1-1-2017

Abstraction Hierarchies for Multi-Agent Pathfinding

Aaron R. Kraft
University of Denver

Abstraction Hierarchies for Multi-Agent Pathfinding

A Thesis

Presented to

the Faculty of the Daniel Felix Ritchie School of Engineering and Computer Science

University of Denver

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Aaron R. Kraft

March 2017

Advisor: Prof. Nathan Sturtevant

Author: Aaron R. Kraft
Title: Abstraction Hierarchies for Multi-Agent Pathfinding
Advisor: Prof. Nathan Sturtevant
Degree Date: March 2017

Abstract

Multi-Agent Pathfinding is an NP-Complete search problem with a branching factor that is exponential in the number of agents. Because of this exponential feature, it can be difficult to solve optimally using traditional search techniques, even for relatively small problems. Many recent optimal solvers have attempted to reduce the complexity of the problem by resolving the conflicts between agent paths separately. Very little of this research has focused on creating quality heuristics to help solve the problem. In this thesis, we create heuristics using sub-problems created by removing agents from a complete problem instance. We combine this with the Independence Detection technique for solving the problem by separating agents into independent (non-conflicting) groups. The results showed moderate improvements in state expansions and computation time in problems with a large number of conflicting agents.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Imagine you're retrieving an item from the storeroom of a crowded shop. The items are held on shelves and other people are also trying to obtain and return items to the shelves, similar to the situation depicted in Figure 1.1. While you navigate the room, you will have to choose a path that avoids the other people, but unlike the shelves, the people do not stand still. You could simply wait for the others to pass, but this would waste time, a resource we have in limited supply. As people, we use a set of strategies for handling these types of situations; for machines we treat them as search problems.

Consider, then, the set of robots that move shelves through a warehouse such as those built by Amazon Robotics for use in Amazon warehouses. The robots must find a path from the current location of a chosen shelf unit to its eventual destination. The warehouse can be broken down into a grid of space available for moving and for storage, and each robot must avoid other robots in order to correctly reach its destination. What these problems have in common, in addition to pathfinding, is conflict. Even though the environment may not change, there are still locations that must be avoided at a given time because of the action of the other agents, in this case people or robots. This is the essence of Multi-Agent Pathfinding (MAPF).

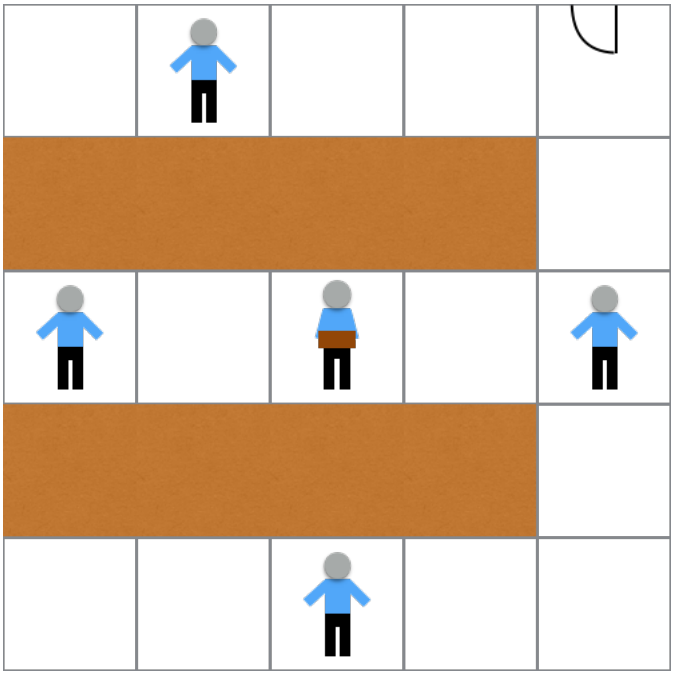

Figure 1.1: Example of a problem involving choosing a path around other people.

Multi-Agent Pathfinding is a problem that has gained a lot of recent research attention due to its computational challenges and many practical applications. It can be used to solve a number of real world problems involving robotics, traffic and logistics. Because the problem requires a rigorous examination of the many path possibilities, it can be difficult to find the least costly solution (in terms of movement steps) quickly. A number of new techniques have been developed in recent years to find these optimal solutions faster, such as the M* algorithm [19], the ICTS algorithm [15] or a reduction to other problems such as Boolean Satisfiability [17] which can then be solved using solvers tuned for that problem. This thesis will examine techniques that have as yet seen limited consideration in the published literature on the problem.

Multi-Agent Pathfinding considers the problem of moving a set of "agents" each from a start position to a goal position in a shared space. An agent can be thought

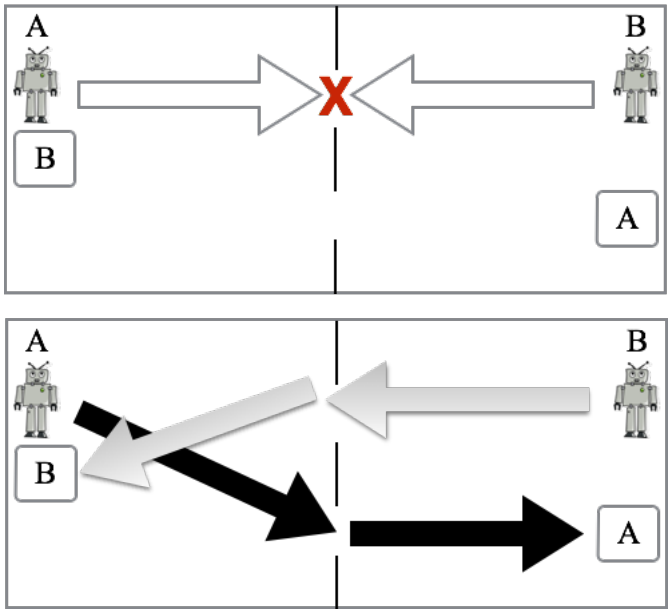

Figure 1.2: Example of a conflict that can be resolved without waiting.

of as a robot or vehicle that is able to move independently to get from one location to another. These agents must each find a path to their goal location, and need to avoid conflicts with other agents by avoiding locations other agents are occupying. One could imagine the problem as a set of robots in a maze: finding a path for a single robot through the maze is a straightforward enough problem, but the problem becomes more interesting by adding other agents that may need to move through the same areas. Choosing how to resolve the conflicts forms the crux of the problem: if two robots are about to enter the same location at the same time, it may be possible for one to sidestep the other, leaving both to finish their route in optimal time, or it may be necessary for one to wait while the other moves through the conflicting area, leading to a loss of time and/or energy.

Consider Figure 1.2. Robot A and B both have to pass through the divider in the center of the map to get to their goal locations. If both robots attempt to pass through the hole in the top part of the divider, they will conflict, and one of them

will have to wait. However, if robot A goes through the hole in the bottom of the divider instead, it will be out of the way in time for robot B to pass, and there will be no waiting.

One of the considerations that must be made with regard to solving this problem is whether or not to attempt to find an optimal solution. Because of the nature of the conflicts that need to be resolved, solving the problem optimally can take an enormous amount of computation time and memory. This means that in some applications it is impractical to try to find the optimal solution. On the other hand, if the process of executing the plan is slow or or otherwise very costly (in terms of money, fuel or energy, for instance), it will make the time invested into creating an optimal plan much more worthwhile. This thesis is concerned only with optimal solutions to the problem.

Solutions to this problem can be used in a number of different applications. Automated traffic control systems could potentially use this to direct traffic through intersections quickly and reliably [3]. In this case, individual vehicles would be the agents with start and goal locations derived from the current location and eventual destination of the vehicle. Any algorithm for this type of problem would need to avoid conflicts and find paths passengers would find reasonable. Furthermore, with the use of centralized pathfinding capabilities, an optimal or near-optimal solution could be used to route vehicles in a way that saves many of them time and fuel by exploiting the ability to choose routes for each agent with the others in mind.

Another possible application is the automated handling of freight. Ports often use large, mobile cranes to move freight around a stockyard. If multiple cranes are set up to work in the same area, they would have to move around without conflict using a pathfinding system that solved instances of this problem. The potential to save time, money and energy is again readily apparent when the problem is solved with a highly efficient or optimal path.

A final increasingly relevant example of the possibilities for technology based on efficient solvers to these problems is the routing of drone aircraft [20]. If there are multiple drone aircrafts moving along routes at the same altitude, path planning becomes a potent tool for avoiding collisions and saving time. The more each drone is able to anticipate and avoid possible collisions with other drones, the quicker and more efficient the entire system can be. This could be facilitated by information exchange, policies for movement in shared spaces or some form of centralized path planning.

One simple way to solve this problem is to generate paths for each agent individually and have the agents follow these paths until they reach a conflict. Once the conflict is reached, either look for an alternate path for one of the conflicting agents, or simply wait until the conflicting area is clear and proceed down the original path as planned. The advantage of this approach is its simplicity and computational speed. Only one path must be generated for each agent, and the mechanics of resolving the conflicts can be relatively simple. The problem with this approach, however, is that the resulting path is not guaranteed to be optimal, and may in fact force agents to wait needlessly when another, quicker, route was available. Additionally, care must be taken to guarantee that a solution can be generated at all.

If instead we wish to solve this problem optimally, all agents' paths must be coordinated to limit the amount of waiting and diversions. To do this, the solver may have to consider multiple possible paths for each agent, and avoid any conflicts. This can be accomplished with a straightforward search approach, which involves branching out through all of the possible paths for all agents and choosing the best one. This method can use well-known search techniques that have been finely tuned and can guarantee optimality. The problem with this approach is the enormous branching factor of the problem: given a state, the number of possible actions that can be performed is exponential in the number of agents. For example, assuming

each agent can move up, down, left or right, each agent has 5 actions (when including wait as an action) available at any given time, meaning that the number of total combinations of actions is 5^n, where n is the number of agents. This exponential increase is shown in Table 1.1 and Table 1.2. Table 1.1 shows the number of actions that are available for a single step of the problem, assuming each agent has 5 available actions. This result is the branching factor of the search tree (the number of branches from each node at each level of the tree). Table 1.2 shows the number of possible states for the given number of agents on a grid with no obstacles. This is the maximum number of individual configurations of the problem instance that may need to be evaluated to find the optimal solution. The data in these tables show that numbers quickly become too large to calculate in a reasonable amount of time for even a relatively small problem space and few agents. This means traditional search techniques may falter over many instances of this problem as a result of the branching factor and size of the state space.

Actions for MAPF (Single Move)		
Actions	Agents	Joint Actions
5	1	5
5	2	25
5	3	125
5	4	625
5	5	3,125
5	10	9,765,625

Table 1.1: Number of MAPF Actions for a given number of agents.

Consequently, some solvers focus on the conflicts in the path [12] [13]. Assuming that the size of the problem map is large compared to the number of agents, the likelihood of conflicts between the agents' paths is low, and when conflicts are found, the available space will allow new paths to easily be created to resolve them. Thus, a solver can find paths for the agents individually, and then attempt to resolve conflicts with a new search one at a time if they occur. This technique handles

States for MAPF		
Agents	Map Size	Number of States
5	3x3	15,120
5	4x4	524,160
5	5x5	6,375,600
5	6x6	45,239,040
6	6x6	1,402,410,240
7	6x6	42,072,307,200
8	6x6	1,220,096,908,880
9	6x6	34,162,713,446,400
10	6x6	922,393,263,052,800

Table 1.2: Number of MAPF States for a given number of agents and map size.

sparse examples well, but will break down as more agents are added, and the conflicts between agents become more likely and more difficult to resolve. The result in cases like these would be an exhaustive search of the search space, with the large branching factor for some problems remaining an issue.

Finally, another technique to use would be to generate several paths for each agent, and combine them, attempting to find a combination that avoids conflicts. This technique avoids the large, expensive, multi-agent search at the possible cost of having to generate a large number of possible paths and evaluate the many combinations thereof. The solutions can be evaluated in order of cost, allowing the guarantee of optimality and reducing the number of paths that must be generated at once.

The focus of this thesis is on creating better cost estimates for performing searches on this particular problem. These cost estimates will allow the search to focus on the actions that are most likely to lead to solutions. This, in turn, will guide the search toward the states most likely to be on the optimal path. In our case, we will create an abstraction to help solve the original problem by removing agents from the problem. Because the main difficulty of MAPF problems is the exponential blow up of the state space and branching factor correlated to the number of agents, by removing agents from the problem we end up with a much easier problem to

solve that can potentially be used to inform the progress of the full search. This, in turn can be combined with the techniques above to potentially reduce the overall search time.

1.1 Summary of Contributions

This thesis will show how to create better cost estimates by breaking MAPF problems into smaller problems by removing agents from the problem. It will discuss the use of hierarchies of abstractions created by removing agents from the problem. It will show how to use the hierarchy of searches already created by the Independence Detection algorithm [16] for creating cost estimates. It will explain why the Switchback algorithm [9] does not improve performance for this problem. Finally, it will show experimental results to validate these ideas.

