UNIVERSITY OF CALIFORNIA

Los Angeles

Optimal and Anytime Approximation Algorithms for Cooperative Pathfinding Problems

A thesis submitted in partial satisfaction

of the requirements for the degree

Master of Science in Computer Science

by

Trevor Scott Standley

2010

The thesis of Trevor Scott Standley is approved.

Judea Pearl

Adnan Darwiche

Richard Korf, Committee Chair

University of California, Los Angeles

2010

TABLE OF CONTENTS

1 Introduction **1**

1.1 Problem Formulation 1

1.2 Related Work 4

1.3 Problem Complexity 5

2 Finding Optimal Solutions **10**

2.1 The Standard Admissible Algorithm 10

2.2 Operator Decomposition 12

2.3 The Heuristic 17

2.4 Partial Expansion 19

2.5 Independent Subproblems 21

2.6 Experiments 25

3 Approximation Algorithms **28**

3.1 Complete Approximation Algorithms 28

3.2 Lower Bound Algorithms 30

3.3 Anytime Approximation Algorithms 31

3.4 Experiments 33

4 Conclusion **37**

References **39**

LIST OF FIGURES

1.1 A small random grid world instance. Cars represent initial agent positions. Flags represent destinations. Obstacles are in black. 2

1.2 Agents cannot pass through each other. 3

1.3 Our formulation of the cooperative pathfinding problem allows rotational moves, which result in each agent occupying a previously occupied cell. 3

2.1 An example in which naïve OD would fail. 13

2.2 The search tree of A* with OD for one timestep. Pre-move positions are shown. Move assignments are marked with arrows. 14

2.3 Two pairs of similar intermediate states. 17

2.4 Running time of each instance for each algorithm. 26

2.5 Proportion of largest non-independent subproblems given by ID solved within two seconds by OD and S. 27

3.1 Running time of each instance and percent of problems solved for each approximation algorithm on 1000 very large problems. 33

3.2 Running time of each instance for each maximum group size approximation algorithm. HCA* only solved 93.8% of the problems. 34

3.3 Solution quality for our MGS algorithms (lower is better). 34

3.4 Solution quality vs. time for our anytime algorithm (lower is better). 35

3.5 Running time of each instance for each lower bound algorithm. 36

3.6 The average solution quality for each lower bound algorithm (higher is better). 36

LIST OF TABLES

1.1 The complexity of various variants of the warehouseman's problem. 8

Acknowledgments

I would like to sincerely thank Richard Korf and Dawn Chen for their many edits, suggestions and discussions. Artwork by Dawn Chen. I would also like to thank John Heidemann for making available the uclathes style file for latex which saved me untold hours of formatting work.

ABSTRACT OF THE THESIS

Optimal and Anytime Approximation Algorithms for Cooperative Pathfinding Problems

by

Trevor Scott Standley

Master of Science in Computer Science

University of California, Los Angeles, 2010

Professor Richard Korf, Chair

In cooperative pathfinding problems, non-interfering paths that bring each agent from its current state to its goal state must be planned for multiple agents. We present the first practical, admissible, and complete algorithm for solving problems of this kind. We also present a class of complete approximation algorithms, and an anytime algorithm suitable for real-time use. To this end, we first propose a technique called operator decomposition, which can be used to reduce the branching factors of many search algorithms, including algorithms for cooperative pathfinding. We then show how a type of independence common in instances of cooperative pathfinding problems can be exploited. We show empirically that these techniques dramatically improve the performance of the standard admissible algorithm for the cooperative pathfinding problem, and that their combination results in a complete algorithm capable of optimally solving relatively large problems in milliseconds. Next, we discuss how modifications to our complete and optimal algorithm can result in the first complete approximation algorithm capable of solving very large problems. We conclude by presenting an anytime algorithm that quickly finds a solution, and then uses any remaining allotted computation time incrementally improving that solution until it is optimal.

CHAPTER 1

Introduction

Pathfinding, or planning a route to a destination that avoids obstacles, is a classic problem in AI. When only a single agent is present, the problem is usually effectively solved using the A* algorithm [HNR68]. When the problem contains multiple agents, however, care must be taken to avoid computing a solution that leads any subset of agents to conflict, for example, one that requires two agents to occupy the same space at the same time. While cooperative pathfinding is becoming increasingly important in modern video games, it has many applications outside of entertainment, such as robotics, aviation, and vehicle routing [WB08, SO96]. This thesis is an extended version of [Sta10], and includes an additional chapter on approximation.

1.1 Problem Formulation

There are many distinct types of cooperative pathfinding problems, and each can be solved using a variant of our algorithm. One example is planning the motions of robotic arms, each of which must accomplish a separate goal without moving into one another. Scheduling trains in a railroad network without sending a pair of trains on a collision course, and deciding actions for automobiles approaching an intersection so that each may pass through safely and quickly [DS04] are also cooperative pathfinding problems. For the sake of clarity, simplicity, and comparison to existing algorithms, the testbed for

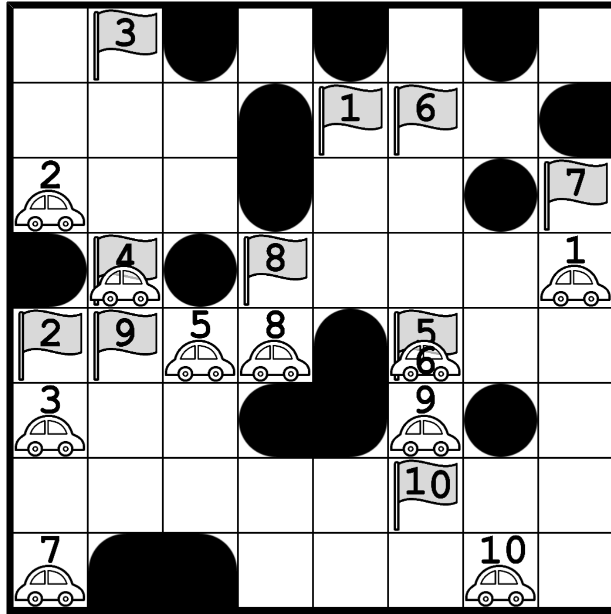

Figure 1.1: A small random grid world instance. Cars represent initial agent positions. Flags represent destinations. Obstacles are in black.

our algorithm will be an eight-connected grid world like the one in Figure 1.1.

In this formulation, each agent occupies a single cell of the grid world, and has a unique destination. During a single timestep, each agent either waits, or moves to any of its free adjacent cells. A cell is free if it does not contain an obstacle. Diagonal moves are allowed even when the two cells adjacent to both the starting cell and the diagonally adjacent cell are obstacles. The cost of a single agent's path is the total number of timesteps that agent spends away from its goal, and the cost of the entire solution is the sum of all path costs. Two agents must not occupy the same tile at the same time.

Moreover, transitions in which agents pass through each other including diagonal crossing are prohibited even when those agents never occupy the same position during the same timestep such as those in Figure 1.2. However, we do allow rotational transitions like the ones depicted in Figure 1.3.

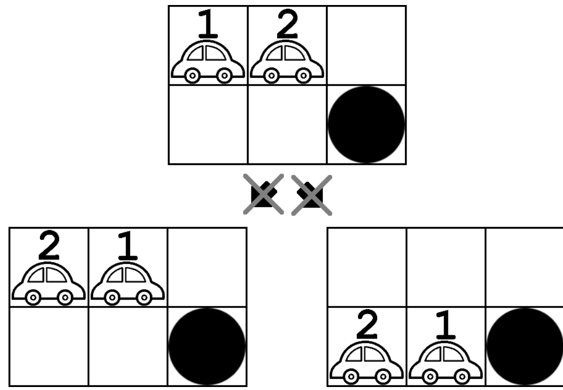

Figure 1.2: Agents cannot pass through each other.

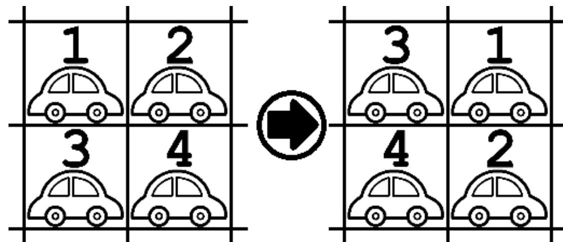

Figure 1.3: Our formulation of the cooperative pathfinding problem allows rotational moves, which result in each agent occupying a previously occupied cell.

