4(b)). Now the node with the lowest f-cost is Paris, with $f = 1637$. Its successors are London, Amsterdam, Luxembourg, and Madrid, but Luxembourg has already been expanded, so it is not added to the search tree. Amsterdam was already in the search tree, and the $g-$cost of the path Rome-Luxembourg-Paris-Amsterdam is 1296+428=1724. This is larger than the $g-$cost that was previously stored for Amsterdam, so the values for Amsterdam do not need to be updated. This is also the case for Madrid: the cost of reaching Madrid via Paris is higher than the cost of reaching Madrid directly from Rome, so the values for Madrid do not need to be updated. As a result, only London is added to the search tree (Figure 2.4(c)). London is now the node with the lowest f-cost, and since this is the goal state, the algorithm is finished.

2.2.2 Weighted A*

A variant of the basic A* algorithm is the Heuristic Path Algorithm (HPA), also called weighted A* (WA*), which was proposed by Pohl as a way to find solutions more quickly [23][24]. This algorithm uses an evaluation function in which the $g-$ and $h-$terms are weighted, so $f(n) = (1 - W) \cdot g(n) + W \cdot h(n)$, which can be re-written as $f(n) = g(n) + w \cdot h(n)$, where $W = \frac{w}{1+w}$. WA* will produce optimal paths when $w \leq 1$, but this is not necessarily the case when $w > 1$. Although the solution may not be optimal for values of w larger than 1, it can reduce the amount of work done because the search is more greedy and focused. Davis *et al.* showed that if $w > 1$, the solution is at most a factor w larger than the optimal solution [5].

2.3 Pathfinding on a Map

Pathfinding on a map is a subclass of search problems, in which an agent navigates from its start location to its goal location in some map-based environment. This is used, for example, in video games and in robotics. Examples of map environments are 2- and 3-dimensional grid-based maps and a map in which obstacles are represented by polygons, defined by line segments.



(a) Map (b) Graph

Figure 2.5: A grid-based map and its graph representation

A map can be represented as a graph. In grid-based maps, each grid cell becomes a node in the search graph, and an edge is added between two nodes whenever the agent can move directly from one to the other. A small example is shown in Figure 2.5. In Figure 2.5(a), each location is either passable (white squares) or an obstacle (black squares), and in Figure 2.5(b) circles denote nodes, and lines denote edges in the graph.

Map graphs have some characteristics that make search challenging. One of them is that map graphs often have irregular structures imposed by the obstacles in the map. Because of these irregular structures, there are no symmetries that can be exploited as often seen in some puzzle problems such as the 15-puzzle or Rubik's cube. In addition, the branching factor, *i.e.* the number of actions available to the agent, is at most 8, which is higher than in some other search problems such as sliding tile puzzles. Another challenge is that the environment is potentially dynamic, since new obstacles may appear or existing obstacles may be removed. Lastly, the start and goal locations can be any state in the environment, so we cannot pre-compute start or end moves.

2.4 Abstraction

Heuristic search can be very expensive since in the worst case, A* search needs to expand every state. Therefore, for large state spaces, A* will run out of memory. To reduce the amount of work that needs to be performed, it is possible to use abstraction [15]. During abstraction, the original graph G is represented by a smaller graph A that retains the essential structure of G, removing details and reducing the size of the search space. This process can be repeated by abstracting the the abstract graph A, giving graph A_2, and so on, until the graph is reduced to a single node. The set of smaller and smaller graphs is called an *abstraction hierarchy*. Assuming we choose a good abstraction, *i.e.* one that does not remove the essential characteristics of the map, a path in the abstract graph can be found quickly, since it is smaller than the original graph. This abstract path can be used to guide the search on the original graph.

Formally, an abstraction is a mapping from a graph G to an abstract graph A. The size of the graph is reduced by mapping multiple nodes in G to the same node in A. We say that a node n in G that is mapped to node a in A is a *pre-image* of a, and that a is the *image* of n. Node n is also sometimes called a *child node* of a, and a the *parent node* of n. An edge exists between two nodes a_1 and a_2 in the abstract graph A whenever there exists an action on a child of a_1 that takes the agent to a child of a_2.

The first step to finding a solution in the original graph is to find a path in the abstract graph, for example using A* search. Next, the abstract path $n_1 n_2 ... n_k$ is *refined* by searching for a path that connects the start state, which is a child of n_1, to a child c_2 of n_2, then a path that connects c_2 to a child c_3 of n_3, and so on until the goal has been found. A final solution is obtained by concatenating the sub-paths between abstract nodes, which is not necessarily an optimal solution.

An example of path refinement is shown in Figure 2.6. The squares on the botton show the original states, and the circles at the top of the figure are the abstract nodes. The gray lines indicate which states in the original space have been mapped to each abstract node. The start and goal states are indicated by S and G, respectively. The solid lines between the abstract nodes show the abstract path, and the dashed lines show the refined path. During refinement, we first replace the abstract edge between n_1 and n_2 with a path in the original search space. This is done by finding a path between S and a child state of n_2, in this case c_2. Next, the abstract edge between n_2 and n_3 is refined, by finding a path between c_2 and a child of n_3, in this case c_3. The last step is to find a path from c_3 to the goal state.

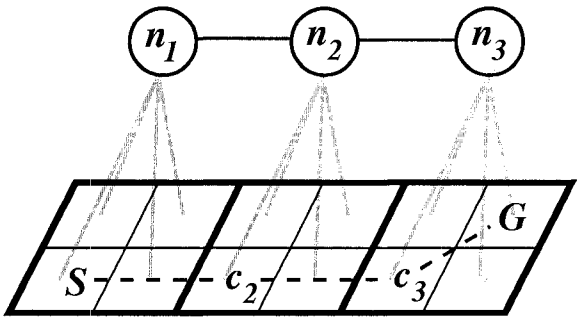


Figure 2.6: A path refinement example

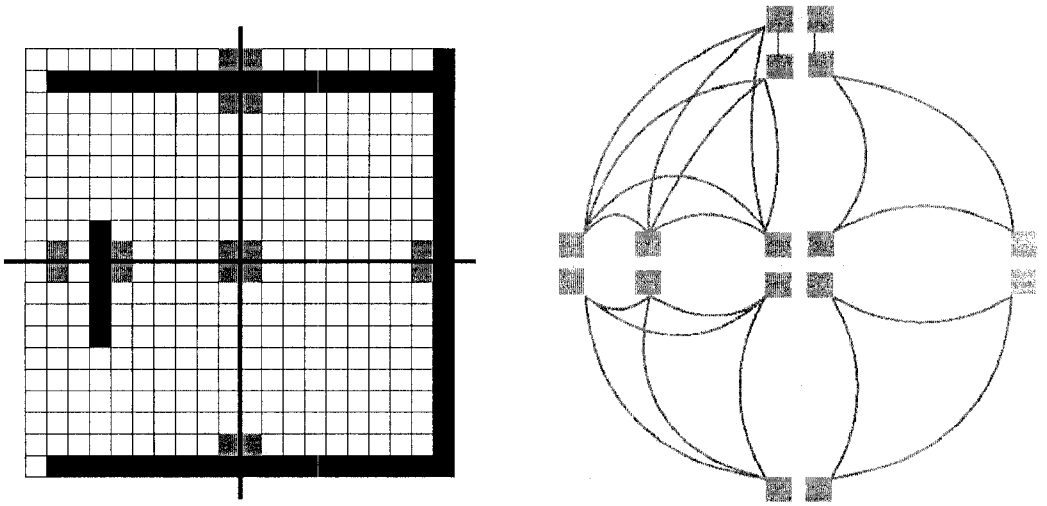
This process can be generalized for multiple abstract graphs. First, search is done in some abstract graph A_k. Next, this path is refined to a path in the abstract graph A_{k-1}, which, in turn, is refined to a path in abstract graph A_{k-2}, and so on until a solution in the original graph G is found. We need to choose the appropriate level of abstraction for the initial abstract search: one that abstracts away enough detail to reduce the search space, but does not abstract away too much because this increases the cost of refinement.

The idea of abstraction has been applied to single-agent pathfinding on a map, for example in the Hierarchical Pathfinding A* (HPA*) algorithm [3]. The abstraction mechanism used by this algorithm does not use a direct mapping from the map graph to the abstract graph, but it uses a similar approach in the sense that it uses a smaller abstract graph to guide search on the map level.

In HPA*, a grid-based two-dimensional map is divided into square sectors, called *clusters*, of a user-defined size. An example map, divided into 4 clusters of size 10x10, is shown in Figure 2.7(a), where the black rectangles represent obstacles and the thick black lines indicate sector boundaries. Along each border between two adjacent sectors, an entrance is defined for each obstacle-free segment along the border. In the abstract graph, each cluster is represented by a group of abstract nodes that represent the entrances to that cluster. For each entrance that is at most 5 grid cells wide, a single abstract node is created in the center of the entrance. For each entrance that is larger than 5 grid cells, two abstract nodes at the ends of the entrance are created. In Figure 2.7(a), the entrance nodes are shown as grey squares. For example, along the bottom border of the top-left cluster, there are two entrances: one between the border of the map and the obstacle, and one between the obstacle and the top-right cluster. The first of these entrances is only three grid cells wide, so it is represented by a single abstract node in each of the clusters that it is adjacent to. The other entrance is wider, and is therefore represented by two abstract nodes in each of the adjacent clusters. Abstract edges are added between corresponding entrance nodes in adjacent clusters (inter-cluster edges) as well as within clusters whenever there exists a path between the two nodes on the map level (intra-cluster edges). The abstract graph for the map is shown in Figure 2.7(b). Inter-cluster edges are the short light gray edges that connect corresponding entrance nodes in adjacent clusters. The intra-cluster edges are shown as darker grey lines.

To speed up refinement, a map-level path is found and stored for each intra-cluster edge e. This path connects the two entrances that are connected by e, and they cannot go outside the cluster boundaries. The cost of each intra-cluster edge, which is used during search in the abstract graph, is set to be equivalent to the cost of the low-level path. Storing these paths simplifies the refinement process because abstract edges can be directly replaced by the paths that have been stored for them.

It is possible to build an abstraction hierarchy by adding multiple abstract graphs. For HPA* this is done by combining multiple clusters at the previous level into a single higher-level cluster. The higher-level abstract nodes are the abstract nodes that lie on the sector boundaries of the previous level. Again, edges are added between corresponding entrance nodes of adjacent sectors as well as within sectors. A path at the previous level is stored for each of the intra-cluster edges, and the cost of the intra-cluster edge is equivalent to the



(a) Map with entrance nodes

(b) Abstract graph

Figure 2.7: An example of the cluster abstraction used by HPA*. Figure adapted from [3]

cost of the lower-level abstract path. In the example from Figure 2.7, a possible additional level of abstraction is to combine the two clusters on the left-hand side of the map into a single cluster, as well as the two on the right-hand side. The abstract nodes at this level would be the ones along the vertical center line.

To find a solution to a search problem, the algorithm first performs an A* search on the abstract path. Next, the algorithm refines the abstract path by replacing each edge by the path that was stored for it, giving either a lower-level abstract path or a map-level path. The last step performed by the HPA* algorithm is path smoothing, which replaces portions of the map-level path by straight lines. This process reduces the length of the path.

Some enhancements to HPA* have been proposed by Jansen and Buro [16]. Their paper introduces a faster way to perform path smoothing, proposes a different algorithm to compute the costs of intra-cluster edges, and suggests computing and storing edge costs on demand rather than pre-computing them.

It is not always desirable to compute a complete path before executing it, either because the agent does not have sufficient time to plan its entire path before it has to make a move, or because the world is dynamic and computing a full path is inefficient because the agent will likely need to re-plan before it reaches its goal. The agent can then compute partial paths instead, and one way to do this is with Path-Refinement A* (PRA*) [31]. This algorithm first builds an abstraction hierarchy of the map, then finds a high-level plan, and refines this plan into low-level actions as needed.

PRA* abstracts cliques, *i.e.* groups of fully-connected nodes, into single abstract nodes. An advantage of this is that it is possible to get from any child node to any other child node of an abstract node in one step, which simplifies the refinement process. The abstract

node's location is set to be the average location of its children. Additional abstract layers can be added using the same process. The abstraction hierarchy is complete when there is a single node for each connected component of the original graph.

An example of this abstraction is shown in Figure 2.8. Figure (a) shows the original graph, with two sample cliques indicated by dotted lines. Figure (b) shows one way in which the map-graph can be abstracted, again with one sample clique shown. Figure (c) shows the second abstract layer of the graph.

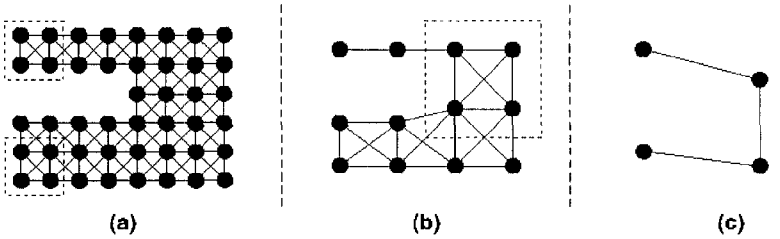


Figure 2.8: An example of a clique abstraction

During path planning, the agent first finds a complete abstract path at some level in the hierarchy using A*. This ensures that the agent does not get trapped in a dead end, because it has some knowledge of the structure of the entire map. Then at each lower level of abstraction, search is restricted to the children of the nodes that make up the abstract path, which reduces the cost of search. Partial refinement is done by only finding a map-level path for only the first k nodes of the abstract path.

Abstraction has not only been used for pathfinding in grid-based world. Triangulation Reduction A* (TRA*) finds paths in environments with polygonal descriptions [6]. It first creates a triangulation of the free space, and then builds a graph from this triangulation. It reduces this graph and performs a search on this abstract representation of the map.

Many other abstractions are possible. An analysis was first done by Holte *et al.* [15], and empirically verified by Sturtevant and Jansen [33], that shows that search effort can be reduced most significantly by maximizing the number of children of each abstract node, and minimizing the maximum length of a shortest path between any pair of the nodes that are abstracted together.

2.5 Multi-Agent Pathfinding

In the simplest form of pathfinding, only one agent needs to plan across the map. However, in many domains there are multiple agents in the environment. Consider a video game in which one of the maps is a market with many people moving around between the different stalls. All these agents are traversing the world and they need to perform pathfinding at the same time.

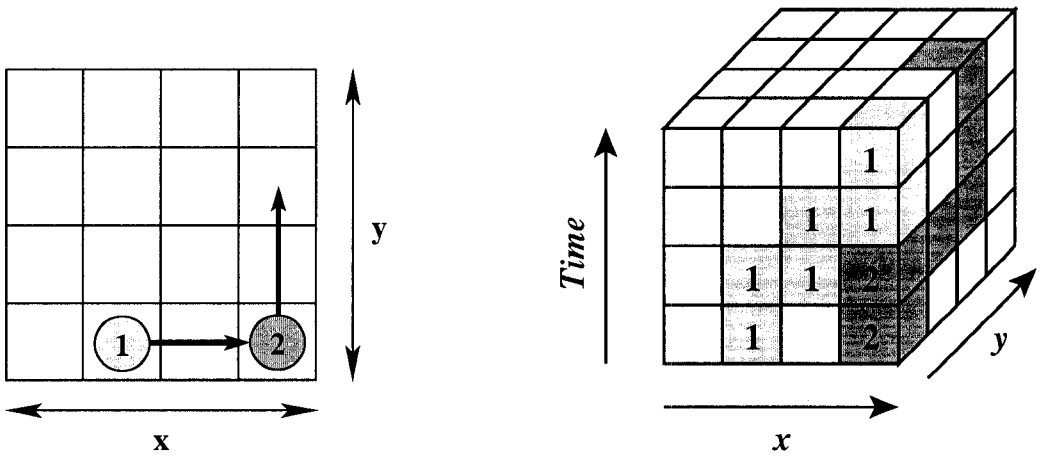
There are different scenarios for the multi-agent case: adversarial (try to stop other agents from reaching their goals), cooperative (agents need to work with other agents to get to their goals), and neither adversarial nor cooperative (the agents may benefit from coordinating with other agents but it is not required for them to reach their goals).