1.2 Thesis Structure

This thesis begins with the background information required to solve MAPF problems using A* and heuristics, as well as the hierarchical A* algorithm on which our work is based. It will then provide a summary of the most recent work to solve the problem optimally. Finally, we will discuss the work performed to create new heuristics for MAPF and the experimental results obtained.

Chapter 2

Problem Definition and Alternate Approaches

The work performed in this thesis will rely heavily on existing combinatorial search techniques. These techniques are useful for search problems in general, and can be used to find solutions for many problems, including MAPF. This thesis will outline how these techniques can be used to search for solutions to MAPF and how to improve performance with our additions.

2.1 Problem Formulation

The Multi-Agent Pathfinding problem (MAPF) is formalized as a simple graph $G(V, E)$ and a list of agents $a_0(s_0, g_0)...a_n(s_n, g_n)$, where $s_i \in V$ is the start node for agent a_i and $g_i \in V$ is the goal node for agent a_i. A solution to the problem is a list of paths $P_0...P_n$ each of which takes the corresponding agent from its start location to its goal location, where P_i consists of a list of nodes $(s_i, n_1, n_2...g_i), s_i, g_i, n_t \in V$, that agent a_i passes through. Each pair of consecutive nodes along the path must be connected by an edge, $e_t \in E$. As an additional restriction, no agent may have

in its path the same node or edge as another agent at the same time step; therefore $P_i(n_t) \neq P_j(n_t)$ and $P_i(e_t) \neq P_j(e_t)$, where $e_t \in E$ and e_t is the edge that connects n_t and n_{t+1}.

Each path is given a cost, $c_0...c_n$, which is calculated to be the sum of the cost of the actions taken by the agent along its path to reach the goal. The allowed actions are a move action that allows the agent to travel from one vertex $\in V$ to another along an edge $\in E$, and a wait action that allows the agent to remain in its current location. The cost of a move action is m, and the cost of a wait action is w. The cost of a solution is the sum of the costs of all of the individual paths that comprise the solution.

An important consideration about the state space of this problem is the inclusion of time as a dimension. Unlike a single agent pathfinding problem, not only is an agent's position important, but the time is important. This is because the agent's actions must be synchronized in order to optimally solve this problem; moving only one agent at any given time step is unlikely to result in an optimal solution, and properly navigating around another agent requires the knowledge of not only where that agent is, but also when. Since time can only move forward from the start state, that means there is an implied graph with directed edges from that state.

The object is to find an optimal solution. An optimal solution is a list of paths whose sum of costs is equal to the minimum of the sum of costs of all solutions that satisfy the conditions above.

2.1.1 Variants

Several variants of this problem exist that vary based on the cost function in use. The Sum of Individual Costs variant assigns a cost of one to any individual agent action (including wait actions) that occur before the agent reaches its goal for the final time. Any wait actions that occur after the agent reaches its goal are given

a cost of zero, unless a move action is required at a later time step. The makespan variation of the problem assigns a cost of one to each multi-agent action, defined as an action consisting of exactly one single-agent action per agent each performed in a single time step. Thus, the cost is simply the length of the longest path for any of the agents out of the set of individual paths that gets all of the agents to their goals.

The final variant is the fixed cost variant that is used in the experiments in this thesis. The move and wait actions are each assigned a cost. These costs can be set to different values depending on the relative utility of each; for instance, a low wait cost representing the energy required to keep an agent in place versus the cost of moving. If the move and wait actions are the same, this variant becomes effectively identical to the makespan variant of the problem, where the cost of a path is a multiple of the makespan path cost.

Solvers for this problem fall broadly into two categories: optimal and suboptimal. Suboptimal solvers typically decouple the agents and solve them individually. Optimal solvers must solve the problem for all agents combined in order to prove the universal optimality for the obtained solution. This thesis discusses optimal solutions exclusively.

2.2 Dijkstra's Algorithm

Search problems can be solved optimally using a best-first search [2] method (Dijkstra's algorithm), which is described in pseudo-code in Algorithm 1. The first step in this search is to place the start state into a priority queue data structure that sorts by g-cost (the current lowest cost found to reach a given state), as seen on line 2. During each step of the algorithm, the state at the head of the priority queue is removed and all successor states (states which can be reached by applying one legal action) are generated and placed into the queue, with the g-cost of the

new state calculated by adding the g-cost of the parent state to the cost of the action taken to generate the state (lines 3-13). If a state that already exists in the queue is discovered, the g-cost of that state is updated to the smaller of the newly discovered g-cost or the current g-cost in the queue (line 9). The process of removing a state from the queue, generating successors and updating the cost is referred to as "expanding" the state. This continues until the goal state is removed from the queue, at which point the g-cost of the goal state is the optimal cost, and the path can be reconstructed.

Algorithm 1 Dijkstra's Algorithm (Best-first Search)

```
1:  function BestFirst(start, goal)
2:      add start to priority queue, q with g-cost of 0
3:      repeat
4:          s ← head of q
5:          for all actions for s do
6:              s' ← result of applying action to s
7:              g ← (g-cost of s) + (cost of action)
8:              if s' is in q then
9:                  update g of s' in q with Min(g, g of s' in q)
10:             else
11:                 add s' to q with cost g
12:             end if
13:         end for
14:     until s = goal
15: end function
```

At the end of each step, all states that have been expanded have been closed at their optimal g-cost. Since the priority queue is ordered by g-cost, this means the search will expand all states reachable at each g-cost before examining states at a higher g-cost. As a result, if the goal state is the state furthest from the start state in the search space, it is expanded last. Therefore, it is possible to expand all N states before finding the goal. The cost of maintaining the priority queue depends on the implementation: using a heap, this cost is $O(N \log(N))$ since each state expansion will require a $log(m)$ ExtractMin operation, where m is the number

of states in the queue, and m is bounded by N. Using a bucketed open list, the time spent maintaining the open list can be reduced to a constant, which will result in a worst case time complexity of $O(N)$. The worst-case memory cost of the algorithm is also $O(N)$, since all states that have been generated must be tracked.

2.3 A* and Heuristics

The guarantee of optimality for best-first search requires that an optimal path be constructed from the start state to all states which lie on the optimal path between the start and the goal. This is to say that, if the optimal path from state A to state C passes through state B, then an optimal path from A to B must be found before the optimal path from A to C can be found. Since it is possible that all states in the search space are on the optimal path (imagine a long, winding road with no exits), it is possible that every state must be expanded to find the goal. As a result, there is no way to improve the worst-case of this algorithm; any search algorithm may need to expand every state in the state space in order to find the goal state. This worst case is unlikely, however, and in most problems Dijkstra's Algorithm will not expand all states. The problem, instead, is selectivity. Because Dijkstra's algorithm finds an optimal path to all states in ascending order of path cost, it will typically expand states that are a long way away from the optimal path to the goal. This is a considerable departure from the best case, where only the states on the optimal path are expanded. To do this, however, more information about the problem must be obtained and incorporated in the search. In the A* algorithm, this information comes in the form of a heuristic [6].

A heuristic, in A* search, is an estimate for the cost of the optimal path to the goal from the current state. The heuristic cost (referred to as the h-cost) can be added to the g-cost of a state to provide an estimate for the total cost of the optimal path through the current state from the start state to the goal state (called the f-

cost). The states are then ordered by f-cost. See lines 2 and 8 in the pseudo-code below for an example of how the heuristic is incorporated into the search.

Algorithm 2 A* Search

```
1: function A*(start, goal)
2:     add start to priority queue, q with f-cost of h(start)
3:     repeat
4:         s ← head of q
5:         for all actions for s do
6:             s' ← result of applying action to s
7:             g ← (g-cost of s) + (cost of action)
8:             f ← g+ h(s')
9:             if s' is in q then
10:                update f of s' in q with MIN(f, f of s' in q)
11:            else
12:                add s' to q with cost f
13:            end if
14:        end for
15:    until s = goal
16: end function
```

In order to maintain the guarantee of optimality, the heuristic that is calculated for each state must be less than or equal to the actual cost of the optimal path from that state to the goal state. This property is called admissibility. Let $h^*(s)$ refer to the actual optimal path cost from state s to the goal. A heuristic, $h(s)$, is admissible if and only if $h(s) \leq h^*(s)$. An admissible heuristic for any search problem is a zero heuristic, which returns zero for any search state. With this heuristic, the search is identical to the uninformed Dijkstra's algorithm.

Another property of heuristics is consistency. Consistent heuristics obey the triangle inequality; this means the change in h-cost between two states must be less than or equal to the sum of the difference in h-costs when taken through an intermediate state. This property guarantees that a state will never be found at a lower f-cost than that at which it is first expanded. More formally, for three states, state A, state C, and intermediate state B, $h(AC) \leq c(AB) + h(BC)$. This guarantees that, when the states are examined in order of increasing f-cost, a state

that has already been expanded cannot later be found at a lower f-cost, because the g-cost of a given state on the open list is guaranteed to only stay the same or decrease (by definition), and h-cost cannot decrease by more than the increase in g-cost.

The quality of a heuristic can typically be determined by how large it is for a given state. Since admissible heuristics cannot overestimate the actual cost of the optimal path to the goal, the greater the heuristic value, the closer it is to the actual cost. This also means that, when given the choice between two consistent heuristic estimates for a state, it is prudent to choose the larger of the two. As a result, a common method for combining heuristics to generate heuristics using multiple means, and then choosing the maximum. If all of the chosen heuristics are admissible, then the result is guaranteed to be admissible, and it may also be of higher quality than a heuristic generated using a single method.

2.3.1 Grids in MAPF

The experiments in this thesis were performed on a four-connected grid. In this type of grid, all acceptable states for individual agents are locations on a grid, which can be denoted by an x and a y coordinate. Successor states for the single-agent problem, then, are the locations directly above, below, to the left and to the right of the current location (either the x or y coordinate can decrease or increase). Some locations may be considered "obstacles", locations where an agent cannot move.

A simple single-agent heuristic for the grid search problem described above is called "Manhattan distance". This is the sum of the difference of the x and y coordinates between the current state and the goal. On a four-connected grid with no obstacles, this heuristic is the exact cost of the optimal single-agent path to the goal. This may be expanded to the multi-agent problem by finding the sum (or the maximum, in the case of the makespan variant) of the Manhattan distances for the

individual agents. The quality of this heuristic, however, depends on the number of conflicts that occur between the agents in their paths to the goal. In this case, the conflicts can be considered the obstacles that make the heuristic less exact. As a result, this heuristic is of much lower quality for very difficult instances of MAPF.

2.4 Pattern Databases

For some problems, calculated heuristics can be very limited. Manhattan distance, for example, cannot account for obstacles or conflicts in the MAPF grid problems, leaving it less effective for more complicated problems. For problems such as these, it can be useful to create a more complicated heuristic using search techniques. A Pattern Database [1] is one such technique, where a part of the problem is abstracted away and a search is used to generate a heuristic. Each state in the original state space is given a mapping to a state in the abstract state space. The abstract problem is then solved using A* or best-first search for all states so that an optimal path is known. This can then be used as an estimate for the optimal cost of the path in the original state space. In most cases, the pattern database is calculated ahead of time in order to trade off upfront computation time for search speed. Pattern databases done in this way are typically used in situations where several searches are being performed, thus allowing the cost of the upfront computation to be offset over time.

Chapter 3

Survey of Recent Published Research

MAPF has been an active area of research in recent years. Quite a few new search based solvers have been described in recent publications. Much of this has been spurred by the deficiencies of the traditional A* search. Because the branching factor of the problem is exponential in the number of agents, the open list of states for even small problem instances with several agents can become far too large and difficult to manage in memory effectively. In addition, the process of expanding each of these states is extremely costly; generating all of the successor states requires at least constant time per state (since each state has to be generated once), and so the exponential blowup affects the time complexity of the search solution as well. In order to overcome these problems, these new search techniques make use of knowledge about the problem to avoid these costly expansions as much as possible.

3.1 Partial Expansion A*

One of the major problems resulting from the large branching factor of MAPF is the resulting size of the open list after several state expansions. These states must be maintained in memory, using a data structure such as a priority queue or a hash table, and maintaining these data structures as they grow large in memory can become very difficult. Furthermore, in problems with enough agents, there may not be enough memory at all to store all of the successors to more than a few states. Partial Expansion A* (PEA*) directly addresses this concern by storing in the open list only the successors with an f-cost below a certain bound [21]. The bound is maintained as the current best f-cost. When a state is expanded, any successors generated that are above this bound are thrown out. Once all successors have been generated and either placed in the open list, or thrown out, the current state is reinserted with a new f-cost equal to that of the lowest f-cost among all of the successors above the bound. This allows the algorithm to re-expand the state once the bound is raised and generate the successors at that next f-cost.

This technique vastly reduces the number of states that are stored on the open list for typical MAPF problem instances, and can reduce the overhead involved in maintaining the data structures essential to A*. The downside, however, is that many states are generated, and then thrown out. These states may become relevant when the bound increases, and so they may be generated multiple times. In fact, any time a state is re-expanded, all of its successors are generated again, including those that have already been closed, and those that may never be expanded. Thus, PEA* may significantly increase the time required to solve a problem instance.