1.2 Related Work

Unfortunately, it has been shown to be PSPACE-hard to compute any set of paths that bring each agent to its goal state without conflict in several cooperative pathfinding domains, even those without obstacles [HSS84, HD05]. Furthermore, the standard admissible algorithm for cooperative pathfinding problems has a branching factor that is exponential in the number of agents [Sil05]. For these reasons, existing research on these problems focuses on algorithms for finding good solutions as often as possible rather than developing optimal and complete algorithms. Our algorithm is complete, however, and can find optimal solutions to nontrivial problems, often in milliseconds.

There are two basic approaches to solving cooperative pathfinding problems. In the centralized approach, agents are treated as a composite entity and paths are found for every agent simultaneously. On the other hand, in the decoupled approach, paths are found for each agent one at a time, and information about the paths of other agents is used to ensure that no paths conflict. Centralized approaches typically have the advantage of being complete, but are often intractable for even small numbers of agents, while decoupled approaches are fast, but inherently incomplete.

Silver describes a decoupled algorithm which he claims is widely used in the video game industry, Local Repair A* (LRA*)[Sil05]. In LRA*, a path is computed for each agent independently, and conflicts are not discovered until execution. If the algorithm discovers that following a solution one step further results in a conflict, the algorithm re-plans the path of one of the conflicting agents, disallowing only the move that would immediately result in a conflict. Unfortunately LRA* often results in cycles and deadlock. Silver's solution to this problem, Hierarchical Cooperative A* (HCA*), is another decoupled approach that focuses on reducing the number of future re-plans by creating a reservation table for future timesteps. The algorithm chooses an ordering of agents,

and plans a path for each agent that avoids conflicts with previously computed paths by checking against the reservation table. While the solutions found by HCA* are significantly shorter than those computed by LRA*, the greedy nature of this algorithm still results in suboptimal solutions. Furthermore, in 4.3% of instances, some agents never reach their destinations. Finally, this technique offers no option to spend more computation time to increase either the number of solutions found or solution quality.

Other decoupled attempts establish a direction for every grid position and encourage or require each agent to move in that direction at every step [WB08, JS08a, JS08b]. These methods reduce the chance of computing conflicting paths by creating the analog of traffic laws for the agents to follow. While these methods are effective at reducing the number of incompatible paths, they limit the types of paths that agents can follow, and thus virtually guarantee that the solutions found will be suboptimal. Moreover, as paths are still computed one agent at a time, without regard to the consequences that a choice of path can have on the paths available to other agents, this strategy also suffers from the problem of deadlock, and therefore some agents may never reach their destinations.

Finally, existing centralized approaches such those presented in [Rya08] and [SO96] are complete, but do not aim for low solution costs. Furthermore, the time and memory requirements of these algorithms limit their applicability to problems containing only 100 positions and less than ten agents.

1.3 Problem Complexity

We mentioned in Section 1.2 that complexity results exist for many cooperative pathfinding problems. This section discusses some of these results, as well as results that show that reachability in some cooperative pathfinding problems can be solved in polynomial time. We then discuss the implication of these results on the complexity of the particular

problem formulation described above.

In the warehouseman's problem, axis aligned rectangles of arbitrary width and height are packed inside a rectangular box without overlapping. The problem consists of deciding whether a particular configuration of these rectangles can be transformed into another particular configuration by sliding one rectangle at a time without allowing rectangles to overlap. In [HSS84], the authors show that reachability in the warehouseman's problem is PSPACE-hard.

The proof of hardness in [HSS84] requires many differently sized rectangles. The authors of [HD05] have improved this result using a proof that uses only 2x1 and 1x2 rectangles, and have shown that many related problems are also PSPACE-hard.

The warehouseman's problem is a cooperative pathfinding problem because each rectangle can be thought of as an agent, and since no rectangles may overlap, there are constraints between the moves each agent can make. However, this problem differs from the eight-connected gridworld variant described in Section 1.1 in four ways. First, in the eight-connected gridworld, agents occupy only one tile at a time, while rectangles in the warehouseman's problem have arbitrary dimension. Second, there are no static obstacles in the warehouseman's problem. Third, agents in the eight-connected gridworld move simultaneously, and therefore may make rotational moves that are unavailable to rectangles in the warehouseman's problem. Finally, the eight-connected gridworld variant allows diagonal moves, which are not allowed in the warehouseman's problem.

If rectangles in the warehouseman's problem are required to be 1x1, the reachability problem becomes easy. In [KMS84] the authors present a polynomial-time algorithm for determining whether a configuration of the pebble motion problem can be solved. In the pebble motion problem, agents each occupy a single node in a graph and can successively move to an adjacent empty vertex. The reachability problem asks whether a

particular assignment of agents to nodes can be transformed into another. Since a gridworld is a special case of an arbitrary graph, these results apply to the warehouseman's problem in which rectangles must be 1x1. Solving this problem optimally, however, is NP-hard because it is a generalization of sliding-tile puzzles such as the 15-puzzle, and it is NP-complete to determine whether a particular instance of a sliding tile puzzle can be solved in under a given number of moves[RW90]. Furthermore the authors of [KMS84] prove a polynomial upper bound on the number of moves in a shortest path for a solution, placing the decision variant of the pebble motion problem (deciding whether there exists a path of length k) in NP, and making the decision problem NP-complete.

Although we are unaware of any hardness results for problems in which agents can make *rotational* moves, adding *diagonal* moves does not change the complexity of the problem when obstacles are allowed[1]. By placing obstacles onto the grid in a checkerboard fashion, we can force agents to always make diagonal moves. This is equivalent to a 4-connected grid. Therefore, any hardness results that apply to 4-connected gridworld problems also apply to 8-connected gridworld problems.

We see from a result in [KMS84] that if rotational moves were not allowed, finding an optimal path for an instance of the 8-connected gridworld variant would be NP-hard, and the path would contain a polynomial number of moves. Because the problem is relaxed when rotational moves are allowed, we also see that any instance solvable in the case in which there are no rotational moves remains solvable in our variant, and will require only a polynomial number of moves to solve optimally. It is therefore tempting to place the problem of finding optimal paths in NP, but there are problems that require rotational moves to solve, and it is possible that some of those problems may have super-polynomial length paths. However, it is also possible that the problem of finding optimal

[1] A problem with rotational moves has been studied in [KMS84], but results were only obtained on special case graphs with one agent per node.

Problem Variant	Reachability	Finding any solution	Finding an optimal solution
Only 1x1 rectangles	P	P	NP-hard
Only 2x1 & 1x2 rectangles	PSPACE-hard	PSPACE-hard	PSPACE-hard
Only 1x1 rectangles. Obstacles allowed.	P	?	NP-hard
Only 1x1 rectangles. Rotational moves allowed. Obstacles allowed.	?	?	?

Table 1.1: The complexity of various variants of the warehouseman's problem.

paths in our formulation is easier than finding optimal paths when rotational moves are not allowed. However, it is unlikely that prohibiting rotational moves would have much of an effect on the difficulty of the test instances for our algorithm, and our algorithm could be easily adapted to any of the problems discussed in this section including those without rotational moves.

Table 1.1 summarizes these complexity results. The cooperative pathfinding problem presented in Section 1.1 is as hard as the warehouseman's problem variant in which rectangles must be 1x1, rotational moves are allowed, and obstacles are allowed. It's interesting to note that the authors of [KMS84] were only concerned with solving the reachability problem, and it is possible that their technique could yield a polynomial algorithm for finding a solution to the warehouseman's problem in which rectangles must be 1x1, and obstacles are allowed. If this were to be accomplished there would be a polynomial time algorithm for finding a solution to the problem studied in all of the papers presented in the related work section which are all a special case of the pebble motion problem. However, the solutions found might not have low costs.

Recent publications on grid based pathfinding problems, such as [Sil05], [GCB06],

and [WB08] often cite [HSS84] as containing a PSPACE-hardness result for cooperative pathfinding. However, as their algorithms are only demonstrated in the case in which all rectangles are 1x1, it seems that at least the reachability versions of their problems are solvable in polynomial time. Furthermore, even solving their problems optimally is only NP-hard and unlikely to be PSPACE-hard unless NP=PSPACE. This is because the path (which is of polynomial size) is a proof that a particular instance can be solved in a certain number of moves, and so the decision problem is in NP. Therefore, the result of [HSS84] for the warehouseman's problem does not apply.