A solution for the multi-agent pathfinding problem that minimizes some criterion, for example the total path length for all agents, can be found by centrally planning for all the agents at the same time, but in general this is infeasible. If there are n agents in the map, and the branching factor (the number of actions each agent can take at any point) is b, there are b^n possible combinations of actions for all the agents at each step. For example, if each agent has eight actions available to it all all times, and there are 5 agents, there are $8^5 = 32768$ possible sets of actions. If there are 20 agents, this blows up to $8^{20} \approx 1 \times 10^{18}$. When this number of combinations is high, search is expensive because whenever a node is expanded, many successors are generated.

If the constraints are relaxed and we do not require a solution that minimizes some criterion like total path cost, search can be done for each agent individually, rather than for all agents simultaneously. One way to do this is by using the A* algorithm to plan for each agent separately, where each agent either ignores all agents except its immediate neighbours, or views them as static obstacles. The agent re-plans whenever it collides with another agent. We will refer to this algorithm as Local-Repair A* (LRA*) [29]. In practice, this approach does not generate believable behaviour because the agents do not take the dynamic aspect of other agents into account and therefore many collisions occur. This is undesirable since creating realistic behaviour is important.

Another approach for multi-agent path planning is to let agents share information about their planned paths with other agents [9][29]. This turns the 2-dimensional search into a 3-dimensional one, where the third dimension is time. The agents reserve their paths in space-time, so that no other agent can plan to be at the same location at the same time. This idea is illustrated in Figure 2.9. Figure 2.9(a) shows a map with two agents and their planned paths. The light grey agent, Agent 1, plans to move to the right, and the dark grey agent, Agent 2, plans to move up. Figure 2.9(b) shows a data structure that could be used for communication between the agents. In this figure the z-axis is time, and agents share where they plan to travel at each point in the future. The bottom layer of the data structure shows where the agents currently are. The next layers show where the agents plan to move at future time steps. For example, Agent 1 plans to move from its current location to the one to the right of it in the next time step, so these two locations are marked off by Agent 1 in the data structure. Agent 1's goal location is Agent 2's start location, but Agent 1 can see in the shared data structure that Agent 2 will move out of the way.

This idea of reservations in a time-space table has been used for managing traffic flow



(a) Map (b) Reservation table

Figure 2.9: Example of a reservation table

through an intersection [9]. The goal of this traffic management work by Dresner and Stone is to plan efficient collision-free paths for cars through intersections. The intersection is split up into grid cells, and the reservation table is maintained by an intersection manager. When a car approaches the intersection, it sends a message to the intersection manager with the expected time it will arrive at the intersection, as well as its expected velocity at the time of arrival and the direction in which it wants to leave the intersection. When this new request comes in, the intersection manager simulates the car's movement through the intersection, and attempts to find a way for this car to make its way through the intersection so that there is no conflict with the reservation table (*i.e.* the new reservation does not occupy any space-time slot that has already been claimed). If such a route is found, it sends a message of approval back to the car, together with any special instructions. Otherwise, a rejection message is sent back and the car will have to slow down and try again later.

This approach works well in the case where lanes have been predefined and only a small portion of the map is shared by agents coming from different directions. For the intersection management problem, where such lanes are defined, this approach is very efficient compared to traffic lights or stop signs.

Reservation tables can also be used together with heuristic search for multi-agent pathfinding on a map where each agent has its own start- and goal locations [29]. Three related algorithms are introduced by Silver, the first of which is Cooperative A* (CA*). CA* is a distributed search algorithm where the individual searches are performed in space-time. After performing this search, the agent marks its path in the reservation table, and other agents cannot plan to be in the same place at the same time. When an agent is blocked because other agents have filled the reservation table entries it needs, the agent has to wait.

The second algorithm improves the performance of CA* by using a more accurate heuris-

tic. Hierarchical Cooperative A* (HCA*) first performs a search in an abstract search space in which the time dimension and other agents are ignored. This search is in the form of Reverse Resumable A* (RRA*), which begins search at the goal node, and searches backwards to the agent's location. The closed list is maintained between searches, since it contains a perfect heuristic for these nodes if time and other agents are ignored, and it can be used for subsequent searches.

A possible drawback of the HCA* algorithm is that planning cannot be performed in real time. However, in some applications, such as video games, real time planning is important and this can be achieved by an algorithm which is able to interleave planning and plan execution. In addition, when an agent uses the HCA* algorithm, it does not exhibit cooperative behaviour once the agent has reached its goal. It is preferable that the agent can move out of the way of other agents even after is has arrived at its destination. Lastly, the previous two algorithms are sensitive to agent ordering, and although it is possible to determine a suitable order for the agents, the algorithm is more robust if the order of the agents is varied. The last algorithm introduced by Silver, Windowed Hierarchical Cooperative A* (WHCA*), addresses these issues. It does this by windowing the search: each agent only searches to some depth, and begins moving. After some fixed time, the search window is shifted and the next portion of the path to the goal is computed. When an agent performs planning, it does a full search on the abstract level, just like the PRA* algorithm from the previous section, to make sure that it is moving in the correct direction. Cooperative search is only done within the search window, and time and other agents are ignored beyond it. This approach allows agents to interleave planning and execution, the agent who has top priority is varied, and agents who have reached their goals can move out of the way of other agents since this windowed search can continue after an agent has reached its destination.

A drawback of reservation-based approaches is that they look for the shortest path possible. Sometimes shorter paths look more chaotic, while a slightly longer path is more visually pleasing.

Spatial abstraction ideas from Section 2.4 can be combined with the ideas from WHCA* [32]. One possibility is to enhance WHCA* by computing the Reverse A* heuristic on an abstract level rather than on the full space. Another possibility is to combine WHCA* with PRA* to form Cooperative Path-Refinement A* (CPRA*) by using WHCA* on the map level of the abstration, rather than A*. These improvements reduce memory and computation overheads.

Biased-Cost Pathfinding (BCP) is another multi-agent pathfinding algorithm, which was proposed by Geramifard *et al.* [12]. This algorithm focuses on reducing the number of collisions between agents. The approach assumes that agents have different priorities, and that lower-priority agents modify their paths when a collisions will occur. The proposed

method is to find the collisions that will occur after the agents have planned their paths. Then for all agents other than the one which has the higest priority, the heuristic for the collision location is increased, forcing the lower-priority agents to re-plan. This approach is repeated until no collisions can be found, or the amount of time allotted for path planning has elapsed.

2.6 Navigation

Many approaches exist for generating realistic navigation behaviour, both for animals and for humans. These are generally not heuristic search approaches, but rather navigation behaviours that may use path-following. Path-following is the task of traversing a given path.

Reynolds developed the flocking approach, a distributed model for simulating the movement of groups of animals that travel together that can be used for path-following [25]. An agent of such a group, which is referred to as a bird-oid, or boid, uses only local information about the rest of the flock to decide on its movement. The behaviour is governed by three desires that are combined to produce flocking behaviour: avoiding collisions with other boids, moving at the same velocity as nearby flockmates, and staying close to nearby boids. Although the idea is fairly simple, it produces realistic behaviour.

When agents navigate their environment individually rather than as a group, steering can be used [26][27]. Reynolds introduces a number of steering behaviours, such as "seek", "pursuit" and "obstacle avoidance". These behaviours produce a vector which represents a force that directs the agent's movement. This steering force is passed to the locomotion controller, which performs the actual movements. One example of a steering behaviour is flow field following, in which the agents follow flow vectors in the environment. These flow vectors are mappings from locations to directions, and the agents follow the directions indicated by the vectors. Path-following is another example of a steering behaviour. Therefore, steering is separate from, but related to, pathfinding, in the sense that a path found by a pathfinding algorithm can be used as a guideline by the steering system.

Force-based approaches, such as potential field methods, have also been used in robotics. An example of this is Arkin's work on robot navigation, which uses potential fields to guide the robots [2]. The potential field consists of vectors which attract the robot towards a goal, and repel it from obstacles. This is combined with high-level behaviours to direct the robot's speed and movement direction.

A similar idea was used for collision avoidance in animation [10]. The system developed by Egbert and Winkler automatically generates repulsive vector fields around objects. Whenever two objects get too close, the repulsive force causes them to move away from one another.

A force model can also be used for pedestrian motion. This can be done by considering social forces, which represent the internal motivations of the pedestrian. Helbing, for example, defined a number of social forces that are combined into such a model [14]. Specifically, the forces that are taken into account by this model are a person's desire to reach his or her goal quickly, a respulsive force exerted on the pedestrian by other pedestrians or obstacles, and attractive forces from certain people or objects. These social forces are quantified, weighted and added to give the pedestrian's "total motivation", which is a vector representing the direction and acceleration for the pedestrian's movement. Experimental results show that pedestrians using this model will form lanes of people moving in the same direction, and that the direction of movement through a narrow doorway alternates. Both these behaviours have been observed in real pedestrians.

Another way to simulate crowds of pedestrians is continuum crowds [34]. In this model, pedestrian movement is viewed as a per-particle energy minimization and it combines global path planning with local collision avoidance. Like real crowds, the simulated pedestrians form lanes of people walking in opposite directions and they can form vortices at crossings.

2.7 Ant-Based Pathfinding

Humans and birds are not the only species that are able to move in their environment in a cooperative manner. In nature, there are examples of communities of insects that exhibit complex group behaviour even though each individual's capabilities are limited. Ants, for example, are almost blind but are able to complete tasks that a single ant would not be able to perform.

In the physical world, ants are faced with patrol tasks when collecting food because the ants move back and forth between the nest and the food source. This task is accomplished efficiently by following pheromone trails left behind by other ants. Research in biology has shown that rules for individual ant behaviour lead to group behaviour in which lanes are formed so that collisions are minimized and traffic flow is maximized [4]. This is similar to what has been observed in pedestrians, as was discussed in Section 2.6.

A term for this phenomenon, stigmergy, was first introduced by the French biologist Pierre Paul Grassé to describe behaviour of termites, but it has later become a term used to describe any emergent behaviour that arises from indirect coordination between agents [7]. By leaving traces in the environment, groups of agents demonstrate intelligent behaviour that the individual agents are incapable of.

These ideas have been used to solve pathfinding-related problems. An example of work which uses this for optimization problems is Ant System [8]. The main idea of this approach is that ants leave behind a pheromone trail as they walk, and other ants will choose the path with the most pheromone with a high probability. After some time, the ants will follow the

shortest path to the goal.

Pheromone-based approaches have also been used in robotics for tasks such as unmanned military aircrafts [21] and to generate complex group behaviours [22].

2.8 Summary

The multi-agent heuristic search algorithms discussed in Section 2.5 do not emphasize natural-looking pathfinding behaviour. LRA* completely ignores the dynamic nature of other agents by viewing them as static obstacles, which leads to collisions. On the other hand, WHCA* stores complete information about other agents' paths, which avoids collisions between agents. However, since this approach attempts to minimize the length of each agent's path, paths sometimes look chaotic.

Instead of storing static data, like LRA*, or fully dynamic data, like WHCA*, we can store static information about the dynamics of the world. For example, we can use information about the direction in which other agents have moved and use this to guide the movement of other agents. This is similar to the pheromones left behind by ants, since in both cases agents use information about other agents' travel when they decide how to move. It is also similar to flow field following, since agents base their movement on directions suggested by the map. The next chapter will introduce a multi-agent heuristic search technique which uses these ideas. We will also discuss how abstraction can be applied to this technique.

Chapter 3

Direction Maps

When multiple agents move around in an environment, they can only exhibit cooperative behaviour if they share information about their movement. Using only a static snapshot of information, such as each agent's current location at the time of planning, is not very useful since these locations change continuously. For example, there may be someone blocking a location ten steps ahead of the agent's current location, but by the time the agent gets there, the other agent has most likely moved. This poses a problem when the agents use Local-Repair A* (LRA*), for instance. Since the agents use static information, they collide frequently.

Agents which use Windowed Hierarchical Cooperative A* (WHCA*), on the other hand, use dynamic information about the world, namely the plans of all other agents. Although agents are able to avoid each other, this approach can have large memory requirements if there are many agents or if the paths are long. Additional drawbacks are that it is expensive to plan with this dynamic data and that behaviour can look chaotic because the agents are trying hard to plan paths that are as short as possible.

Instead of storing static information that reflects the state of the environment at a fixed time t, or storing dynamic information, we can store static information about agent dynamics, such as information about the movement of agents. The method that is proposed here is to perform heuristic search with a *direction map* (DM), by weighting edge costs. For each location in the map, a DM stores a *direction vector* (DV), which represents the expected direction in which an agent will pass through this location.

We visualize direction vectors as arrows. For example, if we let the length be at most one, the x- and y-values range from -1 to 1 and the vectors can be visualized in a unit circle, as illustrated in Figure 3.1. In this figure, only the DVs for the eight movement directions are shown, but an infinite number of DVs is possible. For example, a DV of length 1 that points in the north-east direction would correspond to $\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$.

A small map with example DVs is shown in Figure 3.2. In the figure, black squares are not accesible by the agent. If we assume that the length of all shown DVs is 1 and that they

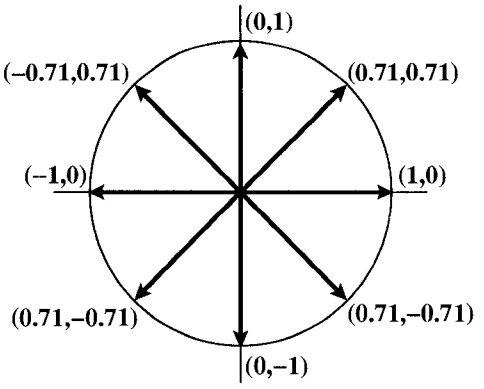


Figure 3.1: A unit circle representation of direction vectors.

always point in one of the eight main directions, the DV in location $B4$ is (1,0), and the DV in location $C2$ is (0,1). A move by an agent can also be represented as a vector, which we call a *movement vector* (MV). A movement vector always has length 1, and points in the direction of the agent's move. For example, if an agent were to move between $A3$ and $A4$ in Figure 3.2, the corresponding MV would be (1,0).

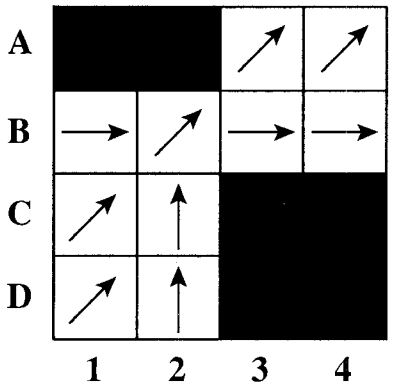


Figure 3.2: An example of a map and its direction map.

The DM is updated on-line. Whenever an agent passes through a location l, the DV stored for l is updated by adjusting the DV a bit towards the agent's MV. During path planning, agents are encouraged to find a path that follows the directions indicated by the DM. This is done by weighting the cost of edges in such a way that a path of length d that follows the DM is cheaper than a path of length d that does not follow the DM. For example, in Figure 3.3, if the agent, indicated by a black circle, were to use A* to find a path between its current location $D2$ and its goal location $A2$, indicated by G, it would plan the path $D2 - C2 - B2 - A2$. However, the direction vectors point in the opposite direction, so when the agent takes the costs induced by the DM into account, it may find that the path $D2 - C3 - B3 - A2$ is cheaper, even though it is slightly longer.

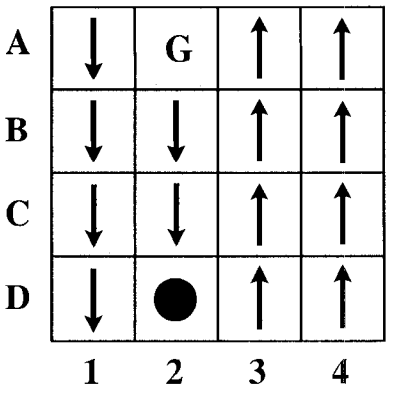


Figure 3.3: An example of how a DM affects path planning.

The next section will describe in detail how DVs are updated. Section 3.2 will discuss how agents use DMs for planning. The last four sections introduce variations of the basic direction map approach, namely increasing the agent's influence on the direction map, local DM approaches, abstraction, and using the DM for greedy search.