3.2 Enhanced Partial Expansion A*

Because PEA* can be important for solving new problems within the memory constraints of a computer, further attempts have been made to develop this technique. Enhanced Partial Expansion A* (EPEA*) builds on PEA* with an attempt to eliminate the additional time required by re-expansion [4]. In this technique, an f-cost bound is decided upon (typically the f-cost of the initial state), and each time a state is expanded only the successors with an f cost equal to the f-cost bound are generated. This must be done with some prior knowledge of the inner workings of the heuristic in use. For instance, in Manhattan distance, it is possible to know which actions are going to increase or decrease the h-cost of a state, based on whether they are numerically closer or further from the goal. This means it is easy to choose which successors to generate based on knowledge of the current g-cost, and the effect that an action will have on the h-cost. This eliminates all of the duplicate state generation that could cause PEA* to be slow while maintaining the memory benefits as well. In addition, since only the states at the bound are ever generated, there are speed savings identical to the memory savings. The primary downside to this technique is the requirement on the heuristic; it must be possible to predict the effect of an action on the heuristic, meaning it will be difficult to combine with more complicated heuristic techniques such as pattern databases.

3.3 Operator Decomposition

An alternate way to reduce the size of the open list that works specifically for MAPF problems is Operator Decomposition [16]. In OD, when expanding a state, rather than generating all of the actions by generating all combinations of acceptable actions for all agents, partial successors are found by applying an action to just one agent at a time. These partial successors are added the open list as usual, but are

marked so that, when expanding this decomposed state, the only actions available are those of the next agent in line. Only fully generated states (states where every agent has performed a single action since the last fully generated ancestor) are added to the path, but otherwise these states are treated as normal. The advantage of this method is that it allows the prioritization of the open list to determine which successors to a given state are generated. This means many states where an individual agent's action was obviously wrong (for example, if the agent moved in the obvious wrong direction) are never generated. For each of these situations, the generation of a large number of successors is avoided ($(n-1)^k$, where k is the number of single-agent actions). The disadvantage is that these intermediate states, which are not complete, can add an additional burden to the open list, thus resulting in higher memory cost.

3.4 Increasing Cost Tree Search

The solutions covered thus far have all used an A* implementation as the basis for further search enhancements. Increasing Cost Tree Search (ICTS), instead, attempts to reduce the problem into a tree of sub-problems that can be solved in order to find a solution [14]. This technique actually consists of two searches: a high level search of a tree called an Increasing Cost Tree (ICT), and a low level search that attempts to find a solution that satisfies the constraints of the current ICT node. An example of the high level ICT can be seen in Figure 3.1. Nodes in the ICT specify a constraint on the cost of the path for each agent in the problem instance. The root of the ICT is a node containing the costs of the optimal single-agent paths for each agent. The low level solver, then, attempts to generate all paths that satisfy the path cost constraint, then attempts to find a combination of paths for all agents that do not conflict. This is accomplished using a technique that examines the paths one time step at a time and creates of tree of possible path combinations. If a tree can

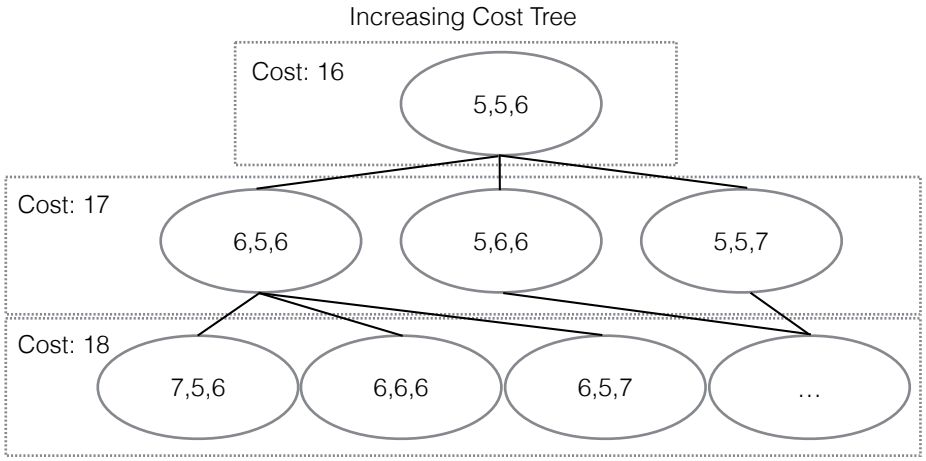

Figure 3.1: Example Increasing Cost Tree

be completed from start to finish for each agent, then a set of paths that comprise a solution can be recovered. If no solution is found, the successors are generated for the current ICT node by adding the cost of an action to the constraint for one of the agents in the node. This tree is searched in a best-first manner (in terms of total path cost), thus guaranteeing that the solution minimizes the cost throughout all agents. This technique is effective at searching a large number of paths quickly because a joint A* search is not required, which avoids the exponential nature of A* state generation for multi-agent problems, but the high level search can generate a large number of nodes before arriving at the correct cost and the low level search still has to evaluate all combinations of paths between agents, which in some instances may be exponentially large.

3.5 Conflict Based Search

Conflict Based Search (CBS) is another technique that breaks the search into high-level and low-level searches [12]. In the high level, a Constraint Tree is created.

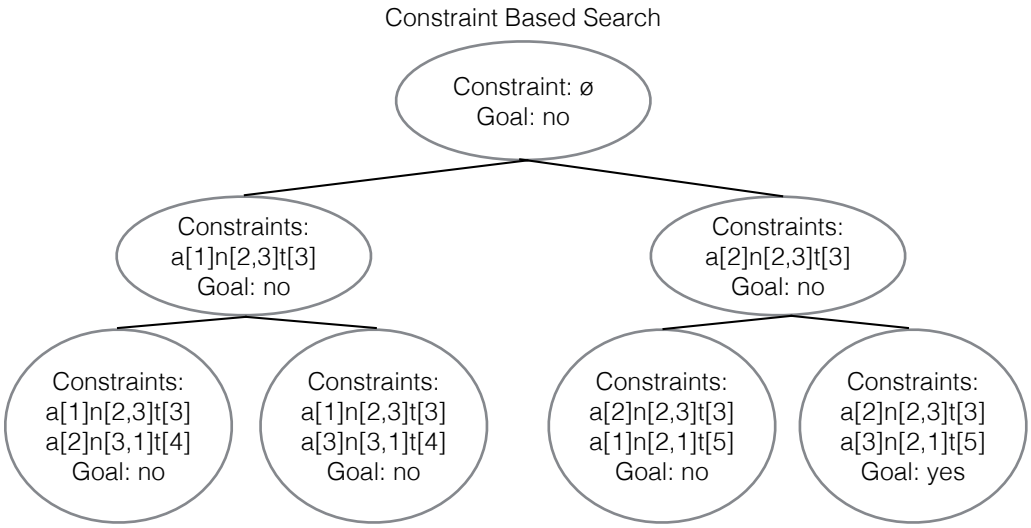

Figure 3.2: Example CBS Constraint Tree

An example of the Constraint Tree can be seen in Figure 3.2. The Constraint Tree is made up of nodes, each of which has a list of constraints. The constraints, in this case, are agent-location-time tuples that specify a location an agent cannot use at a given time. The root of the Constraint Tree has no constraints. Each node in the Constraint Tree is evaluated by performing single-agent searches for each agent to obtain a set of paths. The paths are then evaluated for conflicts. If a conflict is found, child nodes are created for the current Constraint Tree node each with a constraint for the location and time of the conflict, with each new node constraining one of the agents in the conflict. If no conflict is found, then the current node is marked as a goal node and the paths are returned. The Constraint Tree is searched in a best-first manner until a goal node is found. This search can quickly resolve conflicts between agents, especially in problems where conflicts are rare. The Constraint Tree, however, is exponential in the number of conflicts, meaning problem instances with a high likelihood of conflicts will cause problems for this type of search.

3.6 Meta-Agent Conflict Based Search

Meta-Agent Conflict Based Search (MA-CBS) modifies the standard CBS implementation to make use of other optimal MAPF solvers [13]. In MA-CBS, a typical CBS search is performed, but a policy is put into place to combine agents into a "Meta-Agent" once a certain number of conflicts have been found. This Meta-Agent is then solved in subsequent nodes using any MAPF solver (including MA-CBS itself). The policy can be adjusted to balance the cost of searching larger instances at the low level with the cost of creating more branches in the Constraint Tree. This makes it possible to control the exponentially expanding Constraint Tree by delegating some of the conflicts to another available solution method.

3.7 M*

M* is a technique that makes a modification to A* to simplify the search and avoid generating many states in the parts of the search space where there are not conflicts [19]. M* accomplishes this by initially limiting the successors generated at each expansion by defining an optimal policy for each agent and only generating the successor states where all agents move according to their policy. This process continues until a collision is found. Each time a collision is found, all agents in conflict are added to a collision set which is maintained as part of the open list entry for each search state. M* uses a process called back propagation to update the collision set for each search state along the path to the collision state, while also returning each such state to the open list. From that point forward, whenever a state is expanded that has a non-empty collision set, the optimal policy for the agents in the collision set is replaced by a policy allowing movement in any direction. This allows the search to proceed from that point forward in a way that allows the agents in question to avoid a known conflict. This technique is similar in operation to

EPEA*, in that it limits the number of successors generated during each expansion as a way of avoiding the full exponential nature of the problem. The difference, however, is that instead of revisiting states later based on the f-cost of the successor states that were not added to the open list initially, M* only reopens states when a conflict has been found. This means that in problems with few conflicts, the M* search will proceed quite a bit faster than the joint A* search, but the bound on state expansions for problems with many conflicts is actually higher for M* because of the back-propagation step.

Chapter 4

Foundations for Our Contribution

The foundations for the research covered by this thesis span both general techniques for pathfinding and techniques developed for MAPF specifically. These techniques are being combined in new ways as an attempt to create better heuristics for MAPF.

4.1 Hierarchical A*

Hierarchical A* [8] takes abstraction further by creating a hierarchy of abstractions (an example abstraction hierarchy is shown in Figure 4.1). An abstraction for the primary graph is created by grouping multiple states of the primary graph into a single state in the abstraction. Groups are connected in the abstraction whenever states within a given group are connected to states within a different one. The heuristic values are obtained by searching the corresponding states in the abstract graph, and returning the length of the path. Next, another abstraction is created to provide heuristics for search in the first abstraction using the exact same process.

This process is repeated recursively until the final abstraction, which will contain only a single state, which is solved by default because the start and the goal state will be the same. An example abstraction hierarchy is shown in Figure 4.1. Some states in the search graph are grouped together into abstract states A, B, and C in the first abstraction, with S and G remaining ungrouped. In the second abstraction S, A and B are grouped into a single abstract state while C and G are grouped into another. A third abstraction could be created by grouping S and G together from the second abstraction, although this is unnecessary in practice since doing so trivially results in the zero heuristic.

Now, a search can be performed on the primary graph, and heuristic values are generated using searches in its abstraction. These searches, in turn, are improved using heuristic values generated using searches in another abstraction. As a result, a hierarchy is formed with each search being sped up by an abstraction. In Figure 4.1, the heuristic lookup for S in the search graph results in a request for a heuristic from the first abstraction. This results in a search beginning at state S' in the abstraction. Before beginning the search in the first abstraction however, a heuristic value for S' is found using a search in the second abstraction; since there is only a start and a goal state, the returned h-cost would be 1. The search in the first abstraction would then continue, using the second abstraction for heuristics, until it returned a heuristic value for S in the original search (in this case, 2). Looking at the two successor states to S in the search graph, m and o, m is part of the abstract group A and o is part of the abstract group B. Since A in the first abstraction is at a distance of 1 from the goal and B is at a distance of 2, the search would expand m next and continue through n to the goal, avoiding further expansions in the other direction and thus saving time. Even the second abstraction, for all of its simplicity, can save search effort; during the expansion of S', heuristic values for A and B are obtained, and since A is in G'' and B is in S'', the result for A will be 0 and B will

be 1; the search will then proceed directly through A to the goal without expanding B.

Since the process of collapsing multiple states into a single group can only result in abstract paths of an equal length to their counterparts or shorter (as the result of multiple intermediate states collapsing into one group), the resulting heuristic value is guaranteed to be admissible. For example, in Figure 4.1, the optimal path goes S-m-n-G. In the first abstraction, two of these states (m and n combine into the group A, which has the effect of shortening the path to S-A-G. However, even if these states were grouped separately in the abstraction, the optimal path could never be longer than the optimal path in the original search graph because each state is only ever part of a single group and there must be edges between groups if the states inside each of those groups have edges in the search graph.

Note that, unlike with Pattern Databases, Hierarchical A* performs the abstract searches on demand, eliminating the time required to perform the exhaustive search ahead of time, in hope that the search hierarchy would perform its look-ups fast enough to decrease total search time, even for a single problem instance.