CHAPTER 2

Finding Optimal Solutions

2.1 The Standard Admissible Algorithm

The standard admissible algorithm for the eight connected gridworld variant is A* with the following problem representation: A state is an n-tuple of grid locations, one for each of n agents [WB08]. The standard algorithm considers the moves of all agents simultaneously at a timestep, so each state potentially has 9^n legal operators. Each of these legal operators is a solution to the constraint satisfaction problem in which each agent must be assigned a move from $\{N$, NE, E, SE, S, SW, W, NW, and $wait\}$, and there are constraints between sets of legal moves. For example, these legal moves must not lead two agents to pass through each other. In order to generate these operators, a backtracking search is employed to efficiently find all solutions to the CSP at every node expansion. In practice, these CSPs are not difficult, and most of the 9^n combinations of moves are legal. The cost of each operator is the number of agents not executing the $wait$ move while at their goal position. An admissible heuristic, like the one presented in a later section, is always coupled with this algorithm.

While this method can be used to solve some trivial problem instances, each node expansion can add approximately 59k nodes (approximately 1.5MB in our implementation) to the open list when run on an instance with only 5 agents.

In many of the successors of a given node, the majority of agents have moved farther

Algorithm 1 The Standard Algorithm (A*)

1: $open \leftarrow \{(start, h(start))\}$
2: $closed \leftarrow \{\}$
3: **while** $open \neq \{\}$ **do**
4: $\quad x \leftarrow deletemin(open)$
5: $\quad closed \xleftarrow{add} x$
6: $\quad$ **if** x is goal **then**
7: $\quad\quad$ reconstruct the path and return
8: $\quad$ **end if**
9: $\quad$ find each legal successor of S using a backtracking search
10: $\quad$ **for** each legal successor S of x **do**
11: $\quad\quad$ **if** $S \notin closed$ **then**
12: $\quad\quad\quad g(s) \leftarrow g(x) + numAgentsNotStoppedOnGoal(S)$
13: $\quad\quad\quad f(S) \leftarrow h(S) + g(S)$
14: $\quad\quad\quad$ remove any copy S' from open
15: $\quad\quad\quad open \xleftarrow{add} min((S, f(S)), (S', f(S')))$
16: $\quad\quad$ **end if**
17: $\quad$ **end for**
18: **end while**

from their goals, resulting in nodes that have heuristic values as much as n units greater than the heuristic values of their parents. There are typically an exponential number of such unpromising nodes, and A* generates them all, placing each onto the open list. This suggests that the standard algorithm can be improved.

2.2 Operator Decomposition

In the standard algorithm, every operator advances a state by one timestep and potentially changes the position of every agent. We propose a new representation of the state space in which each timestep is divided so that agents are considered one at a time and a state requires n operators to advance to the next timestep. In this representation, a state not only contains a position for every agent, but also contains zero or one move assignments for every agent.

An operator in this representation consists of assigning a move to the next unassigned agent in a fixed order, leaving the moves of the remaining agents to descendant nodes within the same search. The new state representation has a branching factor of 9 rather than 9^n, but because each operator only advances a single agent rather than every agent, the depth of a goal node increases by a factor of n. We call this technique A* with operator decomposition (OD) because the standard operators are decomposed into a series of operators.

Under this representation, there are two conceptually different types of states. We refer to *standard* states as those in which no agent has been assigned a move, and *intermediate* states as those in which at least one agent has been assigned a move. Assigning a move to the last unassigned agent in an intermediate state results in a standard state which is at the next timestep. Note that while these states are conceptually different, A* treats them equivalently, so, in general, the open list will contain both standard and

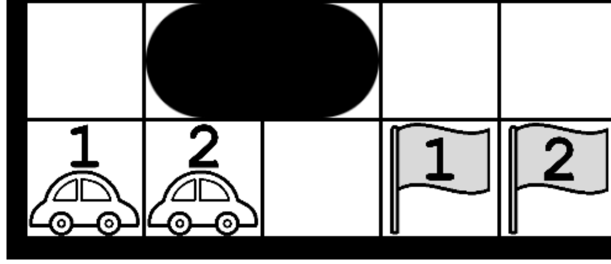

Figure 2.1: An example in which naïve OD would fail.

intermediate nodes, as well as nodes from multiple timesteps.

For a given state, we refer to a *post-move* position as the position that an agent will have after it follows the assigned move, and a *pre-move* position as the position an agent has before it follows an assigned move. We may refer to the positions of agents without assigned moves as either pre-move or post-move positions. Note that within a node, the pre-move positions for every agent are the positions of the agents at a single common timestep.

Note that the $h()$ and $g()$ value for an intermediate node will be different from that of its most recent standard ancestor node. This is because some of the post-move positions may be different than the pre-move positions. Therefore, A* can sometimes avoid expanding intermediate nodes because of cost. The standard algorithm, however, cannot because it uses backtracking search to solve the CSPs which has no concept of cost. It can only prune configurations that are illegal.

Unfortunately, OD will not guarantee optimal solutions if the implementation is naïve. Consider the situation depicted in Figure 2.1. If the algorithm considers agent 1 first, then it might conclude that it has only two legal moves $(N, wait)$. However, the optimal solution requires that agents 1 and 2 both make the move E first. The problem with this naïve version of OD is that it is too strict. It restricts the types of moves an

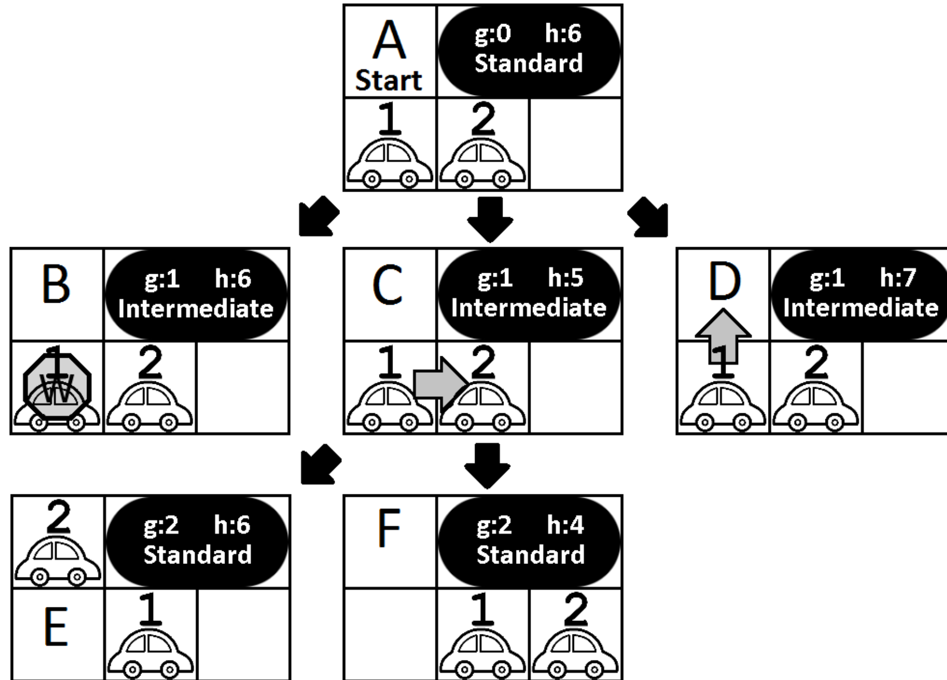

Figure 2.2: The search tree of A* with OD for one timestep. Pre-move positions are shown. Move assignments are marked with arrows.

agent can make on the basis of every other agent's post-move position[1]. The algorithm should require that a new move assigned to an agent be consistent with the existing assignments of moves to agents in a node, but the algorithm should allow agents to move into spaces occupied by agents who are yet to be assigned moves within a node.

Consider again the example of Figure 2.1. When OD is used correctly, A* starts by placing three nodes onto the open list because agent 1 can be assigned N, E, or $wait$. Note that both agents have the same post-move position in the node in which agent 1 is assigned E (node C). This is acceptable because agent 2 has not yet been assigned a move. Then, when that node is chosen for expansion, A* considers assigning agent 2 the move E. The algorithm then checks to see that the move assignment $[1 : E, 2 : E]$

[1]Recall that when the agent has not yet been assigned a move, its post-move position is the same as its pre-move position

is legal (because agent 1 has *already* been assigned a move in this node) and places the resulting child node (node F) onto the open list. The operators $wait$ and W are also checked, but discarded (because the move assignment $[1 : E, 2 : wait]$ results in two agents overlapping, and the move assignment $[1 : E, 2 : W]$ results in two agents passing through each other.