3.1 Updating Direction Vectors

A direction map is a collection of DVs; one for each location on the map. Rather than drawing the vectors by hand, as is done for approaches like flow field following (Section 2.6), we want to use the agents' behaviour to build the DM. Building the DM is formulated as a learning problem, where at each location the DM attempts to predict the direction in which the next agent will pass through this location based on the MVs of agents that have already passed through it. Initially, the DVs are set to (0,0), and the DVs are updated every time an agent enters or leaves a location l. One way to do this is by setting the DV for l to be a recency-weighted average of the old DV and the agent's MV, effectively moving the DV partially towards the MV. In particular, if DV_x and DV_y are the x- and y-components of the DV stored at location l and MV_x and MV_y, the $x-$ and y-components of the agent's movement vector, we update the DV for location l as

$$DV_x \leftarrow (1-\alpha) \cdot DV_x + \alpha \cdot MV_x$$

$$DV_y \leftarrow (1-\alpha) \cdot DV_y + \alpha \cdot MV_y,$$

where α is a user-defined learning parameter that determines how much the DV is shifted towards the MV.

Through minor algebraic manipulation it can be shown that there is a theoretical foundation for these simple update rules. They can be obtained by using perceptrons, which are very simple neural networks. We will first describe perceptrons in general, and then we will show that the update rules from above are equivalent to the perceptron update rule.

A perceptron, or single-layer feed-forward neural network, is a very simple artificial neural network that can learn a function [28]. It takes a set of inputs and computes an output. Figure 3.4 shows a diagram of a perceptron. The arrows on the left-hand side denote the inputs and each input a_i is associated with a weight W_i. These weights are learned by the perceptron. Input a_0 is fixed at -1, and W_0 is called a bias weight.

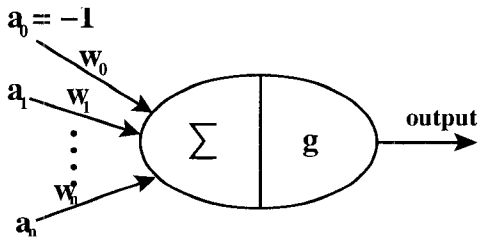


Figure 3.4: A neuron. Figure adapted from [28].

The output of a perceptron, *i.e.* the result of applying the learned function to a set of inputs, is computed as follows. The perceptron computes a linear function of the inputs and weights,

$$in = \sum_{i=0}^{n} W_i \cdot a_i.$$

It is desirable that the perceptron be able to learn non-linear functions, but the above is simply a linear function of the inputs. Therefore, a non-linear function g, called an activation function, is applied to this result, and the output of the perceptron is

$$g(in).$$

Some examples of commonly used activation functions are radial basis functions and sigmoid functions.

Before the perceptron can be used, weights need to be learned. Although this is not the case in our application, in general the weights are learned during a training phase before the perceptron is used to compute outputs. Weights are updated by using a learning algorithm that is provided with a number of training examples, each consisting of a vector of inputs $\mathbf{x}=x_1,...,x_n$ and an output y. The learning algorithm updates the weight for each training example, and then repeats this process for the set of training examples until some stopping condition is satisfied. An example of such a condition is that the total change in the weights does not exceed some threshold.

Pseudocode for the learning algorithm is given in Figure 3.5. Lines 3 through 6 show how the weights are updated based on a training example e. In line 3, the algorithm computes the linear combination of the inputs and their weights for example e. Line 4 computes the difference between $y[e]$, the actual output provided by training example e, and $g(in)$, the perceptron's output for the inputs given by e. This is the error in the output of the

function Perceptron-Learning(α,examples,network) **returns** a set of weights W
input *examples* is a set of examples, each consisting of an input vector $\mathbf{x}=x_1,...,x_n$ and output y
network is a perceptron with weights W_j, j=0,...,n, and activation function g

1 **repeat**
2 **for each** e **in** *examples* **do**
3 $in \leftarrow \sum_{j=0}^{n} W_j x_j[e]$
4 $Err \leftarrow y[e] - g(in)$
5 **for** j **from** 0 **to** n
6 $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
7 **until** some stopping criterion is satisfied
8 **return** W

Figure 3.5: Gradient descent learning algorithm for perceptrons. Adapted from [28]

perceptron. Lines 5 and 6 update the weights in a way that minimizes the sum of squared errors, which is a measure of how well the perceptron's output for each set of inputs matches the actual outputs given by training examples. Each weight W_j is modified by an amount equal to the product of the learning rate α (a parameter to the algorithm), the error in the output, the derivative of the activation function $g'(in)$, and the input x_j this weight is associated with. A derivation of this update rule can be found in [28].

As was mentioned in the beginning of this section, we can use perceptrons to obtain the direction vector update rules. In this formulation, we place two perceptrons at each location in the map: one for the $x-$ and one for the $y-$component of the DV. These are very simple perceptrons, which have no explicit inputs except a bias term, $a_0 = -1$, associated with weight W_0. The output of the perceptron is the $x-$ or $y-$coordinate of the DV, *i.e.* the predicted $x-$ or $y-$coordinate of the MV of an agent passing through the location. Whenever an agent moves into or out of a location, the weights are updated, so there is an implicit input to the perceptron indicating that an agent is passing through the location. This means that the learning algorithm from Figure 3.5 is not performed repeatedly over a set of training examples before the perceptrons are used. Instead, online learning is performed: whenever an agent passes through a location, a training example is generated and lines 3–6 of the algorithm are performed just once for this training example. Learning the DM and using it for path planning are interleaved.

Since there are no explicit inputs to the perceptrons, the training examples that are generated consist only of an output. Since we want the perceptrons to learn a DV, *i.e.* predict the MVs of agents passing through this location, these outputs are the $x-$ and $y-$components of the MV. We will show that if we let the activation function be $g(x) = -x$, the update rules for the DV are equivalent to those for the perceptrons. In this case we can use a linear activation function since the only input to the perceptron is a bias input that is fixed at -1. Since the updates to the $x-$ and $y-$coordinates are similar, we will show the derivation of the DV update rule only for the perceptron which learns the $x-$coordinate for

a location.

When the only input to the perceptron is the bias input $a_0 = -1$ with weight W_0, and the activation function is set to $g(x) = -x$, the output of the perceptron is

$$g\left(\sum_{i=0}^{n} W_i \cdot a_i\right) = g(W_0 \cdot (-1)) = -(W_0 \cdot (-1)) = W_0.$$

Since the perceptron learns DV_x, *i.e.* the output of the perceptron is equal to DV_x, we have that

$$W_0 = DV_x$$

Since there is only one weight, the update rules from lines 5 and 6 of Figure 3.5 can be rewritten as

$$W_0 = W_0 + \alpha \cdot Err \cdot g'(in) \cdot x_0[e].$$

Here, we can replace W_0 by DV_x, and $x_0[e]$, the input from the training example, by -1 since it is the bias input. Since $g(x) = -x$, its derivative is $g'(in) = -1$. Err is simply the difference between the actual output, MV_x, and the output of the perceptron, which is DV_x. Therefore, we have that

$$DV_x = DV_x + \alpha \cdot (MV_x - DV_x) \cdot (-1) \cdot (-1) = DV_x + \alpha \cdot (MV_x - DV_x) = (1-\alpha) \cdot DV_x + \alpha \cdot MV_x,$$

which is the proposed update rule for the DVs.

Now that we have shown that the DV update rules that were given at the beginning of this section are a form of perceptron update rules, we will show an example of how DVs are updated. Consider the map from Figure 3.2 again, let $\alpha = 0.5$ and assume that all shown DVs have length 1. Assume there is an agent in location $B2$ that moves to $B3$, so the MV corresponding to this movement is $(1, 0)$. First, we update the location the agent has just left. The DV for this location was $\left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right)$, so we update the x-component as

$$DV_x = (1 - 0.5) \cdot \frac{1}{\sqrt{2}} + 0.5 \cdot 1 \approx 0.85,$$

and the y-component as

$$DV_y = (1 - 0.5) \cdot \frac{1}{\sqrt{2}} + 0.5 \cdot 0 \approx 0.35,$$

so the DV is approximately $(0.85, 0.35)$.

Next, we update the DV for the location the agent has moved into, $B3$. The DV stored for this location is (1,0), and the DV associated with the incoming direction is $(1, 0)$. The we get

$$DV_x = 0.5 \cdot 1 + (1 - 0.5) \cdot 1 = 1$$

and

$$DV_y = 0.5 \cdot 0 + (1 - 0.5) \cdot 0 = 0$$

so the resulting DV is $(1, 0)$, which is the same as before this update, because the agent moves along the direction of the DV.

3.2 Planning with Direction Maps

Planning is done using A* search, which was introduced in Section 2.2.1. The goal of using a DM during planning is that the agents choose their paths based on the paths that other agents have taken before them. Therefore, they are encouraged to pass through each location in the direction of the DV. This is done by modifying the cost of actions in the environment. Whenever a MV for an edge does not point in the same direction as the DV stored at the begin and end locations of the edge, a penalty is added to the cost of that edge. The size of the penalty depends on how similar the MV for the edge is to the DVs stored for the locations that are adjacent to the edge. The penalty is high when the MV points in the exact opposite direction of the DVs, and low when the MV is only slightly different from the DVs.

In most map-based pathfinding applications, the regular edge cost is equivalent to the length of the edge: 1 for a cardinal edge, and $\sqrt{2}$ for a diagonal one. When a DM is used, the additional cost for traversing an edge is based on the dot products between the edge's MV and the DVs of the adjacent locations. The dot product was chosen because it is a way to measure how similar two vectors are.

Geometrically, the dot product between two vectors a and b is the product of the length of the scalar projection of a onto b, and the length of b [18]. This is illustrated in Figure 3.6. If we let b be the MV, which has fixed length 1, and a be the DV associated with one of the adjacent locations, we can see that the length of the projection depends on two things: the length of vector a and the angle between a and b. The dot product will be negative if the angle between the two vectors is greater than 90°, zero if the vectors are orthogonal, and positive if the angle is less than 90°. The dot product is smaller for shorter DVs, and larger for longer DVs.

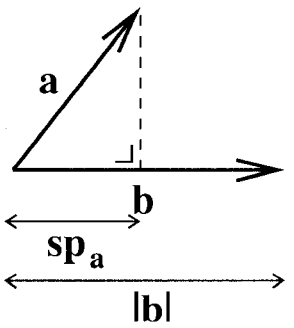


Figure 3.6: Geometrically, the dot product is the product of the length of b, indicated by $|b|$, and the scalar projection of a onto b, indicated by sp_a.

We compute the penalty induced by the DM by computing a weight for each of the two locations adjacent to the edge, say locations a and b, and taking the average. The weight

for location a is a value between 0 and 1. It is computed by first taking the dot product between the DV for a and the agent's MV. Since DVs and MVs can have length at most 1, the dot product will always be a value between -1 and 1. We map these values to 0 and 1 by simple mathematical operations, giving weight:

$$w_a = \left(\frac{1 - (MV_x \cdot DV_x(a)) + (MV_y \cdot DV_y(a))}{2} \right),$$

where $MV_x \cdot DV_x(a) + MV_y \cdot DV_y(a)$ is the dot product between the MV and $DV(a)$. If $w_a = 0$, the DV at a and the MV point in the same direction and are of unit length, and when $w_a = 1$, the vectors are unit length and opposite to one another.

After this weight has been computed for both locations a and b, the edge cost is computed as

$$c_{e,DM} = c_e + w_{max} \cdot \left(\frac{w_a + w_b}{2} \right),$$

where c_e is the unweighted cost of traversing edge e, and w_{max} is a weight parameter, which can be seen as the penalty induced by taking an action with a MV that is opposite to the DVs at both adjacent locations. For example, in Figure 3.2, the penalty for moving from $B4$ to $B3$ is w_{max}, because the MV is exactly opposite the DVs both at $B4$ and at $B3$.

As an example of how a DM is used to compute edge cost, consider the edge between locations $D1$ and $D2$ in Figure 3.2. The DV stored at location $D1$ is $(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$, and the DV stored at $D2$ is $(0, 1)$. The MV associated with traversing the edge from $D1$ to $D2$ is $(1, 0)$. We then compute

$$w_{D1} = \frac{1 - 1 \cdot \frac{1}{\sqrt{2}} + 0 \cdot \frac{1}{\sqrt{2}}}{2} \approx 0.1464$$

$$w_{D2} = \frac{1 - 1 \cdot 0 + 0 \cdot 1}{2} = 0.5.$$

Thus, if we let $w_{max} = 1$, the cost of this edge is

$$c_{e,DM} = RegularEdgeCost + Penalty = 1 + 1 \cdot \left(\frac{0.1464 + 0.5}{2} \right) \approx 1.3232.$$

3.3 Local Direction Maps

The direction map approach has so far been described in a way that requires that all agents have access to a global data structure. However, the approach can be modified so that the agents only use local information. Two possible approaches are described here.

The first approach assumes that there exists a direction map as described above, but the agents can only see the DM within some window of size win. One could imagine that the DM consists of arrows that are drawn on the ground, and that the agents can only see the ground up to some distance, similar to how ants can only sense pheromones within

some distance. During planning, the agents only use the DM for locations close around them, within that window, and ignore it (*i.e.* use regular edge costs) beyond the window. The resulting path may look like the one shown in Figure 3.7. The circle indicates the local radius within which the agent can see the direction map. As a result, the first part of the path may look the way it does in the figure. For the rest of the path, the agent uses unweighted edge costs to plan its path (*i.e.* ignores the DM), so on this simple map, the planned route will be straight lines to the goal. However, the agent does not follow this path all the way to its goal. Instead, it follows this path to the edge of the window, and then re-plans with the window shifted so that it is again centered at the agent's current location. This process is repeated until the agent reaches its goal.

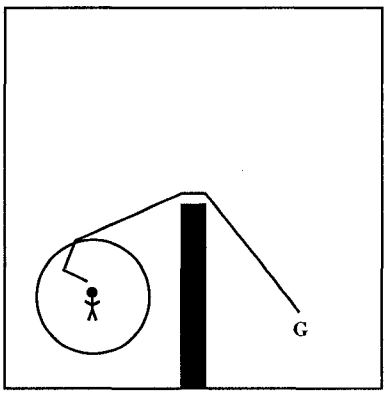


Figure 3.7: An example of using direction maps with local information only.

The second way in which agents can be restricted to use only local information is if each agent keeps its own copy of the direction map. A global DM is stored as well, but agents only have local access to it. The global DM can again be seen as arrows on the ground that an agent can only see within some user-defined radius r_{local}.

Whenever an agent makes a move, it updates the DV for its current location in the global DM in the same way it is done in the basic DM approach as described in Section 3.1. Therefore, the global DM stores up-to-date DVs. However, an agent is only able to access the global DM within radius r_{local} of its current location. After every move, the agent replaces its own copy of the DVs for locations within this local radius with up-to-date DVs from the global DM. Therefore, it has recent information about the DVs for locations near the ones it has recently visited, while the DVs for locations it has not visited for some time may have changed since the agent last updated them. When the agent plans its path, it uses its own copy of the DM, which may not be up-to-date but allows it to make better informed decisions than if it were to ignore the direction map outside its local visibility radius. This way, the agent has some knowledge of the direction map without requiring full access to the global DM.