Because it would be inefficient to perform a search in the abstraction for every heuristic lookup, a shortcut is used called "g-cost caching". In this method, a complete A* search is performed from the start to the goal in each abstraction. Then, whenever a heuristic is requested, it can be found by subtracting the g-cost of the state in the abstraction from the cost of the optimal path in the abstraction. The resulting cost is guaranteed to be less than the cost of the actual optimal path from the abstracted state to the goal. The g-cost, in this case, is obtained by either checking the cost from the list of expanded states (the "closed list"), or by searching until the state is expanded and using the resulting g-cost. The resulting formula for any arbitrary state P in a search graph with a goal G where P' and G' are the corresponding states in the abstraction is: $h(P) = g(G') - g(P')$.

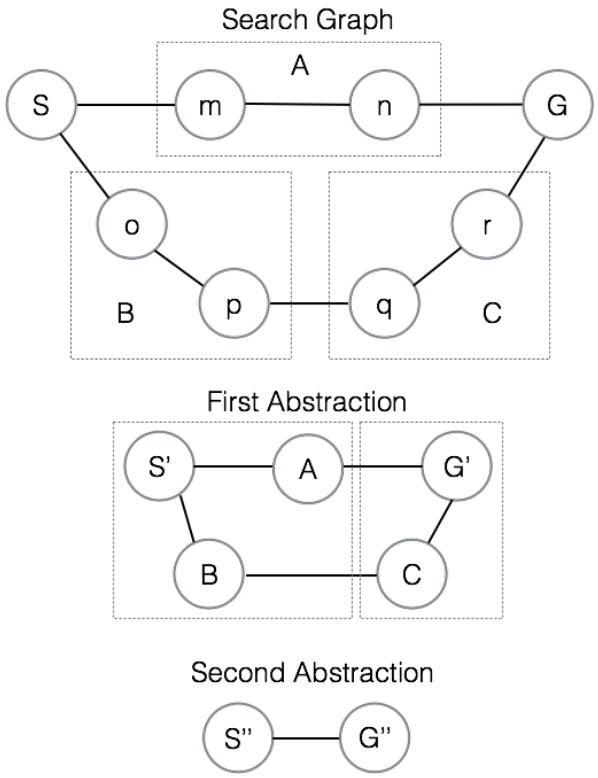

Figure 4.1: Example abstraction Hierarchy.

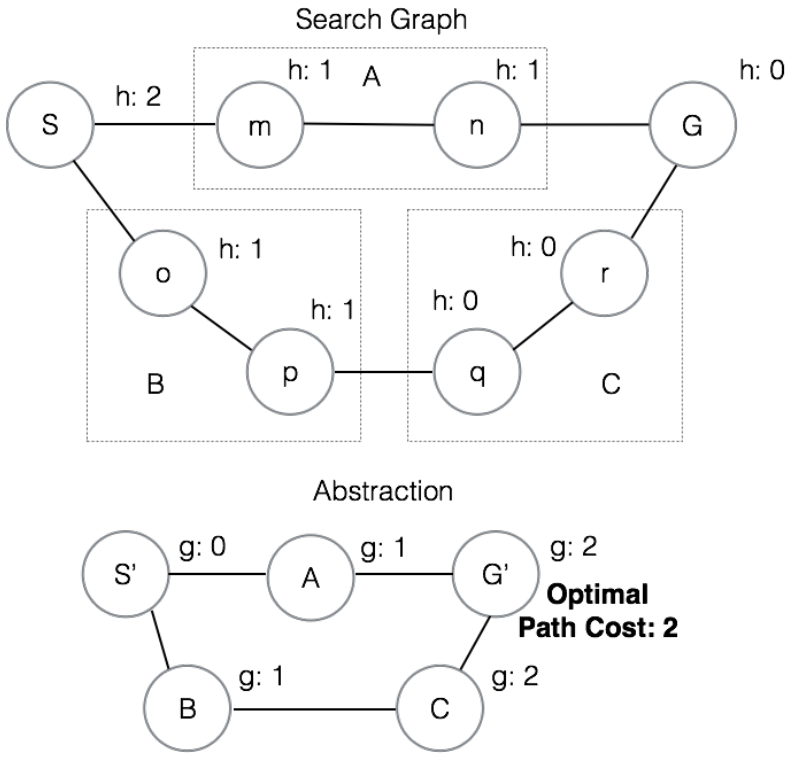

Figure 4.2: Example of heuristic obtained using g-cost caching.

We can see how g-cost caching works in Figure 4.2. We first perform a search from S' to G' in the abstraction in order to obtain the optimal path cost. With this value, we can now begin to fill in h-cost for the search graph by mapping each state to a state in the abstraction, and the difference between the optimal path cost and the g-cost of the abstracted state. For example, the g-cost at S' in the abstraction is 0, so the h-cost of S is 2. Note, however, q and r map to C in the abstraction and as a result both receive an h-cost of 0. This is the downside of the g-caching approach; although the values obtained are exact along the optimal path (since the distance to the goal for states along the optimal path is exactly the optimal cost minus the g-cost), along suboptimal paths the values underestimate the actual distance to the goal, making for a lower quality heuristic.

This technique aims to improve the speed of a search using abstraction as guide. The first abstraction is in place to reduce the number of expansions required to find the goal in the primary search, while the second abstraction is in place to reduce the effort required to search the first abstraction. Ideally, with each abstraction forming a much smaller search space, the effort required to search the abstract graphs is much less than the effort of additional expansions in the original search, leaving a net gain. Unfortunately, as stated above, the quality of g-cost caching as a heuristic can vary dramatically because, as the state lookups get further from the actual optimal paths, the estimate gets worse. In order to mitigate this problem, the heuristic obtained from abstraction can be combined with another heuristic, such as Manhattan distance, that doesn't suffer from this issue. In this case, you can take the maximum of the g-cost caching heuristic and the Manhattan distance heuristic to better select states closer to the goal. For instance, although the g-cost caching heuristic value will go down even for states in the opposite direction of the goal, Manhattan distance will go up, resulting in a heuristic that both directs the

search toward the goal, and has the additional information that can be gathered by searching an abstracted search space.

Hierarchical A* can also be combined with another variety of the A* algorithm (Iterative Deepening A* or IDA*) that uses an iteratively deepening depth-first search to find the goal[7]. This search uses repeated depth-first searches from the start state bounded by f-cost. This type of search works well for exponentially expanding search spaces because the reduced overhead from not tracking states that have already been visited makes up for the search repetition. However, in searches where there are many cycles or many ways to reach the same state, this search is less effective because it will perform exponential depth-first searches even if the state space has many duplicate states. Because we are primarily working in a grid space where many cycles exist, we do not use this type of search in this paper.

4.2 Switchback

A related technique which improves upon Hierarchical A* is called Switchback [9]. This approach also uses a hierarchy of abstractions, but instead of performing a standard A* search from start to goal in the abstracted search space, the search is instead reversed in each abstraction. Reversing the direction of the search results in an interesting phenomenon; instead of needing a new search to the goal from each state in the abstraction when an h-value is needed, or use g-cost caching and lose fidelity, a single search can be performed starting at the goal in the abstraction and the g-costs discovered during that search can be used directly as a heuristic. This is because an A* search with a consistent heuristic closes states only when they have been found at their optimal g-cost. This means the g-cost of the state when closed is exactly the length of the optimal path from the starting state of the search (which,

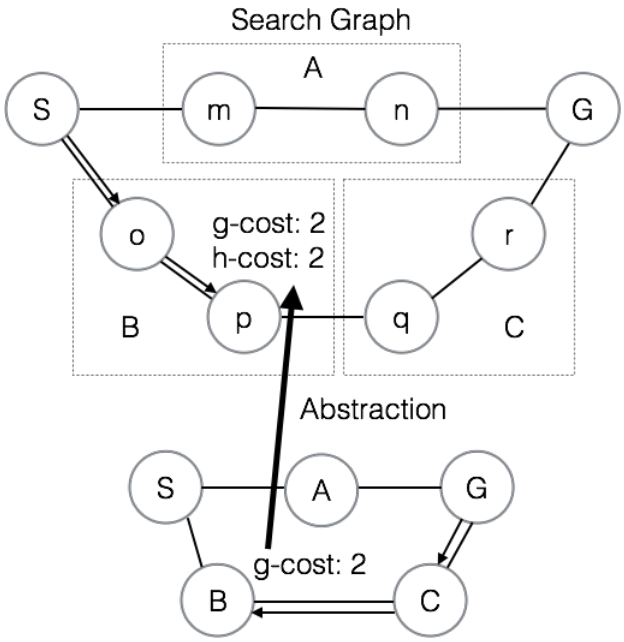

Figure 4.3: Example of Reverse Search as a heuristic.

in this case, is the abstract equivalent of the goal state) [1] This reversal obviates the use of g-cost caching since, when in the reverse search space, the g-cost of the search starting from the goal is the exact length of the shortest path between the state in question and the goal. Thus, the g-cost in the abstraction can be used directly as a heuristic in the non-abstracted state space. With each level of the search hierarchy reversed, then, a better heuristic is obtained on each lookup, and the result is a much faster searches throughout the hierarchy. Figure 4.3 shows the use of reverse search as a heuristic. The g-cost of the search node in the original graph is found to be 2. In order to find a heuristic value, a reverse search is performed in the abstract space. Because of the reversal of directions, the g-cost of node B in the abstract search space can become the h-cost of the node in the original search space.

[1] For inconsistent heuristics this approach could still be used but a full search of the state space would likely be required to ensure each state had been found at its optimal g-cost. In this case even a low quality consistent heuristic such as the zero heuristic may be preferable.

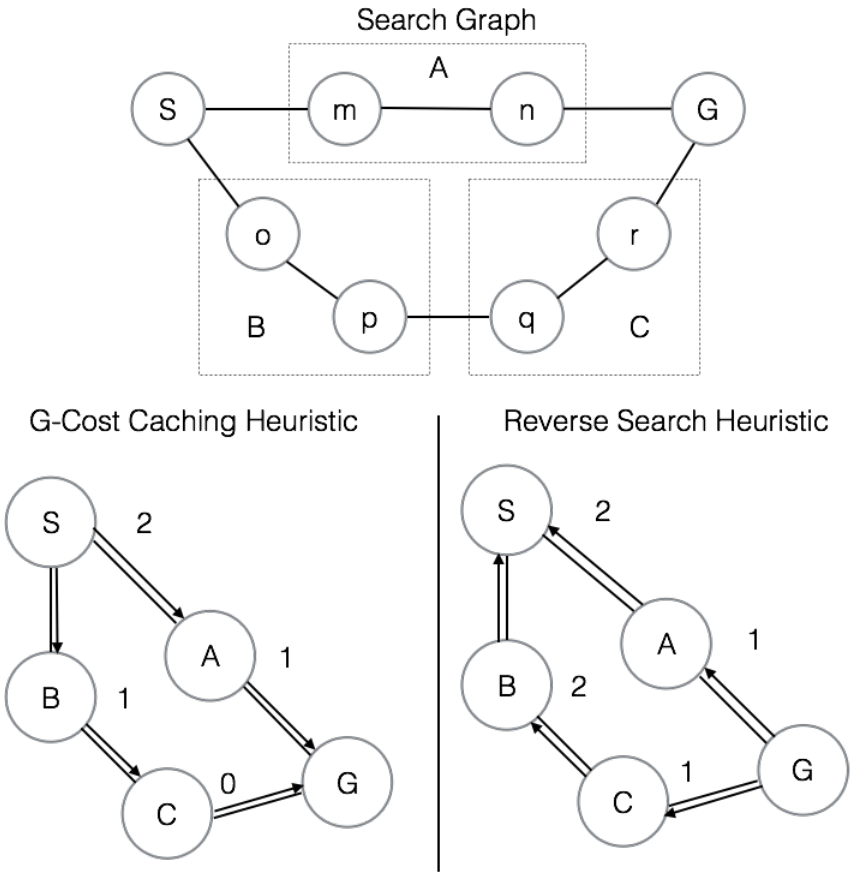

Figure 4.4: Example of improvements from Switchback methodology.

The advantages of this technique are illustrated in Figure 4.4. When choosing the first state to expand after the start state, we check the h-values of states m and o. In the case of g-cost caching, both m and o have an h-cost of 1 and a g-cost of 1 for an f-cost of 2, so either state may be expanded with equal likelihood. This means there is a 50% chance the longer path will be explored first. Using reverse search, however, o has an h-cost of 2 resulting in an f-cost of 3, so m will be chosen next. Next, assuming we tie-break toward the higher g-cost, n (g: 2, h: 1, f: 3) will be expanded after m instead of o (g:1, h: 2, f:3), followed by the goal (g:3, h:0, f:3) and we don't need to explore the longer path.