We see in Figure 2.2 that OD generates five nodes during the first timestep in this example, three intermediate nodes, and two standard nodes. On the other hand, the standard algorithm generates eight successor nodes from the start state because $[wait, wait]$, $[wait, E]$, $[wait, NW]$, $[N, wait]$, $[N, E]$, $[N, W]$, $[E, E]$, and $[E, NW]$ are all legal standard operators from the start state. If OD was not pared with a heuristic, then nodes B and D would each generate three standard successors, and we would have generated eight standard nodes, in addition to three intermediate nodes. These additional six standard nodes, are not generated by A* because their cost is too high. For both algorithms, a node in which both agents have made the move E is expanded next.

When coupled with a perfect heuristic and a perfect tie-breaking strategy, A* search generates $b \times d$ nodes where b is the branching factor, and d is the depth of the search. Since the standard algorithm has a branching factor of approximately 9^n and a depth of t (the number of timesteps in the optimal solution), A* search on the standard state space generates approximately $(9^n)t$ nodes when coupled with a perfect heuristic. A* with OD, however, will generate no more than $9nt$ nodes in the same case because its branching factor is reduced to 9, and its depth only increases to $n \times t$. This is an exponential savings with a perfect heuristic.

OD is capable of generating every legal successor of a standard node for any chosen order of agents by assigning the appropriate move to each agent in turn. Therefore, A* with OD is admissible and complete for all choices of agent order due to the admissibil-

ity and completeness of the standard A* algorithm.

Since agents can be considered in any order, it is natural to ask which order should be preferred, but we were unable to find an agent ordering strategy that was worth the overhead. Our implementation simply chooses an arbitrary static ordering of the agents. This order expands slightly more nodes to find a solution than our best agent ordering heuristic, but expands many more nodes per second.

When the standard heuristic evaluation function for a search problem can be applied to intermediate nodes, OD will save both time and space. All cooperative pathfinding problems have this property because the value of the heuristic can change when a single agent changes its position, but other problems with combinatorial branching factors may benefit from operator decomposition as well.

A* detects duplicate nodes in order to avoid redundant node expansions. One way that A* detects duplicate nodes is through the use of a closed list which stores every node that has been expanded. Like many cooperative pathfinding problems, the state space of the eight-connected grid world variant grows only polynomially with depth (although it is exponential in n). Therefore, because the number of nodes in the search tree grows exponentially with depth, duplicate detection is essential.

The intermediate descendants of a standard node form a tree rooted at the standard node because another move assignment cannot be made to an agent until all agents have been assigned moves, and nodes with different move assignments cannot be the same. Because duplicate intermediate nodes would have duplicate standard ancestor nodes, A* will never encounter duplicate intermediate nodes because their standard ancestors would be duplicates, and one would have been pruned. Therefore, it is reasonable to put only standard nodes onto the closed list. Empirically, this modification reduced the size of the closed list by nearly 93% while improving running time in preliminary

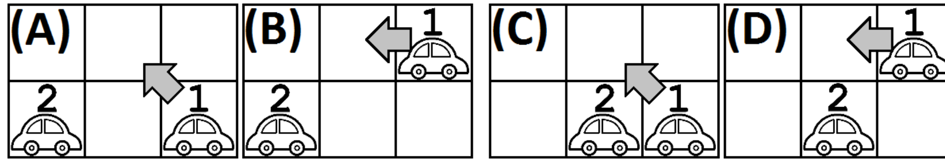

Figure 2.3: Two pairs of similar intermediate states.

experiments.

Alternatively, we can sometimes safely treat nodes that are not equal, but have the same post-move positions, as duplicates. Observe in Figure 2.3 that both (A) and (B) have the same post-move positions. Also, the set of possible moves for agent 2 that do not conflict with agent 1's move is the same in both (A) and (B). These nodes will therefore have the same set of standard descendants, and the algorithm can safely treat them as duplicates. However, agent 2 cannot legally move NE in state (C), but it can in state (D), so treating these states as duplicates would be incorrect even though they have the same post-move positions. This example also shows the importance of storing move assignments in addition to positions in a state. Enabling the algorithm to treat nodes like (A) and (B) as duplicates is well worth the overhead and is used in our experiments.

2.3 The Heuristic

A* makes use of an estimate of the true distance to the goal for every state. This allows A* to intelligently search only a relevant subset of the search space. If this heuristic never overestimates the cost from a state to a goal state, it is called admissible, and A* will return optimal solutions[HNR68]. If the value of the heuristic for a state is never smaller than the heuristic of its parent by more than the cost to move from the parent to the state, the heuristic function called consistent.

One admissible and consistent heuristic for a state in any cooperative pathfinding

problem is the sum of the costs of the optimal single-agent paths for all agents. Conceptually, these distances could be obtained by running a single-agent search for every agent whenever A* generates a node. However, we can simply run a breadth-first search from an agent's goal position to every free grid position to precompute a table of the single agent distances for every agent at every position. During the A* search with OD, the change in heuristic value from a state to a child state is due to the move assignment for a single agent. The change in heuristic value is the difference between the table entry for the agent's post-move position and the table entry for the agent's pre-move position. So the heuristic can be updated using two table lookups. One could also use Reverse Resumable A* (RRA*) [Sil05] to fill in the table values, which improved the efficiency of the breadth-first search option by a small additive constant (less than a millisecond) for each agent in our preliminary experiments.

RRA* was proposed to find heuristic values for single agent search problems, but it is equally applicable to multi-agent search, because the sum of single agent heuristics is admissible for cooperative search. The idea is to run the single agent A* algorithm backward from an agent's goal position for each agent. The heuristic for this single agent reverse search, Chebyshev distance[2][3], is consistent as well as admissible for the single agent search. Because of the consistency property, when a node in this search is placed on the closed list, the g() cost of this node will be the true distance from the agent's position within that node, to the agent's goal position[Nil82].

Our algorithm sets up one RRA* search for each agent. Each search is parameterized by two positions. These positions do not change throughout the search. The first is the agent's start position in cooperative pathfinding problem. This position is made the goal for RRA*. The second is the agent's goal position in the cooperative pathfinding

[2]Chebyshev distance is the analog of Manhattan distance when diagonal moves are allowed.
[3]All moves have unit cost.

problem. This position is made the start state for RRA*. Each RRA* is initially paused, but can be resumed at any time.

When the cooperative pathfinding algorithm needs a heuristic value for a state, it checks each of the RRA* closed lists to see if it contains the right position for the agent being moved during this node expansion. If it does contain the right position for the agent, it uses the g() cost of that node in the closed list as the table entry. If it does not, it tells the RRA* for the agent what position it needs it to find, and resumes it. RRA* continues its search until it expands the needed node, and then pauses. The closed list for this RRA* will then contain the needed value.

Note that RRA* does not stop when its goal node is expanded. The goal node is only needed as a reference to calculate the heuristic (Chebyshev distance) for each newly generated node. After the goal node is expanded, RRA* will continue expanding nodes in best first order, and placing them onto the closed list until the entire search space is explored or the needed value is found, in which case RRA* is paused.

The effectiveness of RRA* depends on the size of the map, and the average distance an agent must travel. The experiments described in the experiments section, which were run on 32x32 maps, reveal that only about 16% of the search space is explored by RRA* on average. When the mapsize is increased to 100x100, this proportion falls to about 10%.

2.4 Partial Expansion

The operator decomposition technique is useful for reducing the number of moves considered by the standard algorithm, but many of the moves considered by the algorithm, even with operator decomposition, are often unnecessary. This section describes a way to further reduce the number of considered moves if operator decomposition is applied.

In [YMI00], the authors outline a technique helpful for reducing the memory requirements of problems with large branching factors. They call their technique partial expansion. Partial expansion is a modification to A*, and works like this: when a node is removed from the open list for expansion, the algorithm generates each possible child and determines the $k_1 + 1$ children with the lowest heuristic values. The algorithm only puts the k_1 best children onto the open list in the hopes that one will lead to the optimal solution. The rest of the children are discarded and the original node is put back onto the open list with the heuristic value of the $(k_1 + 1)^{st}$ child for possible re-expansion. If the node is re-removed from the open list, then the next best $k_2 + 1$ children of that node are determined and the best k_2 of those are put onto the open list and so on. Thus, even if any optimal path must go through a child that does not initially look promising (and therefore is not initially put onto the open list), the parent will be reconsidered, and the children will be put onto the open list after all.