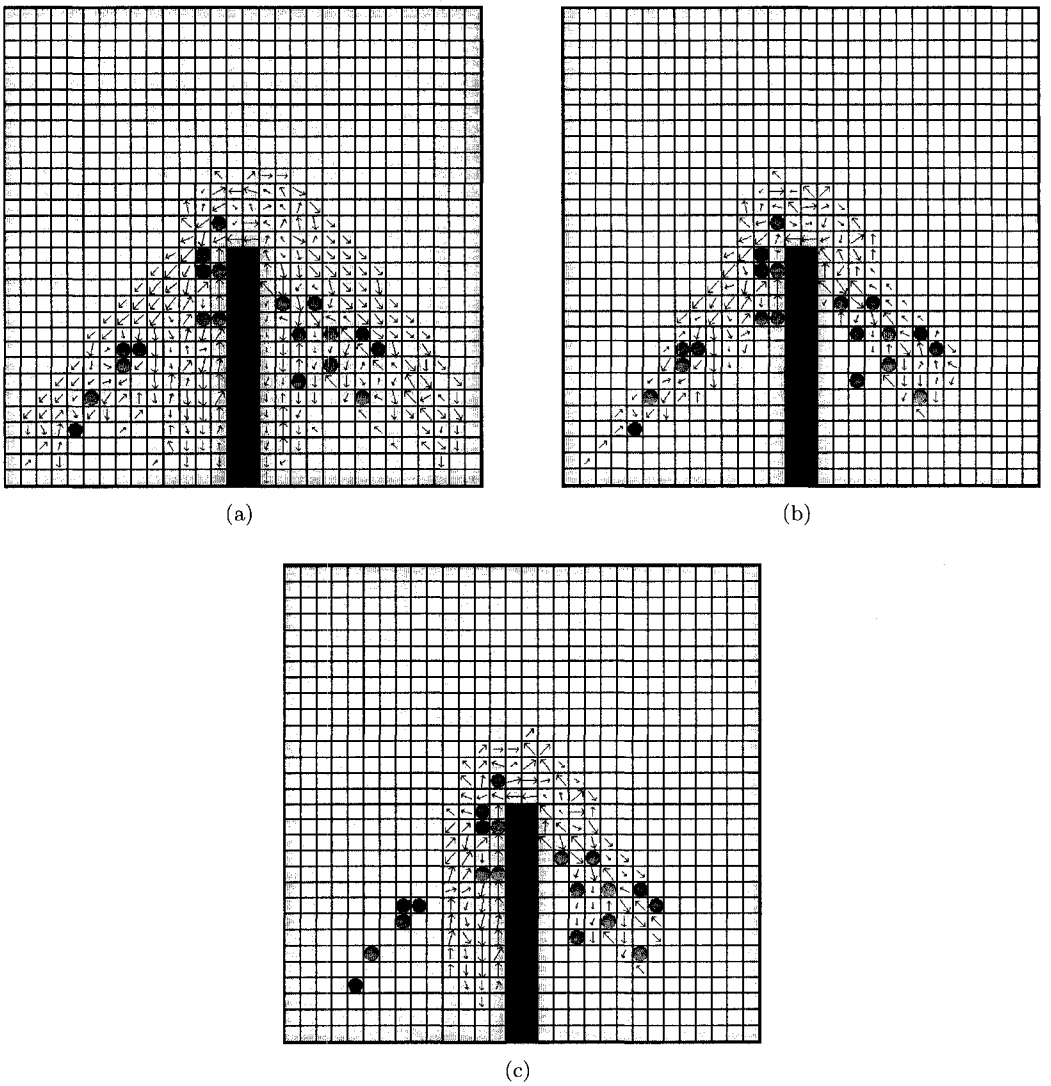


Figure 3.8: Using local maps.

Figure 3.8 shows what the DMs for this method look like in practice. The dots in these figures are the agents. The first figure shows the global direction map, Figure 3.8(b) shows the local copy of the DM for one of the agents, and Figure 3.8(c) shows the DM maintained by a different agent. These last two DMs contain less DVs than the global DM since the agents have only visited a portion of the map.

Advantages of this approach are that it can be used when agents only have local access to the direction map and that it may be cheaper to plan with only a subset of the DVs. A disadvantage is that the cost of storing a DM for each agent may be prohibitively expensive in terms of memory requirements.

3.4 Updating Surrounding Locations

Until now, the agents only modified the direction vectors for the locations they passed through. It may be beneficial to update locations surrounding the ones the agent passes through as well, so that the agent creates a wider directional corridor as it traverses the world.

Whenever an agent makes a move, we update the new location as well as locations within some distance from it. For the agent's new location, we update the DVs using α, just as described above, and for the other locations, we update the DVs with a different learning rate, α_s, which is a parameter to the algorithm. Because we want the agent's movement to have a smaller impact on the DVs of the surrounding locations than on the locations along the path, we choose α_s to be smaller than α. One possibility is to use increasingly smaller α_s for locations that are farther away from the agent's current location.

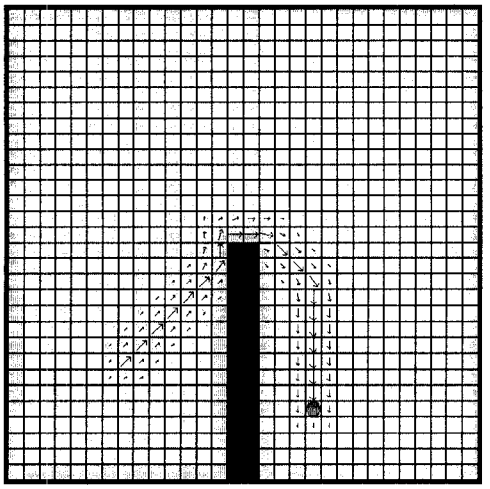


Figure 3.9: An example of updating surrounding locations.

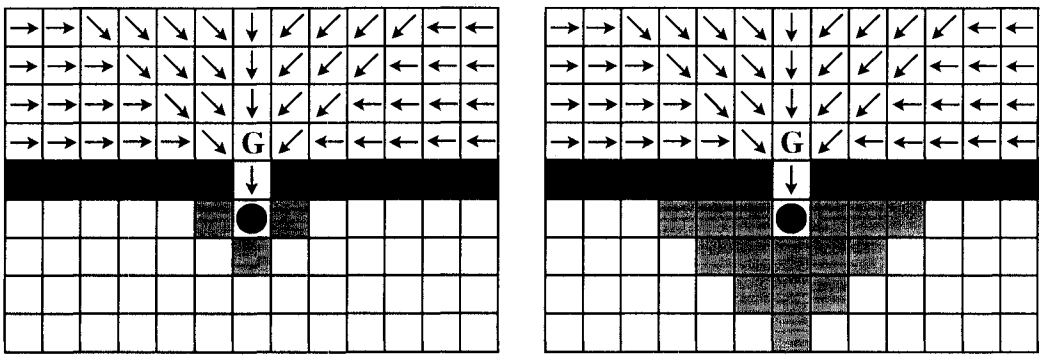
An example of what this looks like for a single agent is shown in Figure 3.9. Here, the agent updates locations it passes through as well as the eight locations directly surrounding it. The path the agent has taken is marked by longer DVs than the locations surrounding the path, indicating that these DVs are learned using a larger learning parameter α, but the surrounding locations are also updated, resulting in a corridor of shorter DVs that surrounds the agent's path, reflecting that these have been learned using a smaller learning parameter α_s.

3.5 Abstraction

Using direction maps is more expensive than simply using Local-Repair A*. One reason for this is that the heuristic only takes the distance to the goal into account, and not the

penalties induced by the DM. Therefore, it may lead the agent astray. Another reason is that the agent may need to plan around expensive edges, which increases the number of node expansions.

This is illustrated in Figure 3.10. The agent, indicated by a circle, is planning a path to its goal, indicated by 'G'. There is only one way to get from the agent's current location to the goal, but the DVs, indicated by arrows, are facing opposite to the direction in which the agent needs to travel to get to its goal. Assume that the cost of moving to any node in the bottom half of the map is just the edge cost. Figure 3.10(a) shows the nodes that will be expanded by the agent before it expands the corridor node if $w_{max} = 1$. All other immediate neighbours will be considered before the corridor node because it is twice as expensive to travel to the corridor node as it is to travel to any of the other neighbours. Figure 3.10(b) shows which nodes will be considered before the corridor node when $w_{max} = 3$. The agent will search more nodes in the bottom half of the map first. Therefore, the number of nodes expanded increases as w_{max} increases.

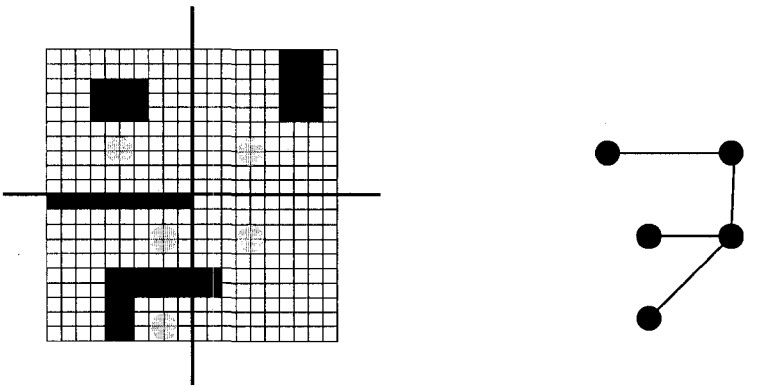


(a) $w_{max} = 1$ (b) $w_{max} = 3$

Figure 3.10: The number of nodes expanded increases as w_{max} increases.

As discussed in Section 2.4, abstraction is an approach that has been succesfully used to reduce the amount of work needed to perform pathfinding, while generating paths of good quality.

When abstraction is used, the map is simplified to reduce the size of the search space. A rough plan is found in the abstract graph, which is then used as a guideline for the low-level path. These ideas can be combined with direction maps in many ways. One approach is to divide the map into square sectors, similar to the way presented by Sturtevant [30]. In each sector, an abstract node is created for each connected component within that sector. An abstract edge is added between abstract nodes A and B whenever there exists an edge in the map graph between some child of A and some child of B. Thus, the abstract graph maintains the topology of the underlying map. An example is given in Figure 3.11. Figure 3.11(a) shows the underlying map as well as sector boundaries and abstract nodes. Figure 3.11(b)



(a) (b)

Figure 3.11: Abstraction example.

shows the abstract graph, including edges.

Paths are planned by first performing an A* search in the abstract graph, without using a DM. Next, this path is refined on the map level, by finding a path between the current location and any child of the next abstract node. In this step, the DM is used to guide the search. In the approaches from Section 2.4, the agent always finds a path to a specific child of the abstract node, but when DMs are used it may be beneficial to search to any child of the abstract nodes, except for the last one. This way, the agent can follow the DM more closely. In addition, these approaches restrict refinement to the children of the abstract nodes, but when the agent takes the DM into account it may be cheaper to follow a path outside of the sectors defined by the abstract path. Therefore, we allow the agent to find a path that is not restricted to these sectors. This is illustrated in Figure 3.12. The agent plans an abstract path through the three sectors marked by thick black lines, but the map-level path, indicated by a dashed line, does not lie fully within these sectors.

After finding an abstract path, the agent can refine the entire path before it starts moving. However, one of the advantages of using abstraction is that the agent can quickly find a high-level path and only do partial refinement, as was done with PRA*, for example [31]. Rather than refining the entire abstract path, it only refines the first part of it, and replans when it gets to the end of this partial path. In our implementation, the agent plans a path to the second next abstract node, and it then cuts this path off after some user-defined proportion.

The amount of work done is reduced because we use the map-level graph and the DM to compute a series of short paths between sectors rather than one long path. In addition, this approach is suitable for real-time search since partial path planning can be done similar to the way it was done in PRA* [31].

It is possible that the agents get deadlocked. This may happen, for example, if two

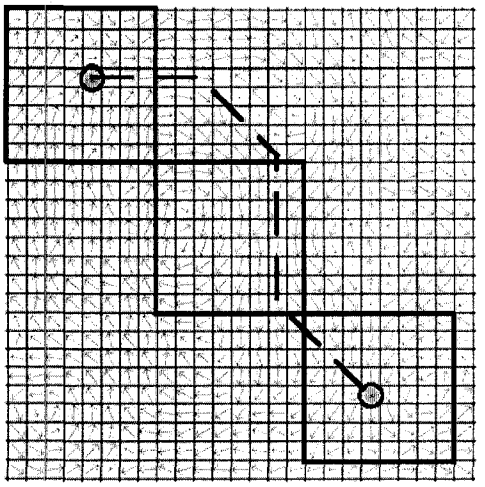


Figure 3.12: An example of path planning with abstraction.

agents are standing at the edge of two adjacent abstract sectors, as in Figure 3.13. The squares indicate different sectors and the gray circles are abstract nodes. The white agent has planned the abstract path A-B-C-D, while the black agent has planned D-C-B-A. The black agent's current abstract goal is C, while the white agent's current abstract goal is B. On the map level, each agent's goal is to get to any child node of its abstract goal. Now imagine that the agents have planned their paths and collide in the location where they are in the figure. Since the white agent's current location is the closest child node of the black agent's abstract goal, and vice versa, both agents are waiting for the other agent to move off its goal and the agents are deadlocked. This situation can be resolved by letting the agents skip the next abstract location and path to the next abstract node instead.

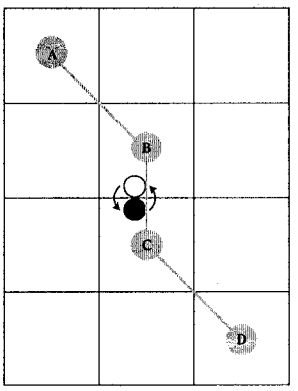


Figure 3.13: A possible deadlock scenario.

3.6 Using Direction Maps for Greedy Search

In Section 2.6 we discussed steering. One form of steering is flow field following, in which agents greedily follow the direction indicated by a flow field. It uses a hand-drawn map which looks similar to a DM. In fact, we can use an established DM similar to the way the flow fields are used.

To do this, we first have to build a DM. This can be done, for example, by letting a number of agents perform pathfinding on a map. Next, we remove those search agents and place greedy agents on the map. A one-step greedy agent will always take the cheapest action. When a direction is stored at its current location, it will find the cost of each adjacent edge and choose the cheapest one. It will not consider occupied locations and it will prefer a location with a DV associated with it to one that does not. If the top two choices are very similar in cost, it will take the second best choice with some probablility, 0.25 in our implementation. If no direction vector is stored at the agent's current location, it will take a random action. The agent only expands a single node at each time step, so this is a cheap way to navigate the world.

An example is shown in Figure 3.14. The direction map was generated by letting 40 agents patrol back and forth 10 times. The DVs surrounding the agent's path are updated as well as the ones on the path. After those ten patrols, the DM agents were removed, and a greedy agent was added in a random location. If the agent is placed on a grid cell where no DV is stored, the agent makes random moves until it encounters the DM. The thick line indicates the path the greedy agent follows once it is on the DM.

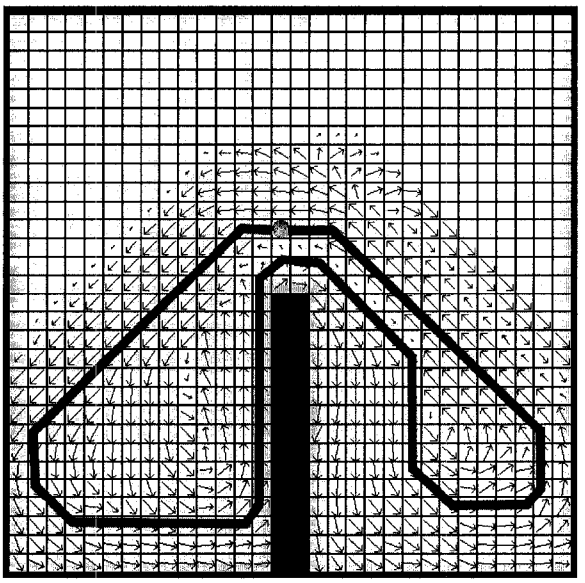


Figure 3.14: Using a DM for behaviour that is similar to steering

Chapter 4

Experimental Results

In Chapter 3, we introduced heuristic search with direction maps as an alternative to the existing multi-agent path planning algorithms which were discussed in Section 2.5. In this chapter, the different direction map approaches from Chapter 3 will be compared, both to each other and to previously developed methods. This chapter contains a representative subset of the full set of experiments that were run.

The experiments were conducted in the Hierarchical Open Graph (HOG) framework [1]. For each experiment, a number of agents were placed on a map and asked to perform a pathfinding task. The task that is used here is a patrolling task, in which each agent must move back and forth between two locations a user-defined number of times. This is a task that is common in, for example, real-time strategy games, where characters collect resources by walking back and forth between the resource and their home base. Performing this task well allows the agents to collect resources more efficiently. If we extend this idea to multiple patrol locations, this could be used by robots in an office that deliver mail or coffee to a number of different offices.

The maps used here are 2-dimensional grids with eight directions of movement: four

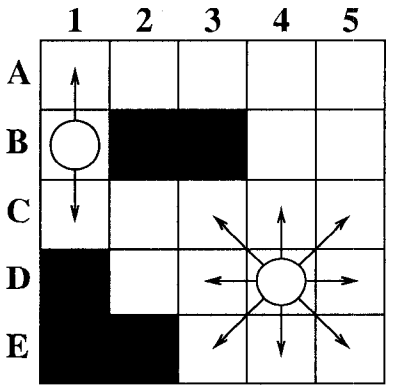


Figure 4.1: An example showing valid moves for agents in different locations

cardinal directions, and four diagonal ones. This is shown for the agent in cell D4 of Figure 4.1, where the arrows indicate valid actions. Each grid cell is either passable or blocked, and agents cannot move diagonally between two passable locations if a location which is adjacent to both locations is blocked. For example, in Figure 4.1, the agent at location B1 cannot make a diagonal move to C2.