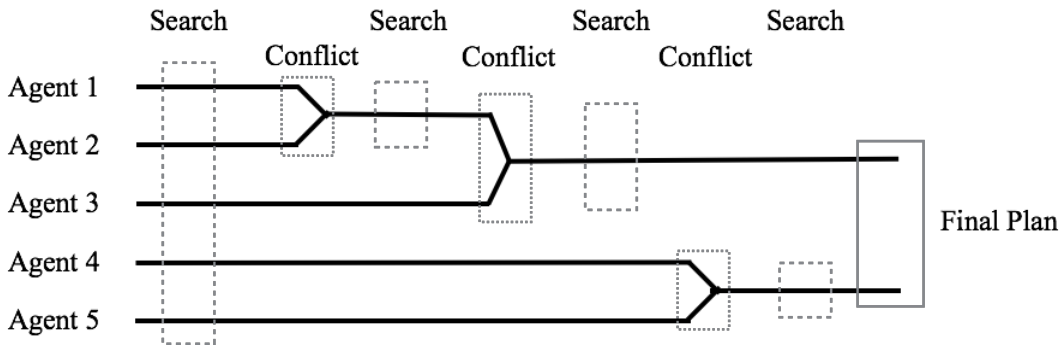

Figure 4.5: Example of agents being combined during ID search.

4.3 Independence Detection

In many MAPF problem instances, there are relatively few agents compared to the overall size of the search space, as well as relatively few places where those agents may conflict. This is especially apparent in large grid problems, where there may be far more open grid locations than occupied locations. As such, it may not always be necessary to perform a combined search for all of the agents; it may indeed be enough to perform a single-agent search for each agent and then check that the resulting paths do not conflict. This gives way to the tactic used by the Independence Detection (ID) technique [16]. An ID solver separately searches for optimal paths for each agent. Once this is accomplished, all paths are checked for conflicting states or edges. Each of these conflicts are either resolved by revising one of the agents' paths or by combining the agents and searching for a combined solution using A* or another MAPF solver. The essential benefit of this algorithm comes as a result of the exponential nature of the problem: each agent that is added to the problem multiplies the search space so that even several searches with fewer agents will not be as costly in terms of time and memory as a single search with all agents. Figure 4.5 shows an example of the way agents will be combined into multi-agent search groups during the operation of an ID search.

The details of the ID algorithm are shown in the pseudocode Algorithm 3. All agents are initially placed into groups of their own. A group is a collection of one or more agents that will have their paths found in a single A* search (line 2). Once this is done we enter our main loop. A path is found for each group individually, and these paths are collected and placed in the Conflict Avoidance Table (CAT) (lines 4-7). The CAT is used during conflict resolution searches to help avoid paths that would conflict with one of the other agents in the problem which may not be part of the current problem. At this point, a simulation is performed, stepping through the paths one time step at a time, checking for conflicting states or edges (lines 8-9). If two (or more) groups conflict, a search is performed for each conflicting group to determine if new paths can be created that avoid the conflicting states or edges (lines 14-18). If a new path is found that does not conflict, this path is added in place of the original, and the simulation is restarted. If a new path could not be found satisfying the conditions, the conflicting groups are combined into a single group that will be jointly searched using a multi-agent search (line 19). To prevent problems with a group repeatedly conflicting with the same groups, whenever a conflict is found between the same agents more than once, the agents are combined to prevent further attempts to avoid the same conflict (lines 11-12). This process is repeated until non-conflicting paths are found for all agents or groups, or all agents are combined into a single group, at which point the combined search is performed.

ID works to solve MAPF problem instances more efficiently by using searches containing as few agents as possible. Because the state space of a multi-agent search with k agents is exponentially larger than that of a search with $k-1$ agents, the savings when using this technique can be immense. When there are few conflicts between agents, only a few searches, each with a small number of agents, will be run. This means the running time of this algorithm is dominated by the most expensive search, i.e. the search with the largest number of agents. As a result, even when ID

Algorithm 3 Independence Detection

```
1: function IndependenceDetection(start, goal)
2:     Add Agents a_1...k to Groups g_1...k
3:     repeat
4:         for all groups 1...k do
5:             p_i ← A*(agent_i[start], agent_i[goal])
6:         end for
7:         Add all paths to Conflict Avoidance Table (CAT)
8:         for all paths p_i ∈ p_1...p_k do
9:             check path for conflicts with p_1...p_{i-1}
10:            if Conflict is found with group g_j then
11:                if Groups g_i and g_j have conflicted before then
12:                    Merge groups g_i and g_j
13:                end if
14:                illegal ← p_j
15:                path ← A*(g_i[start], g_i[goal], illegal)
16:                if Cost(path) > Cost(p_i) then illegal ← p_i
17:                    path ← A*(agent_j[start], agent_j[goal], illegal)
18:                    if Cost(path) > Cost(p_j) then
19:                        merge agents i and j and start again from line 2
20:                    else Restart from line 8
21:                    end if
22:                else Restart from line 8
23:                end if
24:            end if
25:        end for
26:    until No conflicts
27: end function
```

combines all agents and performs a search of the full k-agent problem, it will not take significantly more time to run than a joint A* search [16]. The result, then, is an algorithm that solves easy problems at a greatly reduced cost without any asymptotic increase in running time over a full joint A* search.

One of the downsides of the Independence Detection process is that it can be sensitive to the order in which conflicts are evaluated. The independent groups within a problem can vary, and solutions can be found with different sets of independent groups. Since it is a harder problem to determine true independence, ID as implemented combines groups if they conflict more than once. This can lead to different sets of independent groups simply based on which order we choose to resolve the conflict. This can be seen in Figure 4.6. When conflicts between agents are resolved in alphabetical order (from A to E), groups are combined at each step until all agents except for B have been combined. On the other hand, if conflicts are resolved in a permuted order (E, D, A, C, B), the largest group will contain only three agents, allowing the problem to be solved 5 times faster than in the original order and around 25 times faster than the joint A* search.

4.4 Pathmax

Inconsistent heuristics can pose a dilemma for A* search: a better heuristic can often lead to much better performance, but an inconsistent heuristic can result in reopened nodes because the f-cost of nodes found during a search are no longer guaranteed to increase uniformly from parent to child. The reopening of nodes can lead to an increase in the bound of node expansions from $O(N)$ to $O(2^N)$ [10]. One way to resolve this problem is the pathmax method [11]. Pathmax can resolve inconsistencies in the search graph by propagating h-costs from states to their successors and vice versa. This is accomplished by applying the following rule: for a given parent search state S, and its successors $s_0 ... s_n$, the h-cost of a given

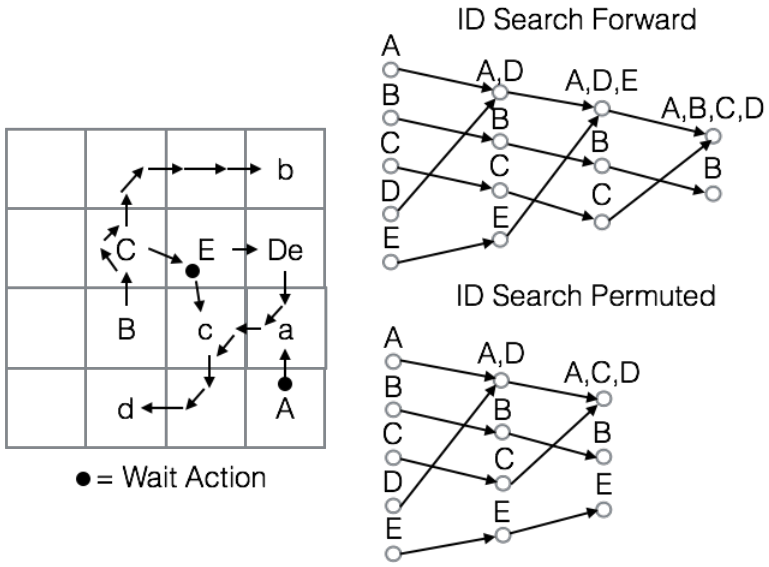

Figure 4.6: Example of multiple independent groupings for the same problem found based on ordering.

successor s_i is $\max(h(s_i), h(S) - \text{cost}(S, s_i))$. The parent then can also be updated by the rule $h(S) = \max(h(S), \min_{i=0}^{n}(h(s_i) + \text{cost}(S, s_i))$. Bidirectional pathmax (BPMX) [5] takes these rules and applies them in both directions.

One other use for pathmax is to make use of incomplete heuristics. Suppose we want to save time by only computing a heuristic along the optimal path in an abstraction. If we simply use those values along with another, lower quality heuristic for states that aren't along that path, we may end up prioritizing states that aren't on that path, even though they might not be the best choice. Consider Figure 4.7; if we use the incomplete heuristic for the optimal path and a zero heuristic everywhere else, o (g: 1, h:0, f:1), p (g: 2, h:0, f:2) and q (g: 3, h:0, f:3) will all be expanded before states along the optimal path starting at m. If, instead we use pathmax to perform updates, o will get an updated heuristic value of 2 and f-cost of 3, and there will be a 50% chance that we will choose to expand m (g: 1, h:2, f:3) first, followed by n (g: 2, h:1, f:3) and the goal.

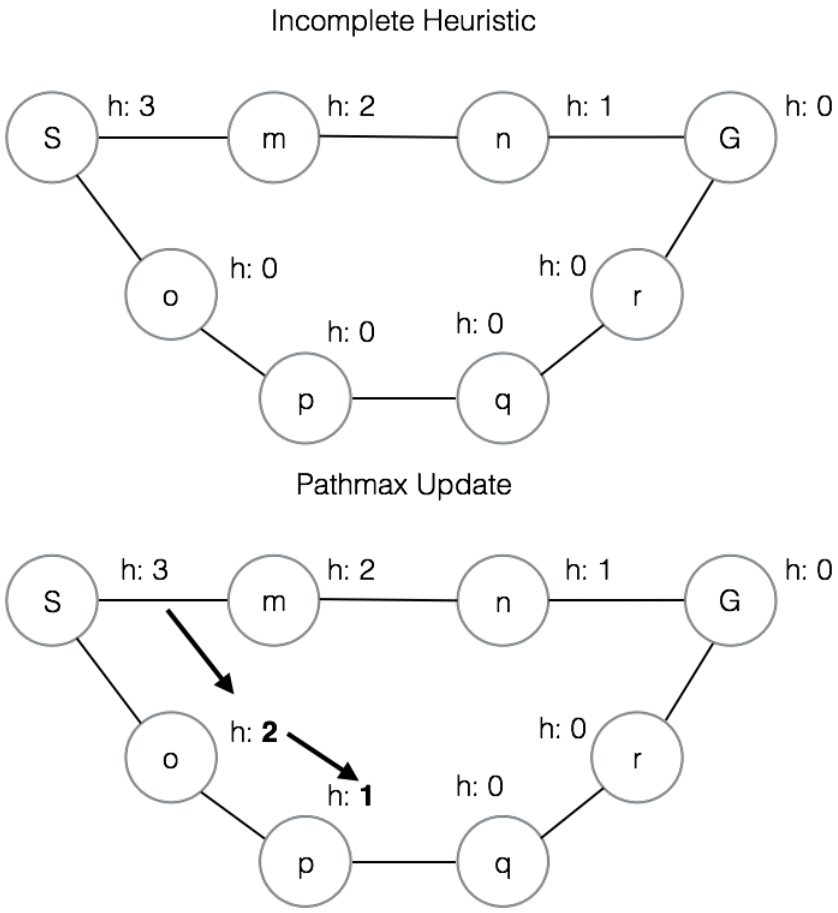

Figure 4.7: Pathmax updates to an incomplete heuristic.

Chapter 5

Heuristics for Multi-Agent Pathfinding

Since the version of Multi-Agent Pathfinding we've been examining is based upon the simpler single-agent grid-based pathfinding problem, the most basic heuristics for MAPF build upon heuristics created for that problem. Manhattan distance, as described above, is adapted for this problem space by adding together the individual costs calculated for each agent. This can work well for very simple instances of the problem, where conflicts are unlikely to occur, but the issue for this type of heuristic is the inability for any single-agent heuristic to account for conflicts. As more conflicts occur, these cost estimates become less and less adequate for predicting the best path, and therefore we must find more powerful methods for generating heuristics.

One method for creating a heuristic for any problem is to create an abstraction, and use information gained from that abstraction to predict the cost of paths in the primary problem space. In order to prevent this method from negatively affecting performance, the cost to search the abstraction must be significantly lower than the cost of searching the primary problem space. In addition, to avoid the problem

stated above, the abstraction must have a way of taking into account the conflicts between agents, so that the resulting heuristic is more robust and can provide better estimates than simpler, combined single-agent heuristics.

5.1 Abstraction for MAPF

One simple method for creating an abstraction is to simply remove agents from the problem. For example, a 5 agent problem can instead be represented as a 4 agent problem and a single-agent problem. The resulting abstraction will take into account all of the conflicts that occur between the 4 agents in the single group, as well as providing a high quality heuristic incorporating all of the obstacles (if necessary) for the individual agent. A final heuristic can be computed by either adding the two costs (for sum of cost variants) or by finding the max of the two costs (for the makespan variant). Since the 4 agent problem is 5 times smaller than the 5 agent problem, the cost of searching this abstraction is significantly reduced, minimizing the performance impact of the search.