During this process, nodes are often regenerated and their heuristics are recomputed. If memory is the bottleneck, then this is not very important, however for video games and other real time applications, memory is often a secondary consideration. Furthermore, the branching factor of the algorithm with operator decomposition is only 9. Thus it may seem that the idea of partial expansion provides little benefit for a problem such as this, however if we use operator decomposition for this problem, it is possible to precompute which $k + 1$ children will have the lowest heuristic values and store the result for real-time use.

It is not complicated to update RRA* or backward breadth-first search to store which move brings the agent closest to the goal, in addition to the heuristic value. It can even be updated to store the $k + 1$ such moves for an agent. These moves are those that result in having the lowest heuristic value.

Now A* can quickly determine which moves result in the $k+1$ states with the best heuristic values, and can generate only those nodes in time linear in k.

Our preliminary implementation used $k_1 = 1$, $k_2 = 3$, and $k_3 = 5$, and thus only had to store pointers for the best 5 moves using 15 bits total. With this scheme, the ratio of the final size of the open list to the final size of the closed list is reduced from about 7 to less than 3, resulting in about a 2x improvement in runtime, as well as a 2x reduction in memory use.

2.5 Independent Subproblems

Although operator decomposition allows the algorithm to avoid considering a substantial portion of the search space, the resulting algorithm is still exponential in the number of agents. It would be ideal if one could plan a path for each agent independently as in the decoupled approach. This seems to be the intuition behind most inadmissible algorithms developed up to this point [Sil05, JS08b, DS04, WB08]. In order to preserve optimality and completeness, slightly more general idea is considered which involves independently planning optimal paths for disjoint and exhaustive subsets of the agents. If these paths were all found to not interfere with one another, then the total solution obtained would be optimal.

As a motivating example, consider the problem of planning automobile traffic. The problem of planning traffic in Australia is independent of the problem of planning traffic in Japan, and the problem of planning the routes of the only three people driving in the same town at 3am may be independent if they don't cross any of the same intersections at the same times.

In general, we would like to have an algorithm for partitioning the agents into disjoint sets in such a way that the optimal paths computed for each set of agents do not

interfere with the others. This way, we can be sure that the combined solution is optimal.

In order to discover such a partitioning, we developed the simple independence detection (SID) algorithm, which assumes that the paths for all agents will be independent, and cooperatively recomputes the conflicting paths using an optimal algorithm like operator decomposition if this assumption fails. Because the planning time is exponential in the size of the largest group, the largest replan dominates the running time.

Algorithm 2 Simple Independence Detection

1: assign each agent to a singleton group
2: plan a path for each group
3: **repeat**
4: simulate execution of all paths until a conflict occurs
5: merge two conflicting groups into a single group
6: cooperatively plan new group
7: **until** no conflicts occur
8: $solution \leftarrow$ paths of all groups combined
9: **return** $solution$

To simplify the explanations below, we only consider the case in which conflicting groups consist of single agents (the extension to general groups is straightforward).

We can improve SID by making the following observation: Each agent will usually have many optimal paths to its goal, and whether two agents are found to be independent depends on which two paths the algorithm finds initially. Therefore, we can sometimes avoid merging the groups of two agents whose paths conflict by finding another optimal path for either agent. In order to ensure optimality, the new path for an agent must have the same cost as the initial path. We refer to this as the optimality constraint of independence detection. Furthermore, the new path must not conflict with the other agent. To realize these goals, we can give the search algorithm a cost limit and an illegal

move table which is populated just before finding an alternate path.

If two agents a_1 and a_2 conflict, and we decide to replan the path of a_1, we would populate the illegal move table with the transitions a_2 makes when following its current path at every timestep. When the search algorithm expands a node while replanning the path for a_1, it checks the table at that node's timestep to see which possible moves will not conflict with a_2's move for that timestep. If such a path is found, then the conflict between these two agents has been resolved.

During these replans, it is important that the algorithm finds a path that creates the fewest conflicts with other agents. This can be achieved using a table similar to the illegal move table, called the conflict avoidance table, which stores the moves of all other agents for every timestep. A* will typically have a choice between several nodes with the same minimum $f()$ cost to expand first, and tie breaking is usually done in favor of nodes with the lowest $h()$ value to achieve the best performance. Alternatively, each node can keep track of the number of conflict avoidance table violations that have occurred on the path leading up to this node, and break ties in favor of nodes with the fewest violations, only relying on the $h()$ to break ties when nodes have an equal violation count as well as $f()$ cost. This method ensures that the path returned during the replan will lead to the fewest future conflicts of any optimal path.

This is the intuition behind our improved independence detection algorithm (ID), which starts by assigning each agent to its own group. It then finds an initial path for each agent independently. Upon detecting a conflict between the current paths of two agents, the algorithm attempts to find an alternate optimal path for one of the conflicting agents, ensuring that the new path does not conflict with the other agent. If this fails, it tries this process with the other conflicting agent. Finally, if both attempts to find alternate paths fail or two agents that have conflicted before are found to conflict again,

Algorithm 3 Independence Detection

```
1: assign each agent to a group
2: plan a path for each group
3: fill conflict avoidance table with every path
4: repeat
5:     simulate execution of all paths until a conflict between two groups G1 and G2 occurs
6:     if these two groups have not conflicted before then
7:         fill illegal move table with the current paths for G2
8:         find another set of paths with the same cost for G1
9:         if failed to find such a set then
10:            fill illegal move table with the current paths for G1
11:            find another set of paths with the same cost for G2
12:        end if
13:    end if
14:    if failed to find an alternate set of paths for G1 and G2 then
15:        merge G1 and G2 into a single group
16:        cooperatively plan new group
17:    end if
18:    update conflict avoidance table with changes made to paths
19: until no conflicts occur
20: solution ← paths of all groups combined
21: return solution
```

the algorithm merges the conflicting groups and cooperatively plans a path for this new group without a cost limit or illegal move table. All plans are found with a conflict avoidance table which is kept up to date so that every path always results in the fewest number of future conflicts.

These independence detection algorithms do not solve pathfinding problems on their own. They simply call a search algorithm many times on subproblems allowing that algorithm to solve many smaller problems rather than the full problem. The independence detection algorithms can use the standard algorithm or operator decomposition as the underlying search algorithm. ID will merge two groups and run a search algorithm on the combined group if it cannot find nonconflicting paths for both groups. Therefore ID is complete when coupled with a complete search algorithm. Both OD and the standard algorithm are complete.

The independence detection algorithm combines the strengths of both the centralized and decoupled approaches to cooperative pathfinding. The algorithm detects when centralized planning is necessary, and uses decoupled planning otherwise.

2.6 Experiments

Experiments were run on an Intel Core i7 @ 2.8GHz using benchmarks like those proposed in [Sil05]: 32x32 grids were generated with random obstacles (each cell is an obstacle with 20% probability). Each agent was placed in a random unique location with a random unique destination.

The inadmissible algorithm, hierarchical cooperative A* (HCA*), from [Sil05], and admissible combinations of the standard algorithm (S), operator decomposition (OD), and independence detection (ID) were run on the same 10,000 instances with a random number of agents chosen uniformly between 2 and 60. Figure 2.4 shows the running

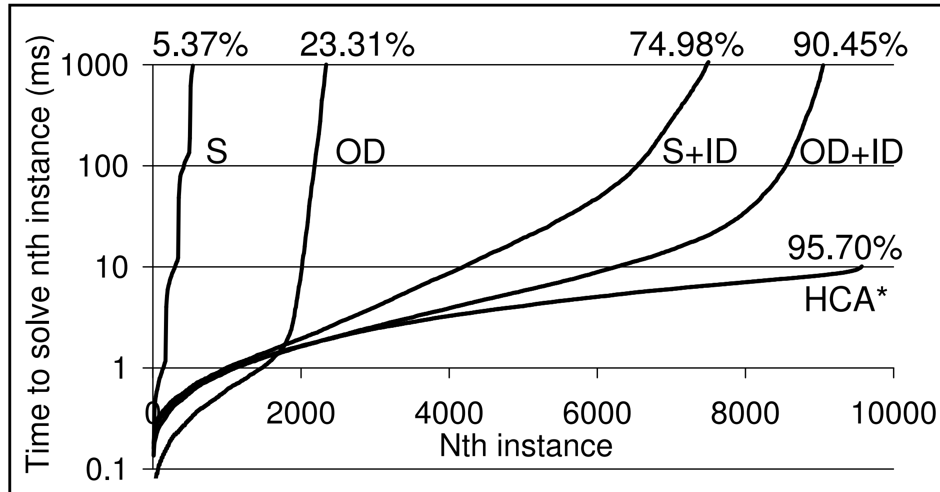

Figure 2.4: Running time of each instance for each algorithm.

time of each algorithm on instances that took less than a second to solve, sorted in ascending order of solution time for each algorithm. The performance of the standard algorithm was highly dependent on the number of agents in the problem, which leads to the stair-step shape observed in the graph. When ID is used, OD not only helps on easy problems, solving them more quickly, but more importantly allows the algorithm to solve the harder problems. OD+ID optimally solved 90.45% of the problems in under a second each. Note that the one second time limit is arbitrary and can be increased in case of hard instances.