A number of different maps were used for the experiments. Experiments were performed on maps of sizes 32x32 and 64x64. The larger maps allowed for experiments with more agents, and therefore they give more meaningful results in terms of how well-coordinated the agents' movement is. Unless otherwise noted, the results presented in this chapter were obtained on the larger maps, shown in Figure 4.2. On the empty map in Figure 4.2(a), each agent's patrol locations are chosen randomly from across the map. On maps (b), (c), and (d), each agent has one patrol location on the right-hand side of the map and one on the left-hand side, and they are restricted to the locations shown in dark grey. Maps (b) and (c) were chosen to evaluate the performance when the heuristic is less accurate. In this case, the inaccuracy of the heuristic is due to the fact that it leads the agents through the barriers

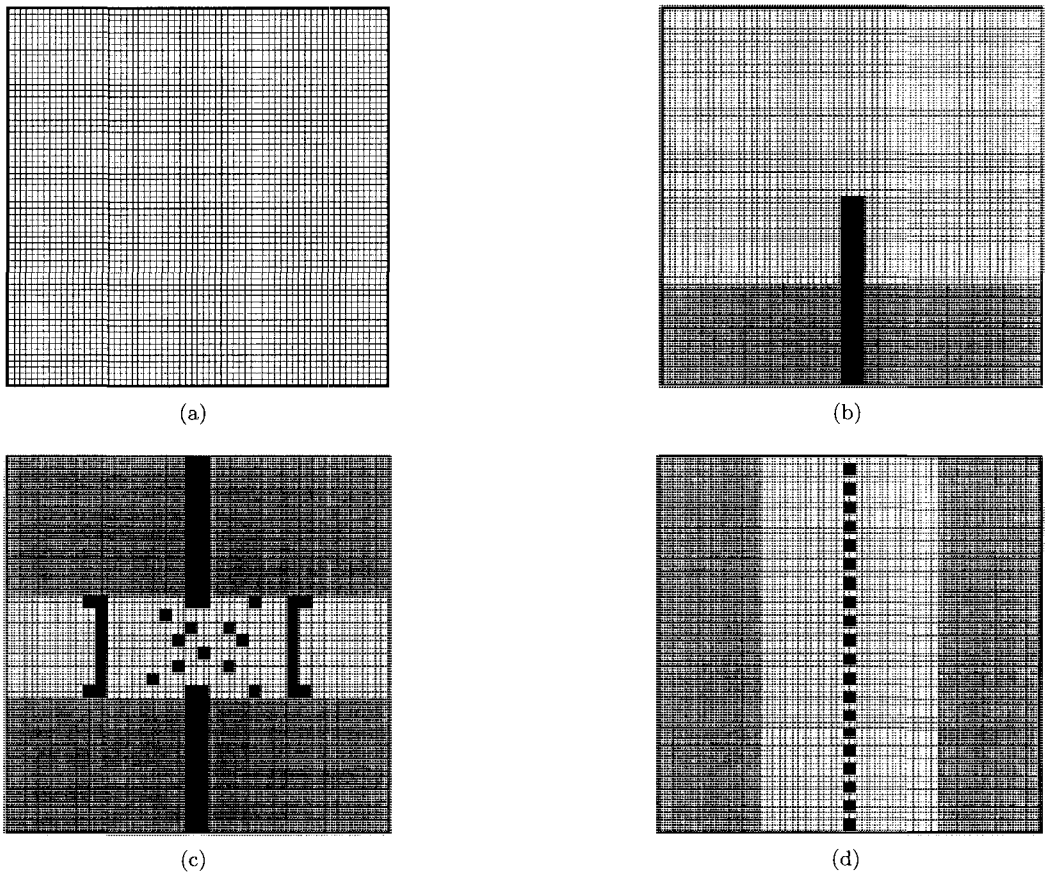


Figure 4.2: Maps used for experiments

in the center of the map. In addition, Map (c) allows us to evaluate how well the direction map approach performs when the map contains a more contrained area with many obstacles (*i.e.* the center of the map). Map (d) was used to determine how well pathfinding with direction maps performs when the agents need to share small passageways, and whether the agents are able to avoid congestion at these passageways. The empty map was used to evaluate the performance when agents' patrol locations are not limited to certain portions of the map. Due to time constraints, experiments were not performed on actual game maps.

Initially, patrol locations were chosen at random, but in that case it is possible that an agent cannot reach its goal. This happens when the goal is surrounded by other agents' goals, and the other agents have already reached their goals, effectively blocking every possible path to the goal. Therefore, we have added the restriction that the patrol locations lie on a checkerboard pattern, *i.e.* that the sum of the $x-$ and $y-$coordinates of any start or goal location is always even.

Most reported results show an average over 50 different runs, where each run consists of a different randomly selected set of patrol locations, but some data was obtained by averaging results of 1000 different runs in order to assert the statistical significance of the data.

For all experiments, the agents move at the same speed, namely one unit of distance per unit of time. Although the simulation time is increased in increments, agents' movement is maintained in real-time. Therefore, when an agent makes a move, it will move on the first time step after the time required to make this move has elapsed. Agents are not slowed down because of time spent thinking (planning), and thinking time is not included in the reported simulation times. The order in which agents plan is determined by the order in which they are placed on the map before the experiment begins, and it does not change. For all experiments in this chapter, the agents perform 20 patrol loops, where for each patrol loop an agent moves from its start location (*i.e.* its first patrol location) to its second patrol location, and back to the start location. The experiment ends when the last agent has finished its last patrol loop.

For any search done on the map graph, with or without a DM, the octile heuristic is used. The *octile distance*, h_o, between two locations $l1$ and $l2$ is the length of a shortest path between the two locations on a empty 8-connected map. Formally,

$$h_o(l1, l2) = \sqrt{2} \cdot min(|l1_x - l2_x|, |l1_y - l2_y|) + ||l1_x - l2_x| - |l1_y - l2_y||,$$

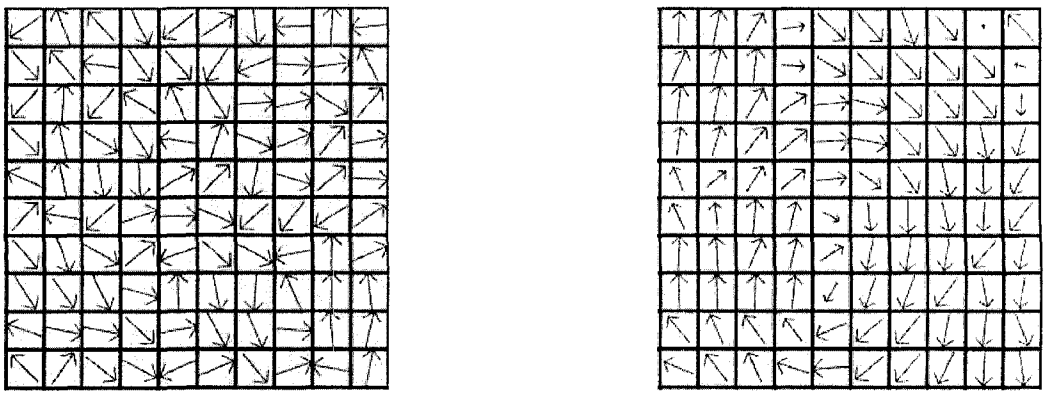
where the $x-$ and $y-$ subscripts indicate the $x-$ and $y-$coordinates of a location, respectively. This heuristic is similar to the Manhattan distance, or l_1 norm, except that has been adapted to include diagonal moves as well as cardinal moves.

In the case of abstraction, straight-line distance between two abstract node locations is used as a heuristic in the abstract level.

The *visibility radius* is one of the parameters for the A* and Weighted A* algorithms. It specifies the distance within which an agent can see other agents when it plans.

We evaluate performance based on a number of metrics: 1) the average number of nodes expanded by the search algorithm per agent per loop, which is an indication of the amount of work done, 2) total simulation time, which is the time that elapses between when the first agent starts moving and when the last agent finishes its simulation, 3) the average distance an agent travels in one patrol and 4) the average number of failed moves, or the number of collisions, per agent, per loop.

One of the goals of direction maps is to create believable behaviour. Although in some applications this is not important, this is desirable in certain types of application where human-like behaviour is required. An example of this is simulations or crowds of non-player characters in video games with realistic graphics. The metrics discussed in the previous paragraph do not give an indication of how well an algorithm performs in this sense. The visual fidelity of the simulations is difficult to quantify, but we attempt to do this with a new metric, which we call *map coherence*. This is meant as a way to express how uniformly agents move in the map. To illustrate this, Figure 4.3 shows two DMs. The left figure shows an example of a map in which the arrows do not follow each other coherently, *i.e.* we cannot trace a clear path through the map by following the DVs. The map coherence in this case would be low. The direction map on the right, on the other hand, shows very distinct paths and clear flow. Therefore, it has higher coherence.



(a) Low coherence

(b) High coherence

Figure 4.3: Illustration of map coherence

Map coherence is computed as follows: for each location v_1 in the map where a DV is stored, we find the movement vector that is closest to the DV of v_1, *i.e.* the one that make the smallest angle with it. We then find the adjacent location v_2 that lies in the direction of this movement vector. We create a new vector by taking the average x- and y-coordinates of the DVs for v_1 and v_2 and compute its magnitude. The map coherence is the average of

these values over all locations where a DV is stored.

Since the length of the DVs can be at most 1, map coherence lies between 0 and 1. Very low coherence is not found in practice because as an agent moves through the world, the MVs that make up its path are often coherent. During our experiments, the coherence was never lower than 0.25 and never higher than 0.95. Therefore, the scale on coherence graphs will range from 0.25 to 0.95.

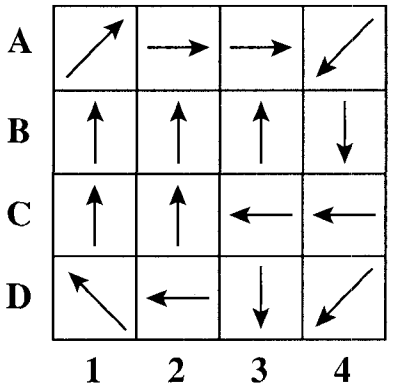


Figure 4.4: Map coherence example

Consider the map in Figure 4.4 for an example of how coherence is computed. For simplicity, we assume that all DVs are of unit length. First, we look at the DV for location C2 as an example. This DV is $(0, 1)$ and it points towards B2, which has DV $(0, 1)$ associated with it. Averaging the $x-$ and $y-$coordinates of these two DVs gives $(0, 1)$, which has magnitude 1. Therefore, the magnitude term for location C2 is 1. Since these two DVs point in the same direction this term is high.

As another example, we look at location C3, with DV $(-1, 0)$. Its DV points to location C2, which has the DV $(0, 1)$ associated with it. Averaging the $x-$ and y-components gives the DV $(-0.5, 0.5)$, which has magnitude $\frac{1}{\sqrt{2}} \approx 0.71$.

To find the map coherence, we would do this for all locations and compute the average value.

4.1 Direction Maps

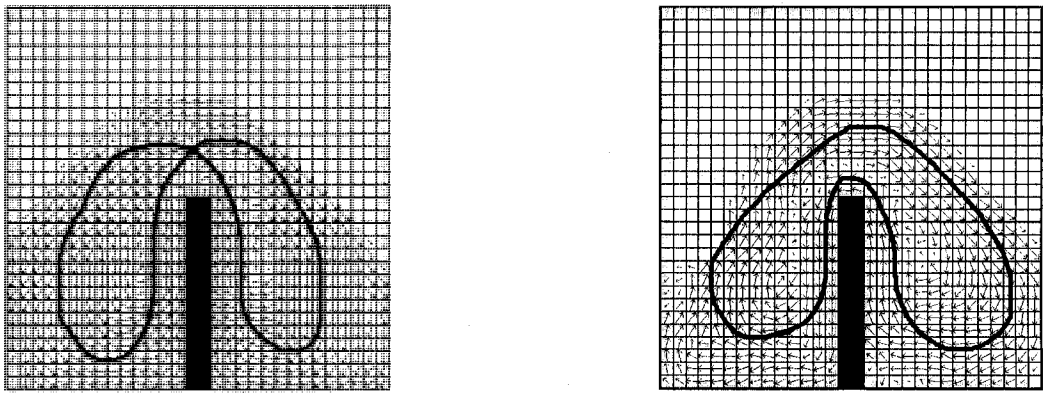
We will first evaluate the pathfinding performance when direction maps are used. Some general observations about the behaviour of direction map-based path planning will be discussed first, followed by an overview of how the parameters affect the performance of DMs.

4.1.1 General Observations

Although the behaviour of the agents is chaotic at first, lanes are quickly formed as the direction map is updated. In map (c) in Figure 4.2, for example, each of the narrow corridors in the center is soon designated as a left-to-right passageway or a right-to-left passageway. This ensures that the agents can move quickly and collision-free from one side of the map to the other.

The lanes that are established are not always the most efficient routes because the DM sometimes initially gets set this way. This happens, for example, when agents who initially defined the lane had to move out of the way of other agents. Although the direction map continues to change throughout the experiment, the agents sometimes choose a path and follow it for the remainder of the simulation.

Another observation is that a different set of start and goal locations can give rise to different lanes being formed even on the same map. For example, on map (b) in Figure 4.2, the agents sometimes cross over above the obstacle in the middle of the map, as illustrated for a smaller map in Figure 4.5(a), and sometimes they do not, as in Figure 4.5(b). When the agents cross over, a bottleneck is created, but since the agents form lanes, the behaviour is more coherent than when no DM is used.



(a) (b)

Figure 4.5: Different lanes can be formed on a map

We measured the number of nodes expanded, distance, and number of collisions for each patrol loop individually. The first loop often gives poor performance – both for direction maps and for previously existing approaches – because the agents are likely to collide in the middle of the map, especially on maps (b), (c), and (d) in Figure 4.2. The last loop of a simulation often shows optimistic numbers because once some of the agents finish their simulations, the task is easier for the remaining agents. Therefore, the tables below will report averages that do not include the first and last patrol loop.

4.1.2 Parameter Variation

The basic direction map approach requires the setting of some parameters. The first set of experiments will analyze how the performance of the algorithm changes as these parameters change.

First, we will vary w_{max}, the penalty added to the edge cost if the direction vector is opposite to the movement vector. The data in Table 4.1 shows the performance of the DM as w_{max} changes between 2 and 20. The data reported here is for map (b) in Figure 4.2, with 100 agents. The visibility radius is 5 and α is set to 0.5, but the results are similar for other values of these parameters. Throughout this chapter, the settings for each table were also used for the corresponding coherence figure.

Table 4.1 shows that as w_{max} increases, the number of nodes expanded increases as well. The reason for this is that if the weight is higher, the agent will need to expand more nodes to find paths around high-cost edges, as was explained in Section 3.5. The simulation time and distance do not change much as w_{max} is varied, but the number of collisions decreases as w_{max} increases, because agents are more likely to follow the DM since not following it is more expensive when w_{max} is higher.

Figure 4.6 shows how the coherence changes as w_{max} increases. Since the results for the different values are so close together, only a subset of the weights reported in Table 4.1 is shown. However, the general trend is the same for all values from the table. The coherence is not very sensitive to a change in w_{max}, but the trend is slightly increasing as w_{max} increases. This is because the agents are more likely to follow the DM when w_{max} is higher, but once the penalty is high enough an increase in w_{max} no longer affects the coherence of their paths.