This process can be completed using any number of underlying abstractions as long as they are disjoint and collectively contain all of the agents in a given problem instance. For example, if you have a 5 agent problem, you can divide the agents into 2 disjoint groups, one of 3 agents and one of 2 agents. All 5 agents must be present, but the size of the groups can be any set of numbers that add up to the total, for instance $3 + 2$, $4 + 1$ or $2 + 2 + 1$. Although a k-agent problem can be divided into up to k groups (each containing one agent), for the research performed in this paper all problem instances were divided into only 2 disjoint groups of agents.

Formally, any k agent MAPF problem can be divided into two problems with r agents and $(k - r)$ agents respectively. The agents in each problem are then a subset of the agents in the full problem. In order to guarantee that our heuristic is admissible, the sum of the heuristics obtained by searching each of those problems

must be no more than the actual cost to get to the goal in the full problem. This can be proven by considering something fundamental about MAPF problem instances: the increase of the cost of paths for agents in MAPF over the cost of the optimal single-agent path is entirely the result of interactions between agents that create constraints on the paths available to the agents. This means that adding agents to a MAPF problem instance can only increase the cost of the paths of those agents already in the problem. As a result, when we take agents out of a problem, the cost of the paths for those agents in the resulting problem must be equal to or less than cost of the paths of those agents in the full problem. Since this is true for every subset of agents in the full problem, the sum of costs of the paths of the abstracted problems (or the maximum of the costs in the makespan problem) must be equal to or less than the sum of the costs of the paths of those agents in the full problem.

This heuristic will be consistent as long as the cost function of the problem obeys the triangle inequality. This is because the change in the heuristic values between two neighboring states is comprised of the cost of the actions the agents need to take to transition between the two states in their abstraction. As a result, the change in the heuristic values can never exceed the cost of the individual actions the agents take between the two states, and thus the cost of the transition between the two states.

Note that in this abstraction, the value r can be any number less than k. This means we can create abstractions that remove a single agent, or two agents, or half of the agents. The advantages of $r > 1$ would be a much smaller search space at the cost of the accuracy of the resulting heuristic. Since the $k - 1$ agent problem is so much smaller than k agent problem, there isn't much of a need to reduce it further, however the additional work may not be necessary. Imagine, for instance, that only 2 agents in a 5 agent problem actually conflict. The result is that no additional information is obtained by adding agents to the group of the conflicting

agents. Thus, all of the complexity of the problem can be captured by simply using a 2 agent search and 3 one-agent searches, whereas running a 4 agent search and a single-agent search would be more costly in time and memory but result in no advantage.

The final heuristic value computed from this method has limitations, however, because of g-cost caching. Because of the nature of the g-cost caching mechanism, the heuristic values returned will be lower as they get further from the optimal path. This means there is a potential for very low values for states that are very far off the optimal path, resulting in those states having underestimated f-costs. This may result in them being expanded before states that are more likely to be on the optimal path. To offset this problem somewhat, we combine the result of the g-cost caching heuristic with the Manhattan distance heuristic by taking the maximum of the two. Since Manhattan distance is a simple calculation, it requires negligible computation time and memory to obtain, which makes it a good quality base heuristic. Since both heuristic calculations are admissible, the maximum of the two is guaranteed to be admissible as well.

5.2 Abstraction Hierarchies

With the above abstraction mechanism, it is possible to create abstraction hierarchies for the purpose of a hierarchical A* search. At each level, an abstraction is created by splitting the problem into two problems with fewer agents. The result is a hierarchy of abstractions, each with a smaller number of agents, until the final level of abstraction where there are simply two single-agent searches. Figure 5.1 shows the process of creating an abstraction hierarchy for a five agent problem. The initial problem is broken into two problems, one with 3 agents and one with 2 agents. These problems are searched and used to create heuristics for the initial problem. Note that on this 1st level of the hierarchy, the group on the left has two

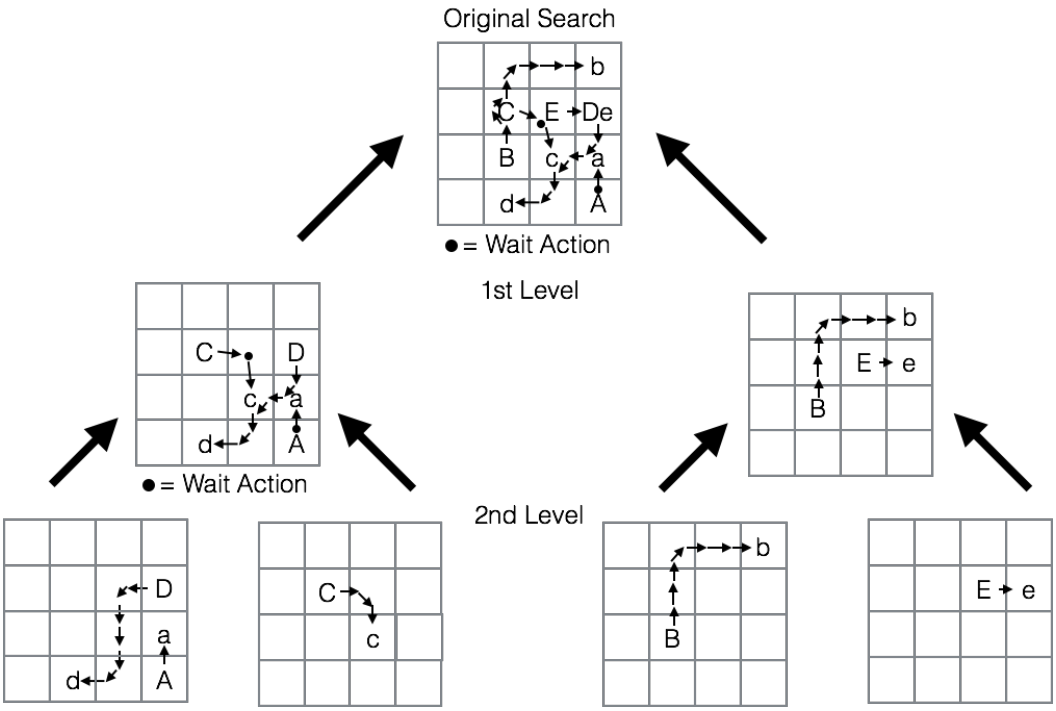

Figure 5.1: Example of a search being broken down into independent searches with fewer agents in a hierarchy.

conflicts: agents A and D conflict at time 2 and agents C and D conflict at time 3. At least one of these agents will need to perform a wait action at the given time. This information will be used to improve the initial search. The second level of the hierarchy is created by breaking each of those problems into two smaller problems. For brevity, a final level of the hierarchy is omitted that would show the problem on the bottom left being broken up into two single-agent problems.

The value of the abstraction hierarchy lies in the way the abstractions are produced. Valtorta's Theorem [18] states that any state that would be expanded in a search without the abstraction must be expanded either in the search or in the abstraction. However, because our abstractions each map multiple search states to a single abstract search state, reductions can be obtained using g-cost caching because the abstract state must be expanded only once, and then can be used multiple times

for heuristic values. Furthermore, the time and memory cost of expanding states in the abstraction is at least 5 times lower than the time and memory cost of an expansion in the search because the number of actions and state space are both 5 times smaller for each agent removed. This means any savings in expansions in the search is likely to offset the overhead cost of maintaining the hierarchy.

5.3 Switchback for MAPF

The best way to offset the detriments of using g-cost caching would be to use the previously described Switchback approach. In theory, this works well; reverse search can be used to find the exact distance in the abstraction which will eliminate the limitation that g-cost caching values have for states that are far away from the optimal path. The primary issue with this approach has to do with a simple observation; although states in MAPF can be described solely in terms of the positions of the individual agents, time is actually an important factor. Since this is a time based problem, and the available states at one point in time depends on the available states in the previous time, the induced graph from MAPF problems is actually directed. For example, the states available during the second time step are only those adjacent to the start. This presents a bit of a problem for reverse search. The actual time step of the problem at the goal is unknown and it's impossible to determine which states are actual reachable from that time step. The only way to create an abstraction, then, is to remove time from the state lookup procedure and assume that the cost does not depend on time. While this may be true when using the fixed cost variant of MAPF,[1] it can result in incredibly bad performance.

[1]The switchback approach is particularly detrimental to the Sum of Individual Costs (SIC) variant because the amount of time required by each individual agent to reach its goal is important. This means that simply swapping the start and goal states for a reverse search will not result in a solution that is equivalent to the forward search. Figure 5.3, for instance, would pose many issues for a reverse search when using SIC.

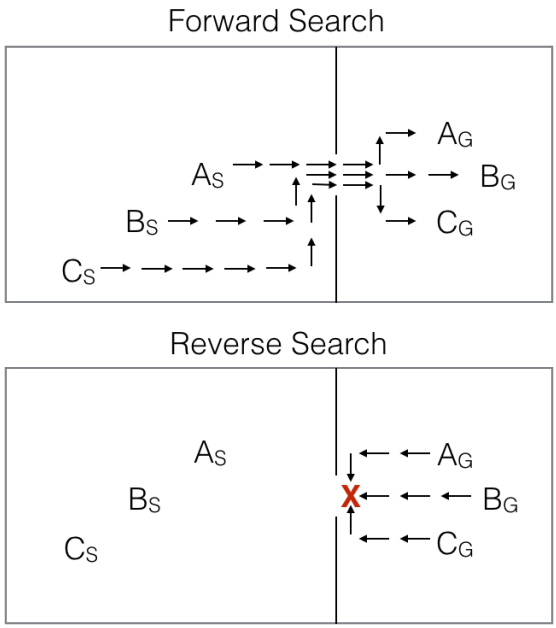

Figure 5.2: Using reverse search to solve a problem may result in conflicts that don't occur in the forward search.

Poor performance using switchback is often because of an asymmetry between the forward and reverse problem. Consider Figure 5.2. In this scenario, the forward search will proceed fairly quickly to the goal, likely without conflicts to resolve. If we search in the reverse direction, however, there will be conflicts as all three agents try to cross through the gap in the wall at the same time. If we were using a reverse search as an abstraction, even if we abstract away one of the agents, there are still conflicts that need to be resolved in the abstraction that wouldn't need to be resolved in a forward search. This will likely reduce the performance of the overall search rather than improve it.

Another issue where asymmetry comes into play is that a reverse search can tend to find its way to states that would not likely be found in the forward search. Figure 5.3 shows a sample problem and a state that reverse search might find that the forward search would likely never generate. The agents, A and B, when searching

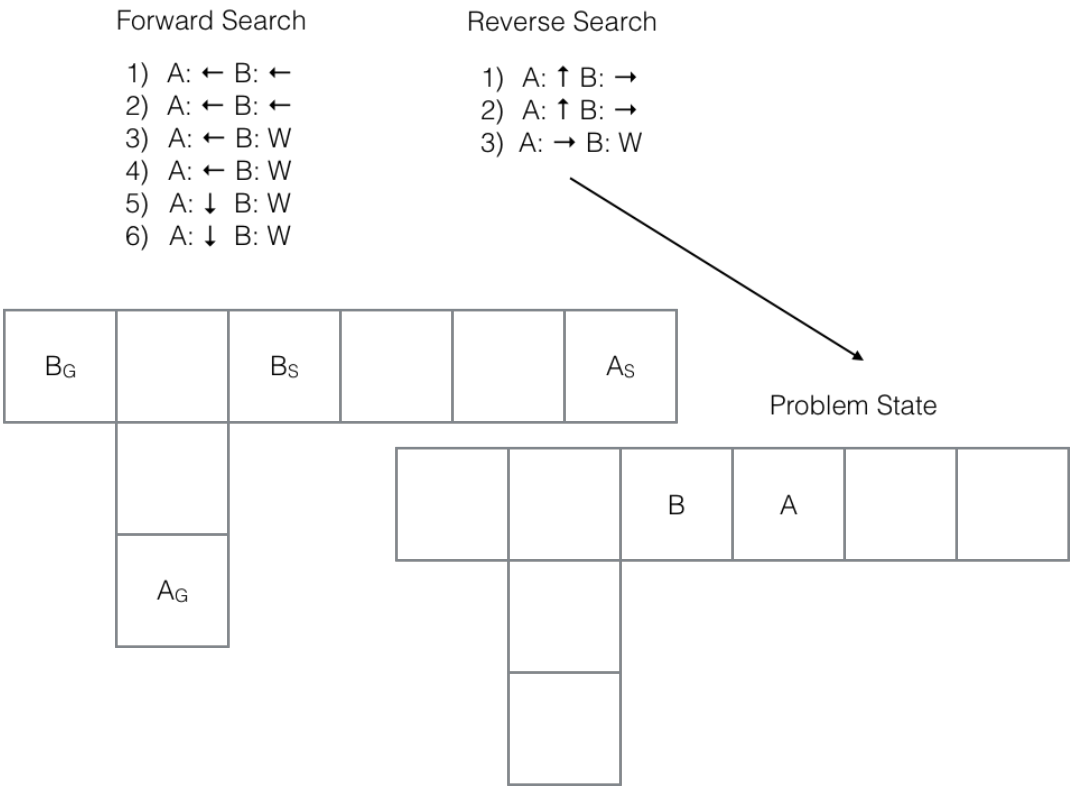

Figure 5.3: Using reverse search to solve a problem may expand search states that the forward search would never need to.

from their start positions A_S and B_S, will likely proceed directly from their start positions to their goals (A_G and B_G) without the need for additional expansions. In a reverse search, however, the search will quickly find itself in the problem state illustrated where B has reached its goal but is now blocking A. The result would be many additional expansions to discover that B needs to wait at its start state until A passes. This means that the reverse search, when used as an abstraction, might spend a lot of time expanding states that aren't needed by the forward search at all; and if that's the case, the additional expansions can considerably increase the search effort rather than decrease it.