Only 69 instances were not solved by either OD+ID or HCA*. Moreover, OD+ID solved 361 out of the 430 problems that HCA* did not solve even though the problems had to be solved optimally for OD+ID. In the problems that were solved by both algorithms and had more than 40 agents, the optimal solutions were an average of 4.4 moves longer than the the heuristic lower bound, while the solutions produced by HCA* were an average of 12.8 moves longer than the heuristic lower bound.

ID partitions a problem instance into subproblems, and the subproblem with the

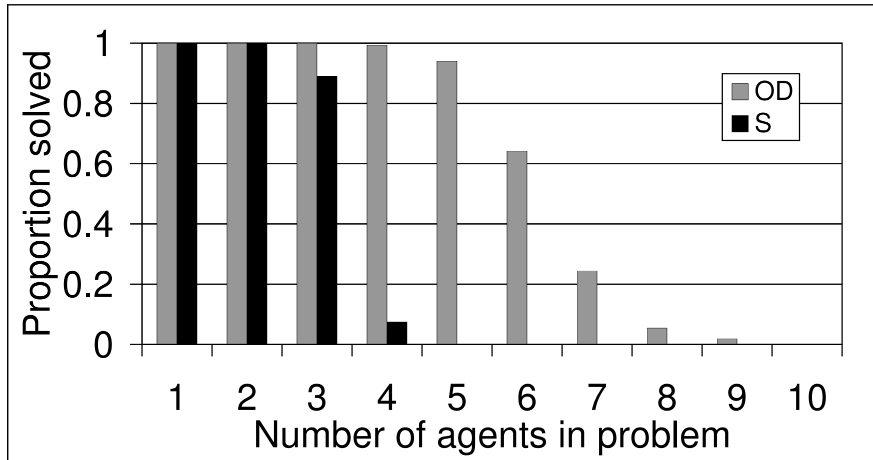

Figure 2.5: Proportion of largest non-independent subproblems given by ID solved within two seconds by OD and S.

highest number of agents typically dominates the computation time for the instance. Experiments show that the strongest indicator of running time on these instances was indeed the number of agents in the largest non-independent subproblem solved by OD. An exponential regression yielded the equation $t = 0.465e^{1.14s}$ where t is time (ms) taken for OD+ID to solve a problem, and s is the number of agents in the largest non-independent subproblem of ID. The regression equation had an R^2 of 0.77. Figure 2.5 shows the results of OD and S on the largest such subproblem solved during each of the above 10,000 instances. In this experiment, OD and S are given two seconds to solve each subproblem regardless of what time was spent solving other subproblems before reaching this subproblem. Since ID could not detect independence within these subproblems, these problems are more challenging for OD and S than random instances with the same number of agents. The ability of the standard algorithm to solve these problems degrades quickly with problem size. In contrast, not only can operator decomposition solve larger problems reliably, but its ability to solve these problems also degrades more slowly with problem size.

CHAPTER 3

Approximation Algorithms

Many applications, especially those in digital entertainment, demand real-time performance and it is often acceptable to sacrifice path quality to satisfy this demand. There have been many algorithms suggested for this purpose, and some are described within this thesis. Unfortunately, we know of no existing algorithms that provide a method for spending more time to achieve better solutions, and all previously proposed complete algorithms are intractable even for small numbers of agents and small graphs. However, the algorithm outlined so far in this thesis can be easily extended to create an approximation that is not only complete, but can also offer a trade-off between solution quality and computation time.

3.1 Complete Approximation Algorithms

There are two constraints within the OD+ID algorithm that force resulting solutions to be optimal. The first is the optimality constraint of independence detection previously mentioned on page 22, which gives operator decomposition a cost limit equal to the optimal cost for a group so that it either produces another nonconflicting optimal solution, or reports that there is none. The other, which is discussed on page 23, constrains A* to expand nodes in order of lowest f() cost rather than in an order which would minimize future conflicts for ID.

In order to produce a fast and complete algorithm, it suffices to drop these two constraints. Dropping the optimality constraint of independence detection allows OD to search for alternate paths of any length during ID rather than just paths that are known to be optimal, and therefore makes it much more likely that an alternate path will be found so that the groups will not be merged. Dropping the second constraint results in fewer future conflicts. This is accomplished by modifying the order in which A* expands nodes on the open list to give priority to nodes with the fewest collision avoidance violations before nodes with the lowest $f()$ cost.

A generalization of this idea results in a parameterized algorithm that offers the time-quality tradeoff mentioned above. We first define a parameter called the maximum group size. We then drop the two above constraints only when merging two groups would result in a group that exceeds the maximum group size. This leads to a spectrum of approximations. If the maximum group size is one, this algorithm will always drop the two constraints, while larger maximum group sizes cause the algorithm to take longer, but produce paths that have lengths closer to optimal.

Note that it is possible that independence detection produces a group larger than the maximum group size. This will happen when no paths of any length exist for two conflicting groups of agents that solve the conflict. In this case, ID will merge the groups as usual to maintain completeness. Fortunately, this does not happen very often as there are an exponential number of possible alternate paths, so there usually exist paths which solve the conflict that the algorithm can find.

Weighted A* (WA*) [Poh70], which is also complete, can theoretically improve the running time of OD+ID further (at the expense of optimality). WA* is the same as A* search, but the $f()$ cost is calculated using a specific inadmissible heuristic constructed by multiplying the admissible heuristic by a constant $w > 1$ thereby allowing A* to

more greedily search for a goal. Theoretically, this would reduce the time needed to find alternate paths for a group of agents. Unfortunately, preliminary experiments showed that WA* did not significantly improve the performance of this algorithm, but did degrade the quality of the approximation.

We call these algorithms the maximum group size algorithms (MGS), and denote an MGS algorithm with a specific maximum group size as MGSn where n is the maximum group size.

3.2 Lower Bound Algorithms

An entirely different approximation than the one mentioned above produces an arbitrarily accurate lower bound on the cost of a solution. When given a cooperative pathfinding problem, this algorithm produces a number that is guaranteed to be smaller than the cost of an optimal solution.

This lower bound can be obtained by ignoring some of the collisions detected by ID. Once again, we define a maximum group size parameter. Collisions that would result in groups larger than the maximum group size are simply ignored. Again, larger maximum group sizes will cause our lower bound algorithm to run for longer before completion, but larger group sizes will also allow the algorithm to explore the interactions between larger groups of agents, which will produce tighter bounds.

The paths obtained during independence detection will contain conflicts in general, so the lower bound algorithm cannot be used to find valid solutions, but perhaps this modification could be useful in providing accurate heuristic evaluation functions for larger cooperative pathfinding problems, or sliding-tile puzzles.

We call these algorithms the lower bound algorithms LB, and denote an LB algo-

rithm with a specific maximum group size as LBn where n is the maximum group size.

3.3 Anytime Approximation Algorithms

Ideally, one would choose how much time to devote to solving a problem, and the algorithm would find the highest quality path it could in the amount of time alloted. Unfortunately, the approximation algorithms presented above do not solve the problem in this way. Once given a maximum group size, they spend an unspecified amount of time to return a result, and there are no meaningful intermediate results.

One way that the approximation algorithms could be adapted for anytime use is inspired by iterative deepening[Kor85]. First, the algorithm uses a maximum group size of 1. Once it has found a solution and if time still remains, it starts over with a maximum group size of 2. It continues to incrementally increase the maximum group size until the time has run out. When the time has run out, it returns the highest quality solution it encountered on any of its runs. Since the running time is roughly exponential in the maximum group size, this adds a constant amount of overhead for incomplete and unused iterations.

While this approach is a useful anytime algorithm, all of the work done for previous iterations is lost completely. Since the algorithm seems to run in time exponential in the maximum group size, this adds only a constant amount of overhead. The algorithm can be improved, however, by making use of the information obtained during previous iterations. Furthermore, by updating the optimal path more frequently, the overhead of incomplete iterations is minimized.