Next, we will determine what happens as α, the learning rate for the perceptrons, changes. Table 4.2 shows an example of how some metrics change as α is varied between 0.1 and 0.9. This experiment was run on map (b) in Figure 4.2, with 100 agents and visibility radius 5. The weight for the DM, w_{max}, was set to 10, an intermediate value from the previous experiment, since the nodes expanded increases as w_{max} increases, and the number of collisions decreases as w_{max} increases.

w_{max}	2	4	6	8	10
# Nodes expanded	5698.95	5211.16	5267.53	5347.89	5438.95
Simulation time	4645.05	4581.51	4626.45	4638.99	4644.99
Distance	145.52	145.76	147.59	148.9	150.45
# Collisions	12.28	9.31	8.65	8.25	8.08
w_{max}	12	14	16	18	20
# Nodes expanded	5467.24	5736.57	5793.98	5790.29	5959.03
Simulation time	4686.18	4725.24	4742.61	4752.84	4788.18
Distance	151.81	153.58	154.56	155.37	156.21
# Collisions	7.81	7.93	7.74	7.59	7.61

Table 4.1: DM, varying w_{max}

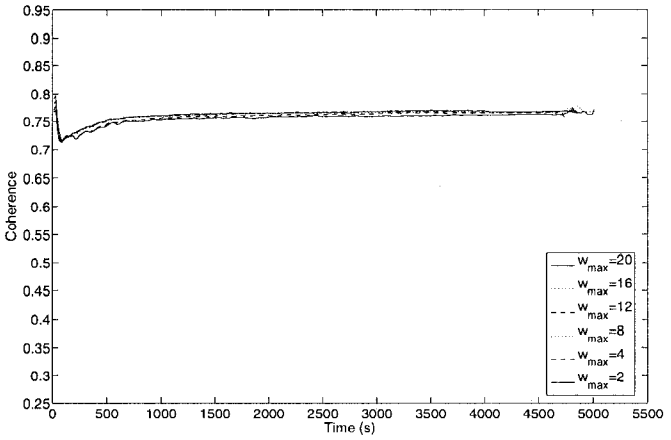


Figure 4.6: Coherence for DM does not change much as w_{max} changes

α	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Nodes	5807.23	5289.14	5306.76	5357.70	5458.28	5597.21	5492.70	5582.98	5550.96
Sim. time	4613.43	4607.61	4610.13	4644.69	4646.73	4682.67	4674.42	4701.45	4712.70
Distance	149.80	149.62	149.57	150.25	150.65	151.19	151.06	151.56	151.84
Collisions	8.55	7.89	7.87	7.92	8.10	8.17	8.02	8.25	8.23

Table 4.2: DM, varying α

Changing the learning rate does not seem to have a strong effect on these metrics, other than that a value of 0.1 performs poorly. As the table shows, there is a slight increasing trend in time and the distance travelled. The number of nodes expanded and the number of collisions oscillates, but with the exception of $\alpha = 0.1$ it only varies within a few percentage points.

Figure 4.7 shows the map coherence for each of the values for α. The figure shows that the coherence increases quickly for lower values of α, but after $\alpha = 0.5$ the graph levels off at approximately the same coherence. The difference is that it takes the DM longer to learn with lower values of α.

Next, we will analyze the behaviour as the visibility radius r changes. For this experiment, the radius is varied between 2 and 10, and the results are shown in Table 4.3. The data is shown for two maps since the behaviour of maps (b) and (c) is different from maps (a) and (d). The reason for this is that the heuristic is more accurate on maps (a) and (d) than it is for maps (b) and (c). For these experiments, w_{max} is set to 10, and the map contains 100 agents. We set $\alpha = 0.5$ since Figure 4.7 shows that the coherence increases as α increases from 0.1 to 0.5, but it does not increase much as α increases beyond 0.5.

Both the time taken to finish the simulation and the distance travelled by the agents increases for maps (b) and (c), because the agents plan paths around other agents, but it does not change much for maps (a) and (d). The reason for this is that the heuristic does not

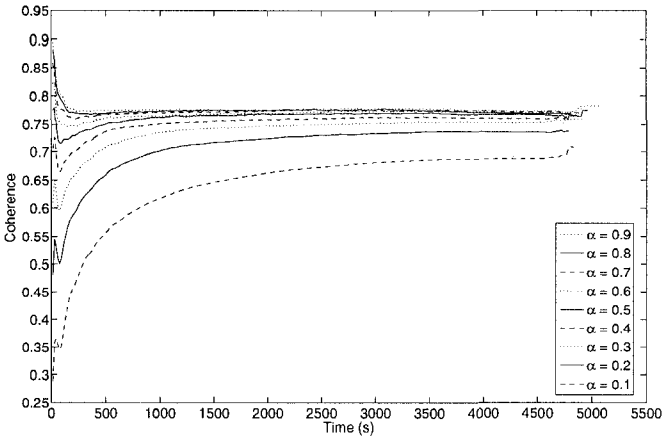


Figure 4.7: Coherence for DM, for different values of α

Map (a)

Radius	2	3	4	5	6	7	8	9	10
Nodes	1510.93	1473.74	1474.49	1470.37	1460.98	1464.68	1456.17	1475.43	1480.58
Sim. time	4118.37	4132.29	4089.12	4093.98	4065.33	4083.30	4063.08	4099.56	4049.64
Distance	83.98	83.52	83.45	83.34	83.22	83.18	83.16	83.17	83.08
Collisions	1.39	1.26	1.23	1.20	1.19	1.19	1.17	1.16	1.15

Map (c)

Radius	2	3	4	5	6	7	8	9	10
Nodes	2377.06	3182.47	3599.1	4001.19	4426.97	4755.75	5236.31	5768.61	6226.42
Sim. time	4085.55	4231.47	4327.41	4382.55	4449.12	4478.97	4539.24	4564.11	4618.35
Distance	120.59	126.52	129.50	131.60	133.97	135.62	137.34	139.13	139.82
Collisions	4.49	5.80	6.20	6.59	6.99	7.21	7.53	7.85	8.03

Table 4.3: DM, varying visibility radius

lead the agents astray on those last two maps. The average number of nodes expanded per agent per loop is usually not affected much, but in the case of map (c) the number of nodes expanded increases, because the center area sometimes gets very congested. The average number of collisions per agent, per loop decreases for most maps as the radius increases. This is to be expected since the agents can see other agents from farther away. However, the number of collisions increases for map (c) in Figure 4.2. This, too, is caused by the fact that the area in the middle of that map gets very congested. On this map, being able to see more of the other agents does not help because the agent does not know whether the other agents are moving towards it or away from it.

Figure 4.8 shows that the coherence is not affected much by a change in visibility radius. The only exception is that radius 2 performs better for the first part of the simulation, because the agent does not move out of the way of other agents who may be moving in the same direction.

Next, we will evaluate how the performance changes as the number of agents on the map increases. Table 4.4 shows the results for 20 to 160 agents, on Map (d) in Figure 4.2, with $w_{max} = 2$, visibility radius 5, and $\alpha = 0.5$. As we would expect, the performance gets worse

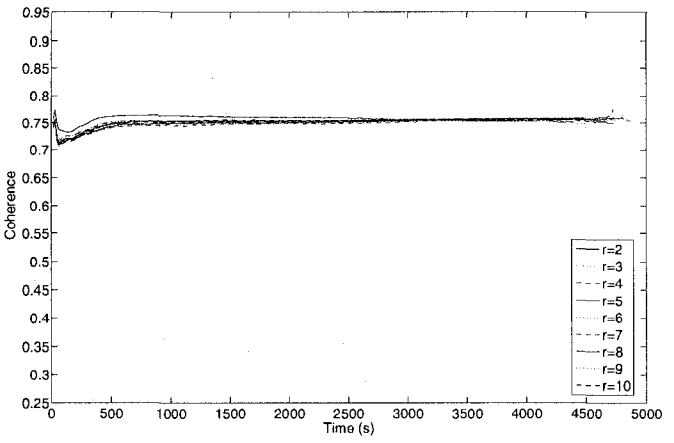


Figure 4.8: Coherence for DM does not change much as the visibility radius changes

as the number of agents increases. The number of failed moves increases because there are more other agents to collide with. As a result, agents need to re-plan more often, which increases the average number of nodes expanded. The distance and time increase because the agents need to take longer paths to avoid other agents. These results are consistent across the different maps.

# agents	20	40	60	80	100	120	140	160
Nodes expanded	649.66	831.45	969.90	1120.92	1213.41	1327.25	1480.54	1622.55
Simulation time	3522.51	3657.75	3760.86	3782.79	3840.93	3924.36	3977.4	4039.86
Distance	110.88	110.85	111.67	112.43	112.93	113.03	114.07	114.62
Collisions	0.37	0.69	1.03	1.36	1.74	2.14	2.65	3.23

Table 4.4: DM, varying the number of agents on the map

Figure 4.9 shows how the coherence changes as the number of agents increases. The coherence decreases as the number of agents increases because agents need to take more other agents into account, which will make movement less coherent.

4.2 Comparing Direction Maps to Other Approaches

Now we have some idea of how the parameter settings affect the performance of the DM approach, we will compare it to two other multi-agent pathfinding approaches: Local-Repair A* (LRA*) and Windowed Hierarchical Cooperative A* (WHCA*). These were discussed in Section 2.5.

Table 4.5 compares the performance of LRA*, DM, and WHCA* on map (c) for 100 agents. The DM uses $w_{max} = 10$, and WHCA* uses window size 16. The perceptron learning rate is $\alpha = 0.5$. The visibility radius for DM and LRA* set to 5. The results are similar for the other maps.

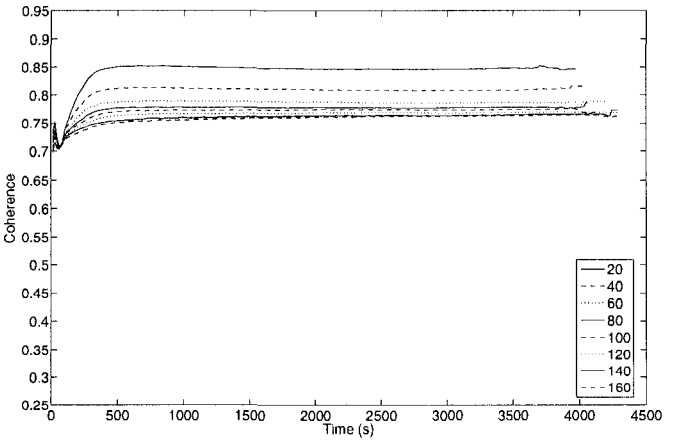


Figure 4.9: Coherence for DM, for different numbers of agents

The table shows that the number of nodes expanded is smallest for LRA* and highest for WHCA*. We expect WHCA* to expand the most nodes because it performs a search both in time and in space. We expect the DM method to expand more nodes than LRA* because it uses modified edge weights which require it to search around more. The time and distance are smallest for WHCA*, and largest for LRA*. WHCA* tries hard to be optimal, so it finds shorter paths and takes less simulation time than either the DM or LRA*. The paths are longer and more time is used by LRA* than by the DM, because agents have to revise their plans every time they collide, which happens significantly more often for LRA* agents. The number of collisions is highest for LRA*, which does not take the paths of other agents into account, and lowest for WHCA*, which plans around the paths of other agents within the window.

Figure 4.10 shows the coherence for these three approaches. The coherence is lowest for WHCA*, because it tries hard to be optimal and therefore sometimes plans convoluted paths. The coherence is highest for the DM because the agents take the movement of other agents into consideration when they plan. Near the end of the simulation, the coherence for LRA* increases. This happens when some of the agents have finished their simulations. The remaining agents can plan straighter paths because they do not need to move around other agents, and this increases the coherence.

	DM	LRA*	WHCA*
# Nodes expanded	4604.71	2184.32	5980.71
Simulation time	4389.77	4996.44	120.92
Distance	132.25	139.18	3508.32
# Collisions	6.90	19.60	3.00

Table 4.5: Comparison of performance of DM, A*, and WHCA*

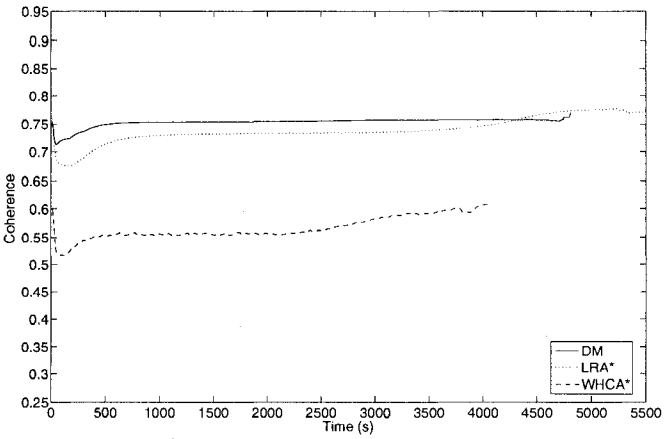


Figure 4.10: Coherence for DM, LRA* and WHCA*

4.3 Weighted A*

Table 4.5 showed that the DM approach expands a lot of nodes compared to LRA*, and one way to reduce this is by using Weighted A* (WA*) instead of A*, both in the DM case and in the case of Local-Repair A*. Weighted A* was discussed at the end of Section 2.2.1.

Table 4.6 shows how the results change when Weighted A* is used, both with and without using a DM, as well as what happens when the A* weight is varied from 2 to 4 to 6. The results shown here are for map (c) in Figure 4.2, with visibility radius 5 and $\alpha = 0.5$. The DM used $w_{max} = 10$.

Notice that using weighted A* does reduce the number of nodes expanded; especially for the DM. In fact, the DM now expands less nodes than LRA*. This is because the search has more of a depth-first aspect, so the algorithm will expand nodes with higher g-cost than regular A* would. Since the weighted edges make the heuristic less accurate, this performs better for the DM method.

When LRA* is used, the other metrics only change within a few percentage points for the different A* weights. In the case of the DM, however, time, distance and the number of collisions increase as the A* weight increases. The number of collisions increases as the A* weight increases because the g-cost portion of the evaluation function now has a smaller effect on the total f-cost of a node, so the DM influences the agent's decision less. As the number of collisions increases, the agent has to re-plan more often, which is the reason for the increase in distance and time.

Figures 4.11 and 4.12 show the coherence for these two approaches. Although the differences are small, the coherence is higher when WA* is used, and it increases as the weight increases. The reason for this is that when agents plan using weighted A*, they plan to

	LRA*	WA*(2)	WA*(4)	WA*(6)
# Nodes expanded	2187.73	1821.72	1737.93	1621.30
Time	4993.83	5046.06	5228.79	5292.24
Distance	140.00	141.95	142.20	142.40
# Collisions	19.92	19.37	17.48	16.14
	DM	WA*(2)	WA*(4)	WA*(6)
# Nodes expanded	3913.05	1353.95	1415.57	1546.99
Time	4389.77	4384.22	4559.52	4724.59
Distance	131.26	130.92	136.90	139.85
# Collisions	6.44	6.71	10.81	13.18

Table 4.6: Comparison of performance of LRA*, DM, with regular A* vs. weighted A*

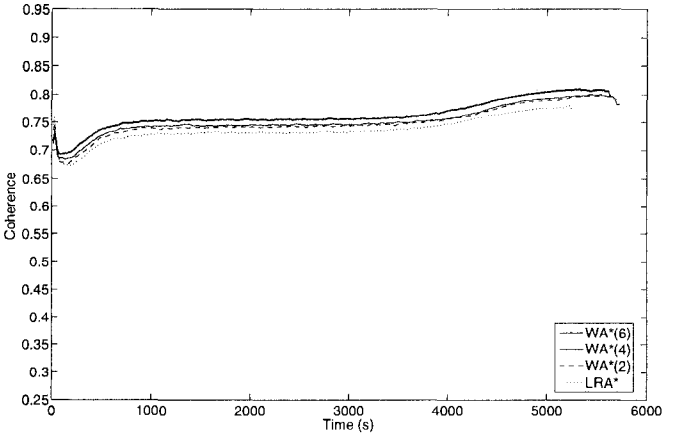


Figure 4.11: Coherence for LRA* and WA*, no DM

reach the large obstacles in the middle quickly, then move along the side of the obstacle for until it reaches the corridor that connects the two sides of the map, and then plans a path straight to its goal. This forms lanes like the ones shown in Figure 4.13. When Weighted A*

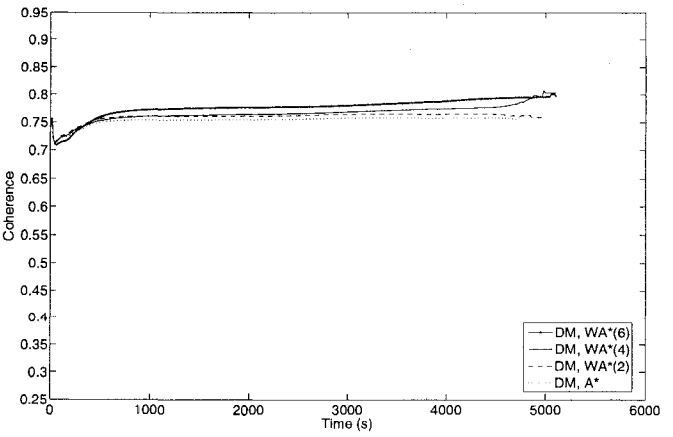


Figure 4.12: Coherence for DM with LRA* and WA*

Sector size	4	8	12	16
# Nodes expanded	4125.68	2882.29	2752.92	4231.32
Time	5028.03	4970.64	4747.80	5240.94
Distance	149.60	142.84	136.30	142.00
# Collisions	13.40	10.78	7.57	10.43

Table 4.7: DM with A* using abstraction with full refinement

is not used, agents moving towards the center generally do not stay as close to the obstacles, so there is less clear lane forming, resulting in lower coherence.