The final problem with the Switchback approach has to do with a small optimization available when using g-cost caching. When using g-cost caching, it is possible to use the g-cost of a state on the open list of the abstraction for a heuristic value because the g-cost is an upper bound for the optimal cost of finding that state from the start (g-cost, by definition, is the lowest cost yet found to reach a state). This means when we subtract it from the optimal path cost, we now have a lower bound on the cost of reaching the goal (since the optimal path cost is fixed, and the g-cost can only go down, the difference between the two can only go up). This can save quite a few expansions in the abstraction when we are trying to find a state, especially in the very large state space encountered with MAPF problems with several agents. This optimization is not available for Switchback because we are using the g-cost directly; since the g-cost of a state on the open list can go down, it is not a lower bound, and therefore might not be admissible until it has been expanded and placed on the closed list.

5.4 Independence Detection

Independence Detection (ID) provides for an ideal platform on which to operate with abstraction hierarchies. Since ID already proceeds in a hierarchical manner (each new search is a more constrained version of searches that have already been performed), there is a potential for reuse. To accomplish this, each search that is performed (except those performed to resolve a conflict by constraining the search) is kept in memory permanently for use as a heuristic in larger searches. Whenever two groups are merged within the ID, the searches that correspond to each merged group are used to generate a heuristic value by partitioning the agents into their previous search groups which can then be used to look up the state in the respective previous searches. The values that are found are then combined to create a final heuristic value for the state, and will be reported if it is greater than the baseline

Manhattan distance heuristic. Because each search being performed is a forward search, g-cost caching is used to find heuristic values for all lookups.

Pseudocode for the heuristic lookup is shown in Algorithm 4. Each time a heuristic lookup occurs on a search with more than one agent, this function is called to retrieve the h-cost. The function first obtains the optimal path cost of each abstraction (lines 2-5). The first abstraction has k agents, labeled $a_1...a_k$, which are the agents in the first of the two groups that were merged to create the current search, the other will then have agents $a_k + 1...a_n$ where n is the number of agents in the full problem. These costs have already been determined because each abstraction was part of an optimal A* search in a previous step during the ID algorithm. Next, a function called Partition is called. This function (lines 13-21) simply divides a complete state consisting of the positions of multiple agents into two states, where each state has the positions for the agents that apply to one of the two abstractions. Once these two sub-states have been created, a lookup is performed for each to determine the abstract cost for the corresponding group of agents (lines 7-8). The Lookup function (lines 23-32) performs this lookup by first checking the closed list for the sub-state and returning the difference between the optimal cost of the abstraction and g-cost of the sub-state, if found (line 25). If the sub-state was not found, Lookup expand states until the sub-state is found on the open list[2] and returns the difference between the g-cost of the sub-state and the optimal path cost (lines 27-30). The result will be the abstract cost for the part of the state pertaining to the agents that can be found in the given abstraction. The sum of these two values is the abstract cost. This cost is compared with the Manhattan distance calculated between the current state and goal state (line 9) and the greater of the abstract cost and the Manhattan distance is returned (line 10).

[2]This process can be repeated until the sub-state is found on the closed list to obtain the best result. However, since the g-cost, by definition, can only go down, the returned h-cost is still admissible ($o - g$ can only go up) and we can avoid several expansions which may be costly in abstractions with many agents.

Algorithm 4 Heuristic Lookup

```
1: function HCost(state, goal)
2:     A_0 ← Abstraction for agents a_1...a_k
3:     o_0 ← cost of optimal path in A_0
4:     A_1 ← Abstraction for agents a_{k+1}...a_n
5:     o_1 ← cost of optimal path in A_1
6:     Partition(state, goal, state_0, state_1, goal_0, goal_1)
7:     h_0 ← Lookup(state_0, A_0, o_0)
8:     h_1 ← Lookup(state_1, A_1, o_1)
9:     m ← ManhattanDistance(state, goal)
10:    return Max(h_0 + h_1, m)
11: end function
12:
13: function Partition(state, state_0, state_1)
14:    for all agents in state do
15:        if agent exists in A_0 then
16:            add state[agent] to state_0, goal[agent] to goal_0
17:        else
18:            add state[agent] to state_1, goal[agent] to goal_1
19:        end if
20:    end for
21: end function
22:
23: function Lookup(substate, A, o)
24:    if substate exists in A.closed then
25:        return o − A.closed[substate].g
26:    else
27:        repeat
28:            expand states in A
29:        until substate exists in A.open
30:        return o − A.open[substate].g
31:    end if
32: end function
```

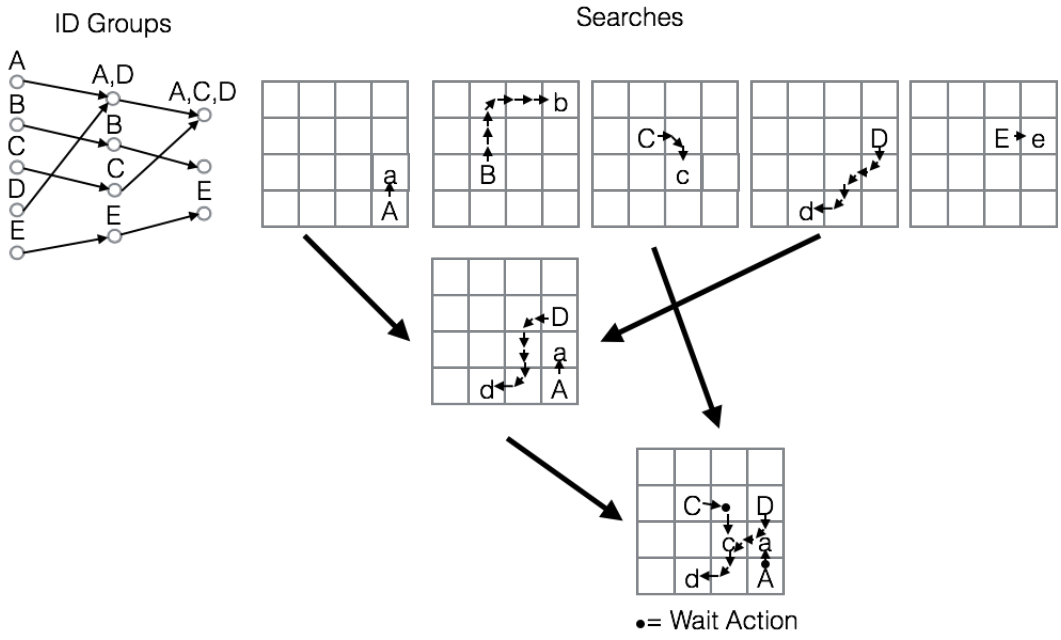

Figure 5.4: Example of the search hierarchy used in Independence Detection.

Figure 5.4 shows how this search hierarchy is created. In the example, agents A and D are merged into a group for search and the previous single-agent searches are maintained to create the heuristic. This process takes place again when the A and D search group is combined with agent C. The multi-agent search performed on agents A and D is now used in conjunction with the single-agent search on agent C to create a heuristic for the full group.

The additional memory cost of abstraction hierarchy is reasonable. A list of prior searches must be maintained in order to perform each new search, but because of the exponential growth of the problem, the memory cost of this process is unlikely to be prohibitive. To illustrate this point more clearly, consider the combination of two groups of size i and j, respectively. If $i = j$ then the new search is 5^i times larger than the either of these searches (since the each new agent creates a search that is 5 times larger than previous searches). This means the amount of space required

to store the previous two searches is only $2/(5^i)$ as much as the current search will likely require. This is not the worst case, however. In the worst case, $i = 1$ or $j = 1$. Assuming, without loss of generality, that $j = 1$, then the combined search is only 5 times larger than the larger of the two searches. This means the previous searches will likely require less than $2/5$ of the space required by the new search. As such, the storage requirement is likely acceptable.

The additional computation time required by using this technique is more problematic. Each state generated during the search will have to be found by searches at each level of the hierarchy. Since each state expansion in a search with multiple agents can generate a huge number of successors, the result can be enormous amounts of search effort in the hierarchy to find heuristic values for generated states that may never be expanded. Even if we stop the search once the abstracted state has entered the open list (rather than waiting for the abstracted state to be expanded), there can be quite a few state expansions in the abstraction to find the state in question.

5.5 Pathmax

One way to reduce the excess effort involved in expanding large amounts of states in the abstraction hierarchy is to use BPMX. Since each abstract search is an already completed search performed previously by ID, we already have an ideal heuristic along that abstract path. This means the abstract start state, and the states near and along the path suggested by the abstract heuristic, all have heuristic values immediately available with no further abstract search expansions. Thus, if we use BPMX to carry these heuristic values forward, we don't have to actually search for new states in the abstraction to get the benefit of the improved heuristic. In fact, since the value of the abstracted heuristic degrades as it gets further from the abstract optimal path (as a result of g-cost caching), the result will likely be

similar, or better, than the fully computed heuristic with no additional abstract search effort.

Chapter 6

Experimental Evaluation

6.1 Independence Detection

To test the efficacy of the hierarchy when combined with Independence Detection, an experiment is devised. Problem instances are created by random on unobstructed grids with no obstacles, and the ID algorithm is first run to completion using a standard Manhattan distance heuristic. Once that is complete, the same problem is run using ID with the search hierarchy in place, using the maximum of the abstraction result and Manhattan distance. This should determine what gains, if any, the abstraction hierarchy offers over search with a much simpler heuristic. States expanded, states generated and time are all tracked to offer insight into the performance of each problem.

6.2 Experimental Conditions

The following experiments were conducted using an implementation of the Independence Detection algorithm performed on an OS X computer with 16 GB of memory and a 2.6 GHz Intel Core i7 processor. Randomized problem instances were generated and run once with the ID algorithm using Manhattan distance as

the only heuristic, and once using the greater of the Manhattan distance and the search hierarchy. Each call to the ID algorithm was limited to 5 minutes of run time. Problems that did not complete within this time constraint using only Manhattan distance were still run with the search hierarchy to determine if the search hierarchy could still complete the problem, however statistics from problem instances completed by one implementation and not the other were not included in any of the averages reported below.

6.3 Results

Average State Expansions on 4x4 Grid				
Agents	Expanded (Initial)	Expanded (Hierarchy)	Improvement	Completed
4	27.9	27.8	0.3%	100
5	49.4	47.5	3.9%	100
6	113.3	105.3	7.1%	100
7	364.2	323.2	11.3%	100
8	565.8	516.6	8.7%	98
9	1,770.5	1,416.9	20.0%	82
10	1,490.8	1,381.9	7.3%	46
11	1,370.0	1,296.3	5.4%	24
12	830.3	725.8	12.6%	6

Table 6.1: Expansion results for 4x4 grids.

Table 6.1 shows the average number of state expansions required to find the goal for problems with differing numbers of agents on a 4x4 grid. 100 randomly generated problems were run for each (not all problems that were generated completed within the 5 minute time limit). There was a reduction of average expansions in each test set when using the abstraction hierarchy, with the best results occurring with the 9 agent problems. In general, most problem instances were solved with the same number of expansions with and without the hierarchy, however when there is a difference, there is often a large savings associated with the use of the hierarchy.

The number of expansions that are required to solve a problem instance reaches its apex at 9 agents, and then decreases. This is unexpected, since the problems should be getting more difficult. The reason for this disparity is the time limit; the hardest problems are no longer being solved within 5 minutes. This is apparent as the number of completed problems drops from 82 to 46 between the 9 and 10 agent problems. It is likely that this is also the reason the best improvements were seen with the 9 agent problems; that is where most of the most difficult problems are being solved within the time limit.

Average State Expansions on 5x5 Grid				
Agents	Expanded (Initial)	Expanded (Hierarchy)	Improvement	Completed
4	26.6	26.6	0.0%	100
5	40.6	40.6	0.0%	100
6	99.4	71.9	27.7%	100
7	148.3	134.5	9.3%	99
8	518.6	461.2	11.1%	100
9	714.0	600.0	16.0%	87
10	650.2	588.9	9.5%	73
11	601.7	580.1	3.6%	45
12	572.6	507.2	11.4%	19

Table 6.2: Expansion results for 5x5 grids.