We start by labeling each group with a lower bound on the cost of any paths for that group. The initial label for each singleton group is simply the initial heuristic value for that agent. When groups are merged, the algorithm must cooperatively plan a path for

the new group. If the size of that group is less than or equal to the current maximum group size, then the cost of the new path will be optimal, and so the lower bound can be updated. Otherwise, the lower bound for the group will be the sum of the lower bounds for the two newly merged groups.

We will still iteratively increase the maximum group size starting from 1, but now we will keep the groupings and paths of the previous iteration along with the lower bounds for each group. Every time we increase the maximum group size, we can easily tell which groups may not have optimal paths by simply comparing the lower bound for each group with the cost of the current set of paths for that group. We can then start finding new and optimal paths for each group that might not currently have an optimal path, but is larger than the maximum group size. After a new optimal path is found for a group, we can update the lower bound for that group. We can also solve any conflicts that this new optimal path creates. Whenever we have nonconflicting paths for every agent, we can update the highest quality solution if our new solution is indeed of higher quality.

This method allows some information obtained in previous iterations to be useful for subsequent iterations, and improves the quality of paths found over the pure iterative deepening approach. This algorithm will produce optimal solutions if allowed to run indefinitely because every time the maximum group size is increased, we have the invariant that every group with a size less than the maximum group size has locally optimal paths for every agent in the group. Eventually, the maximum group size will exceed the size of the largest group, and all groups will have locally optimal paths without interfering.

We refer to this algorithm as the Anytime Approximation (AA) Algorithm.

3.4 Experiments

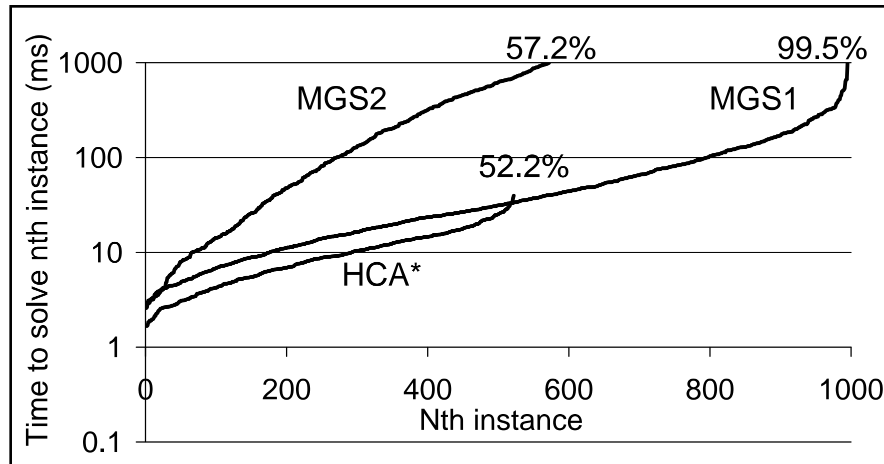

Figure 3.1: Running time of each instance and percent of problems solved for each approximation algorithm on 1000 very large problems.

Once again, experiments were run on an Intel Core i7 @ 2.8GHz, and once again test instances are randomly generated.

Figure 3.1 shows the running time of three approximation algorithms on 1000 randomly generated instances with between 20 and 250 agents each. Two maximum group size algorithms, MGS1 and MGS2, and HCA* were each run on these problems. We see that MGS1, our coarsest approximation, can solve nearly all of the problems in under a second each, while HCA* can only solve slightly over half of the problems. Even the much tighter MGS2 algorithm can solve more problems than HCA*.

In order to illustrate the quality of the approximations produced by our approximation algorithms, 500 instances with between 20 and 100 agents were randomly generated, and the optimal algorithm OD+ID was allowed 40 seconds to solve each problem. Only 289 of these 500 problems were solved by OD+ID. These 289 solved instances are used to compare our approximation algorithms against known optimal solutions. We refer to these problems as our reference problems.

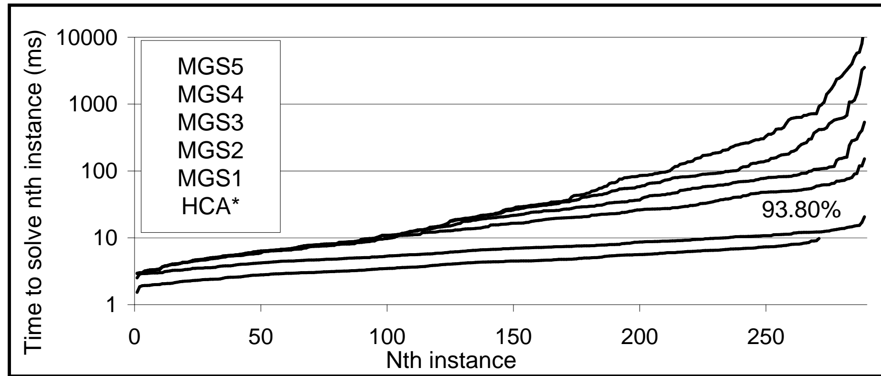

Figure 3.2: Running time of each instance for each maximum group size approximation algorithm. HCA* only solved 93.8% of the problems.

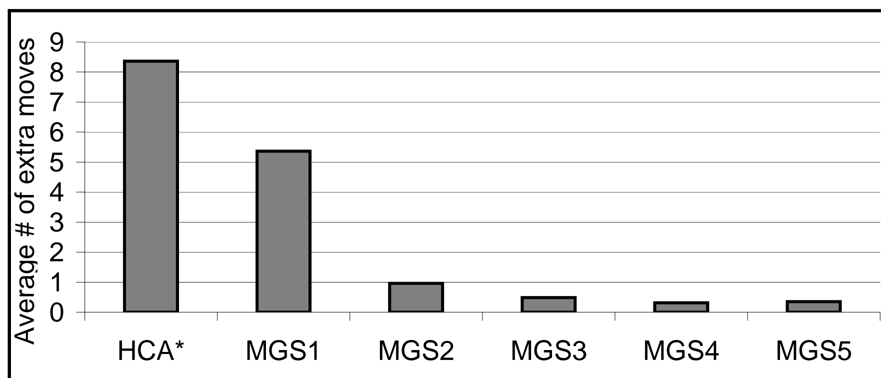

Figure 3.3: Solution quality for our MGS algorithms (lower is better).

Five levels of the MDS algorithm and HCA* were run on our 289 reference problems. The running time is plotted in Figure 3.2. The solution quality is plotted in Figure 3.3. Although HCA* ran faster than our MDS algorithms, it did not solve all of the reference problems. MGS1, however, easily solved every problem in under 20ms each. Furthermore, the solution quality of HCA* was worse than the solution quality of every MGS algorithm. Finally, the solutions found by the MGS2 algorithm were all found in under 150ms each, and had an average of less than one extra move above the optimum.

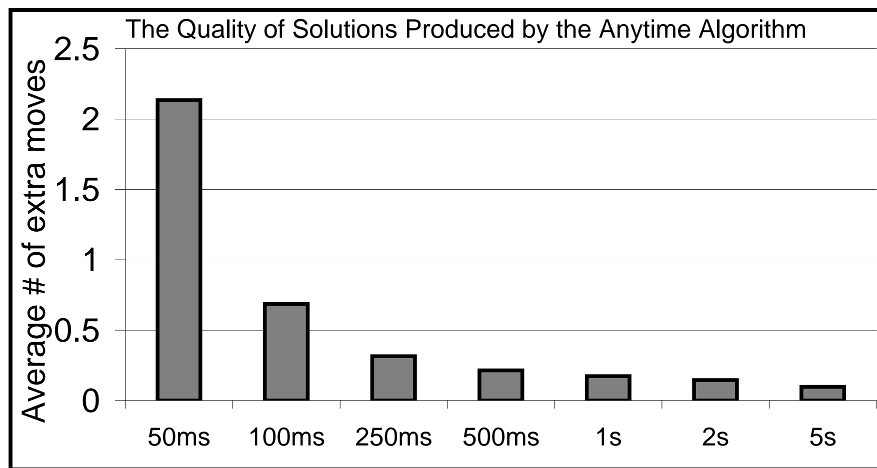

Figure 3.4: Solution quality vs. time for our anytime algorithm (lower is better).