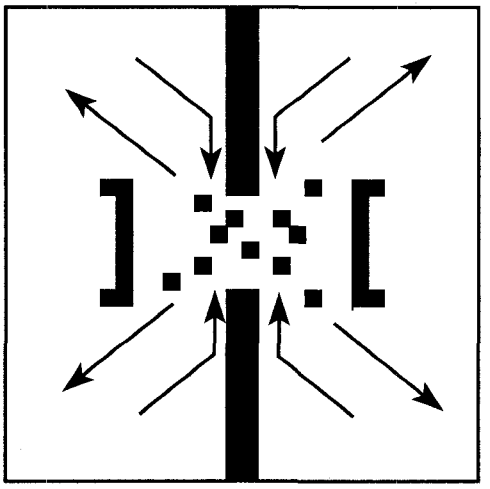


Figure 4.13: Lanes formed by WA*

4.4 Abstraction

Another approach that has been used to reduce the amount of work done during search is abstraction, which was first discussed in Section 2.4 and later applied to DMs in Section 3.5. Abstraction is also useful because it allows the agent to find a high-level path and refine the path bit by bit, which reduces the amount of work done per time step and can be used when real-time performance is required.

Two sets of experiments are presented here. First, we will show results for complete refinement, and then for the partial refinement case. In complete refinement, the agent first finds a solution in the abstract graph and then refines the entire path before executing. Table 4.7 shows these results for four different sector sizes. The experiment was performed on map (c) in Figure 4.2, with 100 agents, $r = 5$, $w_{max} = 10$, $\alpha = 0.5$, and regular A*.

The table shows that the performance is best when sectors of size 12 are used. This is not the case for all of the maps, but it is generally true that the number of nodes expanded decreases first and then increases. When the sectors are too small, the abstract path is less general and the abstract search is more expensive. On the other hand, when the sector size

is too large, the abstract search is easy, but refining is more expensive because the abstract nodes lie farther away from each other.

When we compare the performance to the DM without abstraction, we see that the nodes expanded are reduced only when the right sector size is chosen. In the worst case, abstraction performs roughly the same amount of work as the DM without abstraction does.

The other metrics are worse when abstraction is used, because agents use the abstract path as a guide, which restricts the agent during planning, compared to when abstraction is not used.

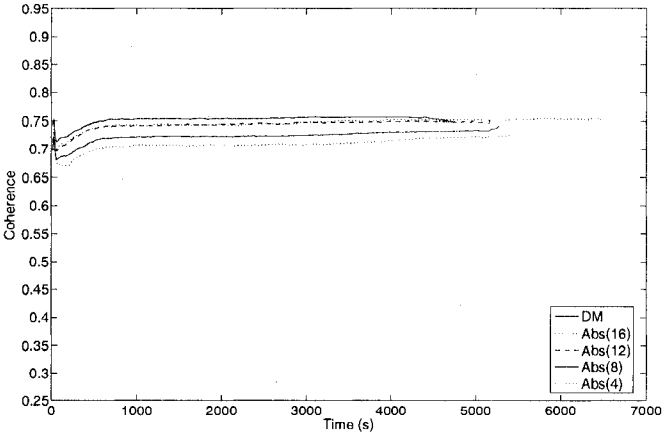


Figure 4.14: Coherence for DM with abstraction and full path refinement

Figure 4.14 shows how the map coherence changes as the sector size increases. The coherence is lower when abstraction is used, again because the abstract path restricts the agent during the refinement step. The coherence increases as the sector size increases because the agent is less restricted during refinement.

In Section 3.5 we also described how abstraction can be used for partial pathfinding. After an abstract path is found, only part of it is refined. A path is planned not to the next abstract node, but the one after it, and this path is cut off after some portion of the resulting partial path. Here, the partial paths are cut off after 60%, and the experiment was performed on the same map and with the same parameters as the full refinement case above.

Table 4.8 shows the results for abstraction with partial refinement, including how the metrics change as the sector size changes. The performance is better than with full-refinement abstraction, because the agents only plan short paths at a time. The number of nodes expanded is not increased because the abstract search is fast.

The best performance is again found when the sector size used by the abstraction is 12, or in general, some intermediate value, for the same reason as in the full refinement case.

Sector size	4	8	12	16
# Nodes expanded	2431.62	2078.51	2729.98	2795.56
Time	4741.80	4580.88	4537.38	4491.18
Distance	140.69	135.66	132.05	131.77
# Collisions	8.85	7.81	7.15	7.18

Table 4.8: DM with A* using abstraction with partial refinement

Figure 4.15 shows the coherence for abstraction with partial path refinement. The result is very similar to what was shown in Figure 4.14, and for the same reasons. The coherence is highest when abstraction is not used, but the coherence increases as the sector size increases because the agents are less restricted during planning.

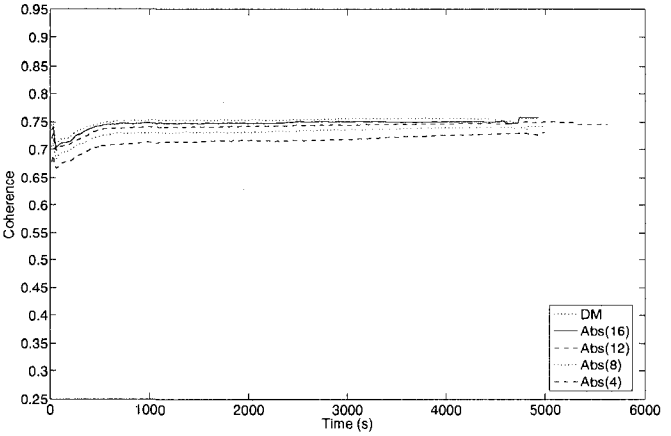


Figure 4.15: Coherence for DM with abstraction and partial path refinement

4.5 Updating Surrounding Locations

In Section 3.4 we suggested updating the DVs for the locations that surround the agent's current location as a way to increase coherence. Visually, the DM is more coherent when surrounding locations are updated. This is shown in Figure 4.16. Although the left-hand figure also shows clear lane formation, the right-hand DM contains wider, more obvious lanes.

Table 4.9 shows the metrics for updating surrounding locations as the update parameter for the surrounding locations is varied. The data shown is for map (d) in Figure 4.2, with 100 agents, $w_{max} = 10$, $r = 5$, and $\alpha = 0.5$. In addition to an agent's current location, we update DVs for the eight surrounding locations. The number of nodes expanded, time taken, and distance travelled all are higher than they are for the regular DM, and they increase as the surround parameter α_s is increased. This is because a larger portion of the map now contains DVs, which the agents need to take into account during planning. The

number of collisions is slightly greater when the surrounding locations are updated, because where the wider lanes cross, there is a larger area of contention, and more collisions occur.

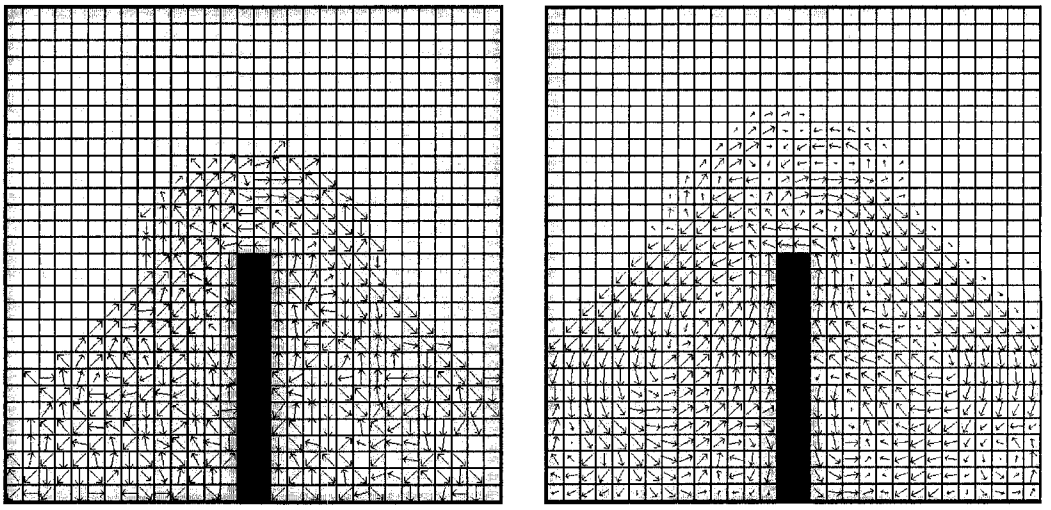
Figure 4.17 shows how coherence changes as the update parameter for the surrounding locations is varied, and how it compares to the coherence of the regular DM approach. The coherence is higher than with the regular DM, and increases as α_s increases.

4.6 Local Direction Map Approaches

Next, we evaluate the performance of two approaches in which the agents only have local access to the DM. These approaches were described in Section 3.3.

First, we will look at the results when we allow each agent to only see the DM within some radius. The agent plans a path, using the DM only within that radius, and it cuts off the planned path at the edge of the window. Table 4.10 shows the data for the regular DM approach and compares it with three different values of the local radius. The numbers shown here are for map (d) in Figure 4.2, with 100 agents, for $w_{max} = 10$, radius 5, and $\alpha = 0.5$. The local DM radius, *win*, is set to either 3, 5, or 7.

Compared to the regular direction map approach, this method expands a similar number of nodes for smaller window sizes, and less for larger window sizes, because the agent only



(a) (b)

Figure 4.16: DMs when surrounding locations are not updated (left) and when they are (right)

	DM	$\alpha_s = 0.3$	$\alpha_s = 0.5$	$\alpha_s = 0.7$
# Nodes expanded	2164.46	2946.99	3264.93	3337.71
Time	4144.92	4254.30	4448.04	4540.86
Distance	122.31	122.84	124.32	126.22
# Collisions	2.06	2.59	2.54	2.58

Table 4.9: Comparison of DM with and without updates to surrounding locations

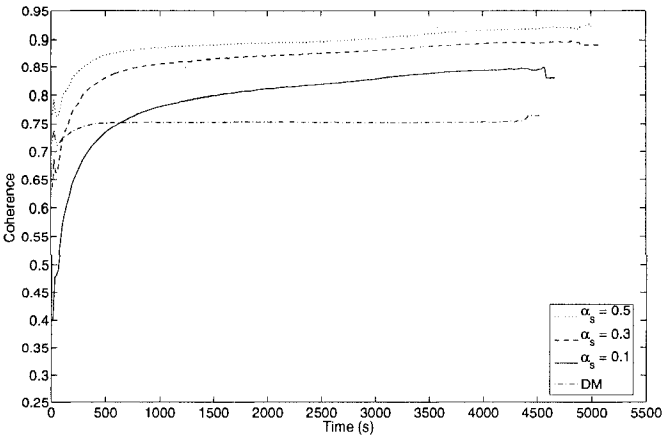


Figure 4.17: Coherence for DM and 3 different values for the learning parameter for surrounding locations

takes the DM into account within a small radius. However, if the radius is very small, the agent may have to re-plan many times before it reaches its goal, so the number of nodes expanded is not minimized at the lowest value of the DM visibility radius. The time taken by the agent and the distance travelled is slightly larger than with the regular DM, because the agent does not have global knowledge of the DM. As a result, the agent may plan a different path than it would if it had global knowledge of the DM. As the agent can see more of the DM around it, *i.e.* if the radius is larger, the time and distance are reduced as well, for the same reason. The number of collisions is lower, because the agent only plans its path until the edge of the DM window. It only moves a few steps each time before it re-plans. The number of collisions increases as the window size increases because the agent takes more steps before re-planning, so other agents are more likely to have moved in the agent's way.

Figure 4.18 shows the coherence for each of the window sizes. The coherence when a DM window is used is lower, but increases as the window size increases. This is, again, because the agent does not have a global view of the DM and the path it chooses may be following the DM only in a local sense.

Next, we look at the case where each agent maintains its own copy of the DM and updates it locally. Tables 4.11 and 4.12 show the results for different local update radii.

	DM	win = 3	win = 5	win = 7
# Nodes expanded	2164.46	2205.27	1316.56	988.46
Time	4144.92	5462.62	4789.50	4625.64
Distance	122.31	158.88	150.48	142.50
# Collisions	2.06	1.59	1.89	1.97

Table 4.10: Comparison of regular DM with local DM window

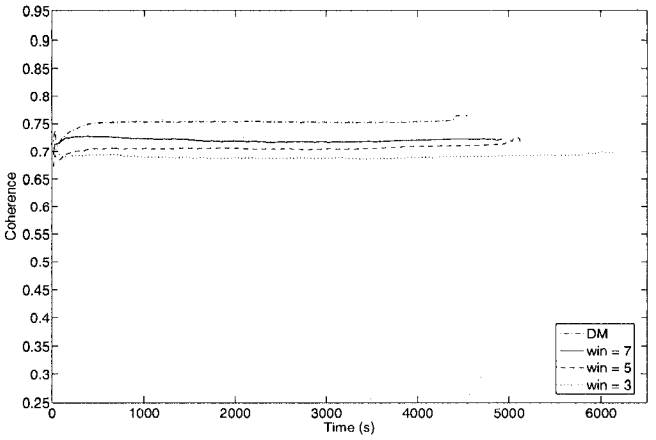


Figure 4.18: Coherence for DM with local DM window

The experiment was performed with 100 agents, $r = 5$, $w_{max} = 10$ and $\alpha = 0.5$. Data is presented for two maps: maps (b) and (d) in Figure 4.2, because the results are different for maps in which the heuristic is less accurate (maps (b) and (c)) than for the other two maps (maps (a) and (d)).

First, we look at the results for map (b), presented in Table 4.11. The results on map (c) are similar. The number of collisions is significantly higher when the local approach is used. This is because the agent does not have up-to-date global knowledge of the DM, and therefore it may not follow the DM for its entire path. The increase in collisions forces the agent to re-plan more often, which increases the number of nodes expanded.

Figure 4.19 shows the coherence for map (b) when the agents maintain their own copies of the DM. The coherence is lower than when the regular DM approach is used, because the agents do not follow the DM as closely as when they use the global DM. Therefore, movement is less coherent, leading to a reduction in map coherence.

Next, we will look at the results on map (d). The results for map (a) are similar. Maps (a) and (d) contain more open space, and they do not contain large, central areas of congestion like maps (b) and (c) do. Therefore, the DM does not aid the agents much in terms of reducing the number of collisions. As a result, when the agents maintain their own copies of the map, possibly containing DVs that are no longer accurate, this does not

	DM	$r_{local} = 3$	$r_{local} = 5$	$r_{local} = 7$
# Nodes expanded	5438.95	5986.98	5995.10	6007.02
Time	4644.99	4780.38	4761.36	4762.47
Distance	150.45	145.39	145.47	145.42
# Collisions	8.08	16.34	16.20	16.15

Table 4.11: DM with locally updated copies of DM for each agent, map (b) in Figure 4.2

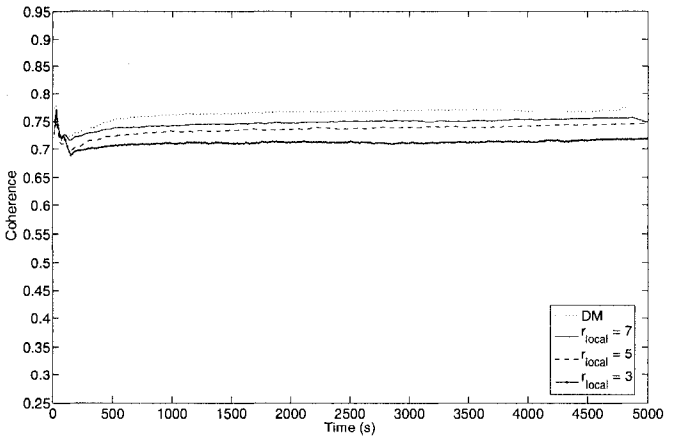


Figure 4.19: Coherence for DM with locally updated copies for each agent, map (b) in Figure 4.2

hurt the performance. This is shown in Table 4.12. The number of collisions is similar to when the regular DM is used, but the number of nodes expanded is reduced. The reason for this is that the agent's copy of the DM reflects the state of the DM when the agent passed through each location. Therefore, the DVs will point roughly in the direction of the agent's path. Therefore, the costs of the edges the agent wants to travel are not much higher than regular edge costs, so the heuristic is fairly good in this case. This reduces the number of nodes expanded. In addition, since the agent does not take the global DM into account, it is able to follow a straighter, more direct, path to its goal, which reduces the distance covered as well as the simulation time.