This trend is repeated in the results on 5x5 grids (Table 6.2). The state expansions required to solve the problems with either technique increased up to 9 agents, then decreased. The heuristic again improved the performance of all but the smallest problems. Notably, there were fewer expansions for most of the problem sets as compared to the 4x4 problems. There are two factors that explain this disparity. First, the larger state space means conflicts are less likely for the same number of agents as compared to the problems on the 4x4 grid. As a result, more of the problems encountered were trivially solved with smaller searches in ID. Second, as we will outline below, the time required to solve a problem is less directly tied to the number of states expanded than the number of states generated. Because of the larger state space, more successor states are generated for each expansion on

average, meaning more time is required. This means that more difficult problems are even less likely to complete in the 5 minute time limit than they were on the 4x4 grid.

Many problems required the exact same number of expansions when using the abstraction hierarchy compared to without. If there is no gain in expansions, then the overhead of using the abstraction hierarchy technique will result in a net loss in performance. One way to look at how the hierarchy affects performance is to find the problems where a difference (either positive or negative) was observed between the two implementations. Table 6.3 shows the results only for those problem instances where the number of expansions differed between the two implementations. The results seem to confirm what we observed before; the hierarchy had the most significant effect on problems of a higher difficulty as measured by expansions (the number of expansions when compared to the overall averages above shows a noticeable increase) and the number of problems where this effect is seen hits its maximum at 9 agents. On average, there is anywhere from a 7% to a 34% savings in state expansions as a result of the use of the abstraction hierarchy for only these problems.

Problems With Differing Results				
Agents	# of Problems	Expanded (Initial)	Expanded (Hierarchy)	Improvement
4	1	71.0	62.0	12.7%
5	4	141.5	93.3	34.1%
6	11	286.0	213.4	25.4%
7	17	1,169.1	927.8	20.6%
8	27	1,152.8	974.5	15.5%
9	36	3,081.2	2,275.7	26.1%
10	19	2375.2	2,111.6	11.1%
11	11	2,068.5	1,907.5	7.8%
12	2	1,328.0	1,014.5	23.6%

Table 6.3: Problems on 4x4 grids without identical expansion results when using the abstraction hierarchy.

In a few rare circumstances, the number of expansions required to solve a problem actually increased the number of expansions. This was because of tie-breaking; the abstraction hierarchy tends to raise the heuristics of states along the abstract optimal path, but sometimes it only raises the f-values to the level of the states around it, leading to a lot of states in the open list with the same f-cost. At this point tie-breaking can lead to good or bad results as a result of different decisions that lead down paths that don't turn out to be optimal. In situations where the two different searches diverge, it seems likely the results would benefit the search hierarchy approach as often as it benefits the search using only Manhattan distance. Table 6.4 shows the results for problem instances where a performance decrease was observed. This happened for no more than 4 problem instances out of any test set and, in the worst case (with the 10 agent set) for only 4 of the 19 problems where a difference in expansions was observed. Therefore, if we assume that tie-breaking is as likely to favor the abstraction hierarchy as not, this still doesn't account for the majority of the noticeable performance improvements in terms of expansions noted above.

4x4 Problems with Performance Decrease for the Hierarchy				
Agents	# of Problems	Expanded(Initial)	Expanded(Hierarchy)	Improvement
4	0	0.0	0.0	0.0%
5	0	0.0	0.0	0.0%
6	0	0.0	0.0	0.0%
7	2	164.5	384.5	-133.7%
8	4	1,100.0	1,388.8	-20.8%
9	4	2,173.3	2,573.3	-18.4%
10	4	1,133.8	1,416.8	-25.0%
11	1	427.0	435.0	-1.9%
12	0	0.0	0.0	0.0%

Table 6.4: Average expansions for problems where performance decreased when using the hierarchy

Some of the problems that were completed by one of the two implementations did not finish under the time limit for the other. This usually happened when ID

was able to solve a problem with the hierarchy that it could not solve without the hierarchy, but occasionally it resulted from a problem that took nearly the entire allotted time without the hierarchy then being unable to complete within the time limit with the added overhead of the hierarchy. Table 6.5 shows the number of problems solved by only one implementation broken down by map size and number of agents. These results seem to follow expectations, with these problems showing up for harder scenarios of 9 agents or more.

Problems Solved By Only One Implementation				
Size	Agents	Initial Only	Hierarchy Only	Both
4x4	4	0	0	100
4x4	5	0	0	100
4x4	6	0	0	100
4x4	7	0	0	100
4x4	8	0	1	98
4x4	9	2	3	82
4x4	10	0	2	46
4x4	11	0	3	24
4x4	12	1	1	6
5x5	4	0	0	100
5x5	5	0	0	100
5x5	6	0	0	100
5x5	7	0	1	99
5x5	8	0	0	100
5x5	9	0	1	87
5x5	10	0	2	73
5x5	11	0	3	45
5x5	12	1	1	19
Total		4	18	1,279

Table 6.5: Generation results for 4x4 grids.

As mentioned above, the time required to complete these problems primarily depends on the time spent generating states and maintaining the open list. As a result, the number of states generated was the primary predictor for the performance of both the uninformed ID search and the search informed by the abstraction hierarchy. Table 6.6 shows the number of states generated for each set of problems on the 4x4 grid.

Average Generated States on 4x4 Grid				
Agents	Generated (Initial)	Generated (Hierarchy)	Improvement	Completed
4	309.9	303.0	2.2%	100
5	2,195.6	1,557.8	29.1%	100
6	33,444.0	28,974.8	13.4%	100
7	506,283.1	456,885.7	9.8%	100
8	1,780,328.2	1,599,195.9	10.2%	98
9	10,253,681.4	7,878,056.7	23.2%	82
10	9,521,856.6	8,683,765.7	8.8%	46
11	8,759,583.6	8,031,484.1	8.3%	24
12	5,824,406.2	5,733,006.2	1.6%	6

Table 6.6: Generation results for 4x4 grids.

Table 6.7 shows the time required for each set of problems on the 4x4 grid. In all cases, any improvements in node generations above are slightly offset by the overhead of the abstraction technique. This overhead comes about from the time necessary to partition each lookup state into two states, and the subsequent hash lookups in the data structures used to store state information in the searches of the abstraction. Note that the overhead does result in a slight decrease in performance for both the 4 and 12 agent problems; these are the problems where the hierarchy resulted in the smallest improvements (around 2%) in terms of state generations. It is possible that improvements to the partition and hash functions of an implementation of the abstraction hierarchy could further reduce this operating overhead, but it is clear that in all but the most trivial problems, there is a significant advantage to using the abstraction technique.

Table 6.8 shows the number of states generated for each problem. Similar to the comparison with state expansions, fewer states are typically generated for problems with the same number of agents as compared to the 4x4 grid.

Table 6.9 shows the timing results for the 5x5 grid problem sets. These results are consistent with the expected overhead cost of performing the abstraction lookups. In this case, however, far more of the problem sets result in very little improvement

Average Time for Search on 4x4 Grid				
Agents	Seconds (Initial)	Seconds (Heuristic)	Improvement	Completed
4	0.00092	0.00098	-6.8%	100
5	0.00714	0.00555	22.3%	100
6	0.12863	0.11892	7.6%	100
7	2.41498	2.28888	5.2%	100
8	9.77897	9.40737	3.8%	98
9	65.57953	53.25150	18.8%	82
10	65.71334	62.95394	4.2%	46
11	73.69389	70.17061	4.8%	24
12	78.22128	80.16969	-2.5%	6

Table 6.7: Timing results for 4x4 grids.

Average Generated States on 5x5 Grid				
Agents	Generated (Initial)	Generated (Heuristic)	Improvement	Completed
4	166.6	166.1	0.3%	100
5	852.2	852.2	0.3%	100
6	94,606.7	26,004.4	72.5%	100
7	215,193.9	179,659.9	16.5%	99
8	3,222,102.7	2,814,070.0	12.7%	100
9	4,568,124.8	3,875,871.1	15.2%	87
10	5,206,031.3	4,940,286.2	5.1%	73
11	4,126,025.9	4,109,119.4	0.4%	45
12	7,169,908.8	7,111,357.4	0.8%	19

Table 6.8: Generation results for 5x5 grids.

when using the abstraction, leading that technique in those problem sets to report a slightly longer average time.

6.4 Summary

The result of using the abstraction hierarchy technique in combination with Independence Detection is generally advantageous primarily for the hardest MAPF problems and can result in average time improvements of up to 20%. The cost of using this technique otherwise is relatively low, but not inconsequential and may be a significant consideration when used in search spaces where conflicts are unlikely. When conflicts are common, however, this technique offers a powerful option that

Average Time for Search on 5x5 Grid				
Agents	Seconds (Initial)	Seconds (Heuristic)	Improvement	Completed
4	0.00051	0.00053	-3.6%	100
5	0.00270	0.00294	-8.9%	100
6	0.33357	0.105	68.7%	100
7	0.91413	0.87505	4.3%	99
8	15.13964	13.96053	7.8%	100
9	24.19194	21.93911	9.3%	87
10	29.33199	29.68583	-1.2%	73
11	26.28467	27.83296	-5.9%	45
12	43.66173	46.89731	-7.4%	19

Table 6.9: Timing results for 5x5 grids.

allows a search with many agents to make use of the search work that has already been performed in previous iterations of the ID algorithm. This technique primarily succeeds in reducing the number of state expansions in the larger searches of the ID hierarchy, but the number of states generated at each step of many-agent problems is still a limiting factor in overall performance.

6.5 Future Work

Search hierarchies may have significant value for this Multi-Agent Pathfinding problems because the problems so easily break down into similar subproblems and because those subproblems will invariably offer useful information about the conflicts which create most of the complexity. The results clearly show that there is value to using this technique with the ID algorithm because the information is already being generated in the early stages of the search and can be used effectively later on. Some work may be required to tune the hierarchy to generate the most possible information without a substantial overhead in terms of time and memory. Furthermore, this paper has only investigated the use of this technique on obstructed, 4-connected grids, but this technique could be used on graphs of any type, and it is likely to offer an improvement in the situations that pathmax works well.

Some possibilities for further algorithmic improvements are to combine search hierarchies with PEA* and EPEA* to reduce the number of states generated, or to use this technique to augment A* when used in conjunction with a technique such as MA-CBS.

6.5.1 PEA* and EPEA*

In order to get the best possible performance, it would be ideal to combine state of the art MAPF search methods. Since PEA* and EPEA* change the method of search without change to the heuristic, they both are likely candidates for combination with MAPF abstraction heuristics. This holds true for PEA* only, however, since EPEA* requires a detailed knowledge of the operation of the heuristic in order to prevent the generation of unneeded states. In order to provide this extra information, a large, precalculated lookup table would likely be required to provide the values necessary for EPEA* to perform its predictions and avoid unnecessary state generations. Because this would seem to be highly expensive in terms of storage space and calculation time, this is not a technique we have attempted to explore at this time.

Bibliography

[1] Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318, 1998.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

[3] Kurt Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *Journal of Artificial Intelligence Research*, 2008.

[4] Ariel Felner, Meir Goldenberg, Guni Sharon, Nathan Sturtevant, Roni Stern, Tal Beja, Jonathan Schaeffer, and Robert C. Holte. Partial-expansion A* with selective node generation. In *AAAI*, 2012.

[5] Ariel Felner, Uzi Zahavi, Jonathan Schaeffer, and Robert C. Holte. Dual lookups in pattern databases. In *IJCAI*, 2005.

[6] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, pages 100–107, July 1968.

[7] Robert C. Holte, Jeffrey Grajkowski, and Brian Tanner. Hierarchical heuristic search revisited. In *SARA*, 2005.

[8] Robert C. Holte, M.B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching Abstraction Hierarchies Efficiently. In *AAAI*, 1996.

[9] Bradford Larsen, Ethan Burns, Wheeler Ruml, and Robert C. Holte. Searching without a heuristic: Efficient use of abstraction. In *AAAI*, pages 114–120, 2010.

[10] Alberto Martelli. On the complexity of admissible search algorithms. *Artificial Intelligence*, 8(1):1–13, 1977.

[11] Lazlo Mero. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23(1):13–27, 1984.

[12] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Conflict-based search for optimal multi-agent pathfinding. In *AAAI*, 2012.

[13] Guni Sharon, Roni Stern, Ariel Felner, and Nathan Sturtevant. Meta-agent conflict-based search for optimal multi-agent path finding. In *SOCS*, 2012.

[14] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. In *IJCAI*, 2011.

[15] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial Intelligence*, 195:470–495, February 2013.

[16] Trevor Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, 2010.

[17] Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Boolean satisfiability approach to optimal multi-agent path finding under the sum of costs objective. In *International Conference on Autonomous Agents & Multiagent Systems*, 2016.

[18] Marco Valtorta. A result on the computational complexity of heuristic estimates for the aalgorithm. *Information Sciences*, 34(1):47–59, 10 1984.

[19] Glenn Wagner and Howie Choset. M*: A complete multirobot path planning algorithm with performance bounds. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3260–3267, 2011.

[20] Thayne T. Walker, David Chan, and Nathan R. Sturtevant. Using hierarchical constraints to avoid conflicts in multi-agent pathfinding. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2017.

[21] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *AAAI*, 2000.