Figure 3.4 shows the average number of extra moves for the solutions produced by the anytime approximation algorithm when run for times ranging from 50ms to 5s. Note that the algorithm performs better on average than MGS2 when run for 100ms, even though MGS2 can sometimes run for nearly 150ms. We see that the paths generated by the anytime algorithm rapidly approach the optimal paths, and the algorithm can generate high quality paths in an amount of time suitable for applications in digital entertainment.

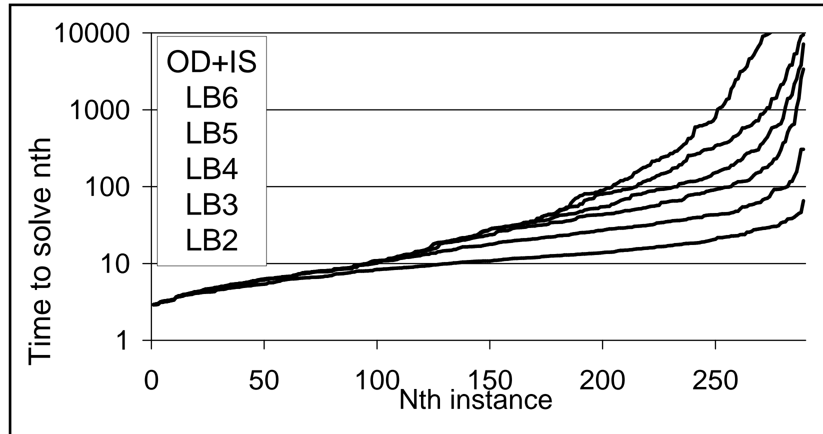

Figure 3.5: Running time of each instance for each lower bound algorithm.

Figure 3.5 shows the running time of several lower bound algorithms together with our optimal algorithm on our 289 reference problems as described above. We see that the lower the maximum group size is, the more quickly the algorithm runs. The most coarse approximation can be obtained in less than 100ms for every problem.

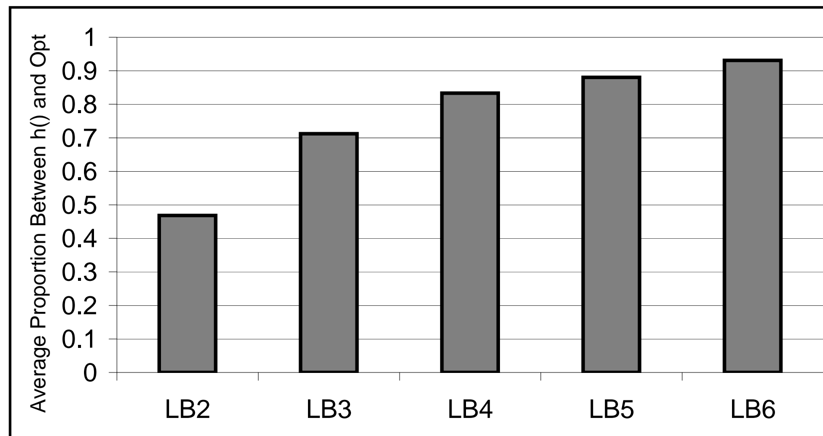

Figure 3.6: The average solution quality for each lower bound algorithm (higher is better).

Figure 3.6 shows the quality of each approximation as an average proportion between the heuristic lower bound for a problem, and the optimal solution cost. The coarsest approximation improves the lower bound by nearly 50% on average for each reference instance, and runs quite quickly. The LB3 algorithm runs almost as quickly, but improves the quality of the approximation to over 70%.

CHAPTER 4

Conclusion

This thesis describes the general problem of cooperative pathfinding. We discussed the pitfalls of modern incomplete and inadmissible algorithms for the cooperative pathfinding problem, and identified two improvements to the standard admissible algorithm: Operator decomposition, which was used to reduce the branching factor of the problem, was effective at allowing the algorithm to consider only a small fraction of the children of each node, and independence detection, which often allowed the paths of groups of agents to be computed independently, without sacrificing optimality. The largest improvement was gained by exploiting this independence. On the other hand, operator decomposition provided a significant benefit even when no independence was present in the problem, and scaled more gracefully to larger problems than the standard admissible algorithm. We also began characterizing the quality of optimal solutions, and the running time of our optimal algorithm. Finally, we showed that this optimal approach suggested complete approximation algorithms with time-quality trade offs, and that these algorithms perform far better than existing approximation algorithms, and are capable of quickly finding solutions to problems with as many as 250 agents on a 32x32 grid.

While both independence detection and operator decomposition are applicable to cooperative pathfinding problems in general, we presented a specific example of cooperative pathfinding which was modeled after typical video game designs, and used this

problem to test the presented algorithms as well as to clarify the algorithm descriptions. The results of the tests show that the standard admissible algorithm can be substantially improved. Furthermore, we showed that when the techniques are used in combination, the algorithm can be made practical.

REFERENCES

[DS04] Kurt Dresner and Peter Stone. "Multiagent Traffic Management: A Reservation-Based Intersection Control Mechanism." In *In The Third International Joint Conference on Autonomous Agents and Multiagent Systems*, pp. 530–537, 2004.

[GCB06] Alborz Geramifard, Pirooz Chubak, and Vadim Bulitko. "Biased Cost Pathfinding." In John E. Laird and Jonathan Schaeffer, editors, *AIIDE*, pp. 112–114. The AAAI Press, 2006.

[HD05] Robert A. Hearn and Erik D. Demaine. "PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation." *Theor. Comput. Sci.*, **343**(1-2):72–96, 2005.

[HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics*, **4**(2):100–107, February 1968.

[HSS84] J.E. Hopcroft, J.T. Schwartz, and M. Sharir. "On the Complexity of Motion Planning for Multiple Independent Objects; PSPACE-Hardness of the Warehouseman's Problem." *The International Journal of Robotics Research*, **3**(4):76–88, 1984.

[JS08a] M. Renee Jansen and Nathan R. Sturtevant. "Direction Maps for Cooperative Pathfinding." In Christian Darken and Michael Mateas, editors, *AIIDE*. The AAAI Press, 2008.

[JS08b] Renee Jansen and Nathan Sturtevant. "A new approach to cooperative pathfinding." In *AAMAS*, pp. 1401–1404, Richland, SC, 2008. International Foundation for Autonomous Agents and Multiagent Systems.

[KMS84] D. Kornhauser, G. Miller, and P. Spirakis. "Coordinating Pebble Motion On Graphs, The Diameter Of Permutation Groups, And Applications." *Foundations of Computer Science, Annual IEEE Symposium on*, **0**:241–250, 1984.

[Kor85] Richard E. Korf. "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search." *Artificial Intelligence*, **27**:97–109, 1985.

[Nil82] Nils J. Nilsson. *Principles of Artificial Intelligence*. Springer, 1982.

[Poh70] Ira Pohl. "Heuristic Search Viewed as Path Finding in a Graph." *Artif. Intell.*, **1**(3):193–204, 1970.

[RW90] Daniel Ratner and Manfred Warmuth. "The (n2-1)-puzzle and related relocation problems." *J. Symb. Comput.*, **10**(2):111–137, 1990.

[Rya08] Malcolm R. K. Ryan. "Exploiting subgraph structure in multi-robot path planning." *J. Artif. Int. Res.*, **31**(1):497–542, 2008.

[Sil05] David Silver. "Cooperative Pathfinding." In R. Michael Young and John E. Laird, editors, *AIIDE*, pp. 117–122. AAAI Press, 2005.

[SO96] P. Svestka and M. H. Overmars. "Coordinated path planning for multiple robots." Technical Report UU-CS-1996-43, Department of Information and Computing Sciences, Utrecht University, 1996.

[Sta10] Trevor Standley. "Finding Optimal Solutions to Cooperative Pathfinding Problems." In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI)*, 2010. To appear.

[WB08] Ko-Hsin C. Wang and Adi Botea. "Fast and Memory-Efficient Multi-Agent Pathfinding." *AAAI 2008*, 2008.

[WB09] K.-H. C. Wang and A. Botea. "Tractable Multi-Agent Path Planning on Grid Maps." In *Proceedings of the International Joint Conference on Artificial Intelligence IJCAI-09*, pp. 1870–1875, Pasadena, CA, USA, 2009.

[YMI00] Takayuki Yoshizumi, Teruhisa Miura, and Toru Ishida. "A* with Partial Expansion for Large Branching Factor Problems." In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pp. 923–929. AAAI Press / The MIT Press, 2000.