	DM	$r_{local} = 3$	$r_{local} = 5$	$r_{local} = 7$
# Nodes expanded	2164.46	1015.05	1013.67	1020.02
Time	4144.92	3799.22	3766.50	3779.70
Distance	122.31	110.09	109.90	109.98
# Collisions	2.06	2.30	2.18	2.13

Table 4.12: DM with locally updated copies of DM for each agent, map (d) in Figure 4.2

Figure 4.20 shows the map coherence when each agent maintains its own copy of the DM, on map (d) in Figure 4.2. The coherence is higher when individual DMs are maintained, because the agent's path leads the agent more directly to the goal, as was explained in the previous paragraph.

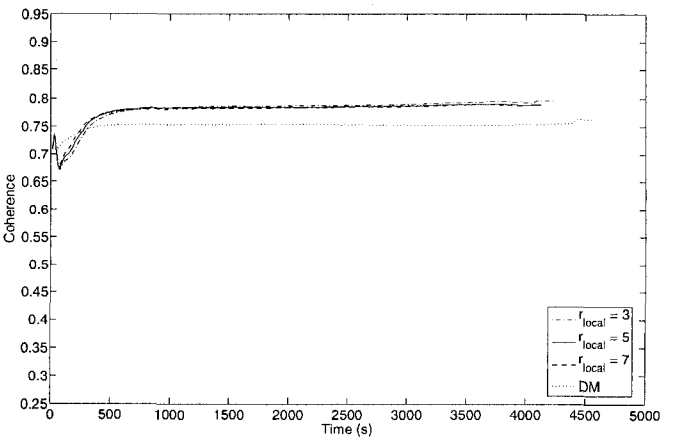


Figure 4.20: Coherence for DM with locally updated copies for each agent, map (d) in Figure 4.2

Chapter 5

Conclusions and Future Directions

In Chapter 3 we have presented a new approach for multi-agent path planning. Rather than taking into account static data about the current location of other agents or fully dynamic data about the paths that other agents have planned, we use static data about the dynamics of the world during planning.

The approach is based on direction maps, a shared data structure which stores a direction vector for each location that has been visited. A direction vector for grid cell a is a prediction of the direction in which an agent will pass through a, and it is learned from the directions in which agents have previously passed through this same location.

During planning, the agents use the direction map to guide them. They are encouraged to follow the direction vector through a modification of the movement cost, which leads to emergent behaviour such as lane forming. The result is highly coherent behaviour, with very few collisions compared to Local-Repair A*.

In Chapter 4 we showed that the basic direction map approach leads to fewer collisions than Local-Repair A*, while expanding less nodes than Windowed Hierarchical Cooperative A*.

We also introduced a new metric, map coherence, which is an indication of how coherently agents move across the map, based on the direction map. Map coherence is higher when direction maps are used than when the agents use LRA* or WHCA*.

Chapter 4 also showed that the number of nodes expanded can be reduced by using Weighted A*(WA*) rather than regular A* during search with direction maps. Using WA* increases the number of collisions but it is still significantly lower than when Local-Repair A* is used.

Direction maps can also be combined with abstraction and partial refinement, which reduces the amount of work done and makes the approach suitable for use in real-time environments.

In addition, we showed that agents can update more than just their current location to generate wider lanes, and that it is possible to use direction maps even when only local information is available to the agents.

Although some extensions to the basic DM approach were discussed in this thesis, there are many research directions which have not yet been explored.

5.1 Combining Direction Maps with Flocking

As was mentioned in Section 2.6, pathfinding and flocking or steering techniques are complementary in a sense. If a group of agents needs to move towards a specific goal, flocking techniques require a path to guide the group in the right direction, and the flock could use a direction map for this. As an example, consider the scenario from Chapter 1 again, shown in Figure 5.1. Assume that a group of agents needs to move back and forth between the left- and right-hand side of the map. An efficient way to do this is to move left-to-right through one of the corridors in the center, and use the other one for right-to-left movement, which is not done by default by flocking approaches. The DM could help guide the flock through the correct corridor, while the flocking mechanism would ensure that the agents pass through the corridors without colliding.

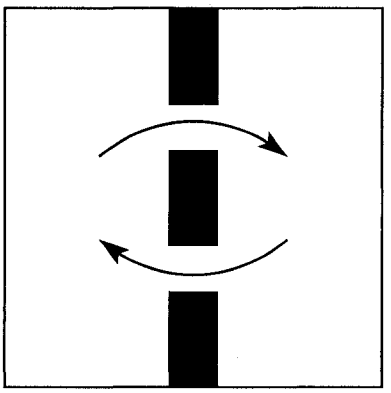


Figure 5.1: An example of a situation where flocking and direction maps could be combined

5.2 Decaying Direction Vectors

In the experiments from Chapter 4, the patrol locations for the agents remained the same throughout the experiment, but in some scenarios they might change. For example, in an RTS game the agents may have exhausted a particular resource and they may start patrolling between the home base and a different resource. We want the DM to be able to adapt to such changes, but so far no experiments have been done to investigate how the DM behaves when patrol locations change.

The environment may change for other reasons too, for example because the world is dynamic. Perhaps an obstacle appears somewhere on the map, in which case we want the DM to change and lead the agents around the obstacle. Since the agents are forced to re-plan after colliding with the obstacle, it is likely that the DM will be changed appropriately. On the other hand, it is possible that a new corridor opens up, perhaps because a door that was previously closed is now open. This could create a shortcut, but the agents may not take this shorter path since the direction map is telling them to take a different route. One possible solution for this is to decay the direction vectors over time. This is somewhat similar to the diffusion of ant pheromones.

There may be other benefits from decaying direction vectors. In Section 4.1.1 we mentioned that the agents sometimes take routes that are suboptimal because of the way the direction map initially gets set. For example, consider the situation in Figure 5.2, where the solid black lines indicate the lanes the agents use. In addition, there is a middle lane, indicated by a dotted line, which runs from right-to-left but is not used by any agents. Perhaps if the direction vectors were decayed over time, the agents that are currently following the top arrow would create a right-to-left lane where the dotted arrow is. This could reduce the distance travelled by the agents.

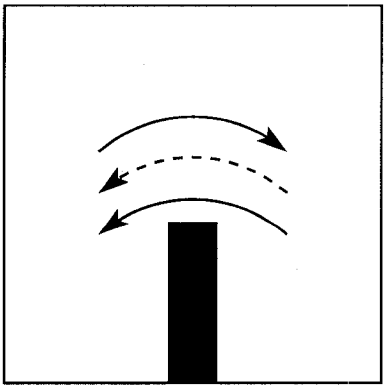
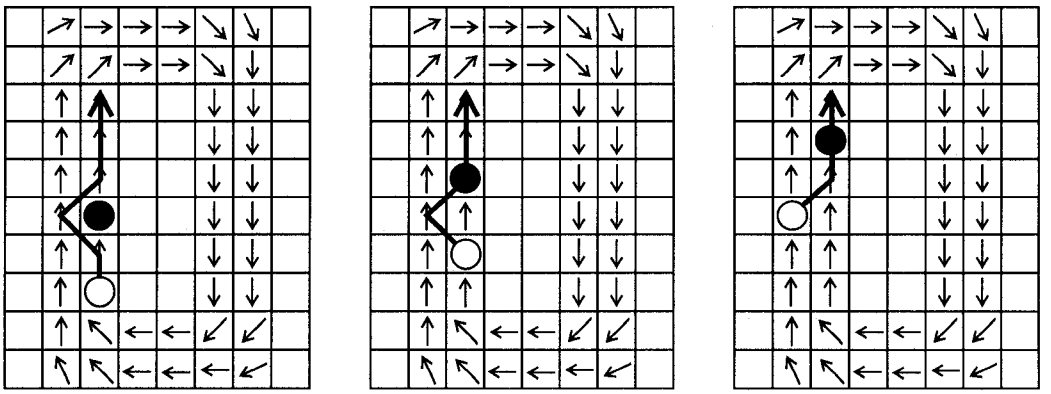


Figure 5.2: Example of a situation where DV decaying may be beneficial

5.3 Using the Direction Map to Predict Movement of Other Agents

In the current implementation, the agents plan around other agents within their visibility radius without taking into account the direction in which this other agent is moving. This can sometimes lead to odd-looking behaviour, for example as shown in Figure 5.3. Here, the white agent wants to follow the arrows around. If its visibility radius is large enough, it will detect the black agent when it plans its path. It then plans a path around the black

agent's current position, as is shown in Figure 5.3(b), even though the black agent is most likely moving in the same direction.



(a) (b) (c)

Figure 5.3: Other agents' movement direction is not taken into account during planning

A way to improve the behaviour of the agents would be to take into account the direction map when considering other agents. For example, we could only view other agents as obstacles when they are either directly beside the agent who is currently planning, or if the agent is likely to move towards the agent if it follows the direction map.

5.4 Abstraction

The way in which abstraction was combined with the direction map, as described in Section 3.5, is not the only way this can be done. The method that was implemented is to perform a regular A* search in the abstract graph, and then refine using the DM on the map level. Instead, we could store DVs for the abstract nodes, which would indicate a general direction for the sector. If the sector size is small, we may want to refine the abstract path without considering the map DM, but if the sector size is larger it may be useful to use the DM on the map level as well.

5.5 Learning Direction Vectors

The DVs have only been learned using the perceptron update rule. Other approaches could be used for this; perhaps a reinforcement learning method where the rewards have some relation to the length of the path taken and the number of collisions. This way, it may be possible to learn a more efficient direction map, but the direction vectors could not be updated after each step anymore since learning would need to be done on a per-loop basis.

5.6 Using Direction Maps in Video Games

In the current state, this approach may not be suitable for use in commercial video games, but there are some modifications which can be made. Two ideas are discussed here: loading a pre-built DM, and DMs for maps that are not grid-based.

For the first while, as the DM is learned, the number of collisions is higher than later in the simulation. This initial behaviour may not be acceptable for use in a commercial game, but it is easy to run the simulation off-line, and load a fully learned DM within a game. This way, the agents will be able to move coherently without having to learn the DM first.

Another reason why this approach as it is currently implemented may not work in video games is that it is designed for a grid-based map in which each agent occupies a cell. In practice, not all environments are grids, and not all grids are as fine as the ones used here. The direction map approach can be modified to work with different types of maps by defining a grid of appropriate coarseness on the map. For example, a direction vector could be stored for a entire corridor or for a doorway. In addition, we could just store DVs for parts of the map where collisions are more likely; crowded areas or narrow passageways are examples of this.

5.7 Conclusion

Although the idea of direction maps is fairly simple, it leads to lane formation and coherent movement. In addition, modifications can be made to the basic approach to suit the requirements for a particular application. For example, in Chapter 4 we showed that higher coherence can be achieved by updating the DVs for locations surrounding an agent's path, and that Weighted A* or abstraction can be used to reduce the number of nodes expanded. Overall, using direction maps is a promising technique for multi-agent pathfinding.

Bibliography

[1] http://www.cs.ualberta.ca/ nathanst/hog.html.

[2] R. Arkin. Motor schema based navigation for a mobile robot: An approach to programming by behavior. In *Institute of Electrical and Electronics Engineers (IEEE) International Conference on Robotics and Automation*, volume 4, pages 264–271, March 1987.

[3] A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path–finding. *Journal of Game Development*, 1(1):7–28, 2004.

[4] I. Couzin and N. Franks. Self-organized lane formation and optimized traffic flow in army ants. *Proceedings of the Royal Society of London, Series B*, 270(1511):139–146, January 2003.

[5] H. Davis, A. Bramanti-Gregor, and J. Wang. The advantages of using depth and breadth components in heuristic search. *Methodologies for Intelligent Systems 3*, pages 19–28, 1989.

[6] D. Demyen and M. Buro. Efficient triangulation-based pathfinding. In *The Twenty-First National Conference on Artificial Intelligence (AAAI-06)*, pages 942–947. AAAI Press, 2006.

[7] M. Dorigo, E. Bonabeau, and G. Theraulaz. Ant algorithms and stigmergy. *Future Generation Comp. Syst.*, 16(8):851–871, 2000.

[8] M. Dorigo, V. Maniezzo, and A. Colorni. The Ant System: Optimization by a colony of cooperating agents. *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41, 1996.

[9] K. Dresner and P. Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 31:591–656, March 2008.

[10] P. Egbert and S. Winkler. Collision-free object movement using vector fields. *Institute of Electrical and Electronics Engineers (IEEE) Computer Graphics and Applications*, 16(4):18–24, 1996 1996.

[11] J. Fulton and J. Pransky. DARPA grand challenge - a pioneering event for autonomous robotic ground vehicles. *Industrial Robot: An International Journal*, 31(5):414–422, 2004.

[12] A. Geramifard, P. Chubak, and V. Bulitko. Biased cost pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 112–114, 2006.

[13] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Institute of Electrical and Electronics Engineers (IEEE) Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[14] D. Helbing and P. Molnár. Social force model for pedestrian dynamics. *Physical Review E*, 51:4282–4286, 1995.

[15] R. Holte, T. Mkadmi, R. Zimmer, and A. MacDonald. Speeding up problem solving by abstraction: A graph oriented approach. *Artificial Intelligence*, 85(1-2):321–361, 1996.

[16] R. Jansen and M. Buro. HPA* enhancements. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 84–87. The AAAI Press, 2007.

[17] R. Korf. Depth-first iterative deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[18] D. Lay. *Linear Algebra and its Applications*, chapter 6. Addison Wesley, third edition, 2003.

[19] E. Losh. The palace of memory: Virtual tourism and tours of duty in tactical iraqi and virtual iraq. In *CyberGames '06: Proceedings of the 2006 International Conference on Game Research and Development*, pages 77–86. Murdoch University, 2006.

[20] R. Murphy. Trial by fire: Activities of the rescue robots at the world trade center from 1121 september 2001. *Institute of Electrical and Electronics Engineers (IEEE) Robotics and Automation Magazine*, 11(3):50–61, September 2004.

[21] H. Parunak, M. Purcell, and R. O'Connel. Digital pheromones for autonomous coordination of swarming UAV's. In *Proceedings of First American Institute of Aeronautics and Astronautics (AIAA) Unmanned Aerospace Vehicles, Systems, Technologies, and Operations Conference*, 2002.

[22] D. Payton, R. Estkowski, and M. Howard. Compound behaviors in pheromone robotics. *Robotics and Autonomous Systems*, 44(3-4):229–240, 2003.

[23] I. Pohl. First results on the effect of error in heuristic search. *Machine Intelligence*, 5:219–236, 1970.

[24] I. Pohl. Heuristic search viewed as path finding in a graph. *Artificial Intelligence*, 1:193–204, 1970.

[25] C. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[26] C. Reynolds. Steering behaviors for autonomous characters. In *Game Developers Conference*, 1999.

[27] C. Reynolds. Interaction with groups of autonomous characters. In *Game Developers Conference*, 2000.

[28] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ, 2002.

[29] D. Silver. Cooperative pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 117–122, 2005.

[30] N. Sturtevant. Memory-efficient abstractions for pathfinding. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 31–36, 2007.

[31] N. Sturtevant and M. Buro. Partial pathfinding using map abstraction and refinement. In *AAAI*, pages 1392–1397, 2005.

[32] N. Sturtevant and M. Buro. Improving collaborative pathfinding using map abstraction. In *Artificial Intelligence and Interactive Digital Entertainment*, pages 80–85, 2006.

[33] N. Sturtevant and R. Jansen. An analysis of map-based abstraction and refinement. In *Symposium on Abstraction, Reformulation, and Approximation*, pages 344–358, 2007.

[34] A. Treuille, S. Cooper, and Z. Popović. Continuum crowds. In *Special Interest Group on GRAPHics and Interactive Techniques (SIGGRAPH) '06: ACM SIGGRAPH 2006 Papers*, pages 1160–1168, New York, NY, USA, 2006. ACM.