University of Basel

Using Value Abstraction for Optimal Multi-Agent Pathfinding with Increasing Cost Tree Search

Master Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
http://ai.cs.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Gabriele Röger

Simon Wallny
simon.wallny@stud.unibas.ch
2013-055-751

2018-04-11

Abstract

Increasing Cost Tree Search is a promising approach to multi-agent pathfinding problems, but like all approaches it has to deal with a huge number of possible joint paths, growing exponentially with the number of agents. We explore the possibility of reducing this by introducing a value abstraction to the Multi-valued Decision Diagrams used to represent sets of joint paths. To that end we introduce a heat map to heuristically judge how collision-prone agent positions are and present how to use and possible refine abstract positions in order to still find valid paths.

Table of Contents

1 Introduction

In AI, many problems can be described as finding a path through a graph. Multi-Agent Pathfinding is the task of finding such a path for not just one agent, but multiple agents each with their own initial and goal positions traversing the graph concurrently. This introduces the additional constraint of avoiding conflicts between agents, such as occupying the same vertex or traversing the same edge at the same time, though there can be any number of criteria for which actions are considered conflicting in certain contexts. And since the number of possible joint paths grows exponentially with the number of agents involved, this problem can be very hard to solve.

The Increasing Cost Tree Search[1, 2] is an algorithm that guarantees optimal solutions to Multi-Agent Pathfinding problems by essentially subdividing the space of joint paths and searching through the subsets in ascending order of optimality. In doing that it has to handle representations of sets of paths that can be very large. In this work we explore the possibility of applying value abstraction to these representations to reduce their size and improve performance.

In Chapter 2 we go into detail on how ICTS works without any abstractions, and how Multi-valued Decision Diagrams are used to encode sets of paths. This includes the Independence Detection framework which, while not strictly a part of ICTS, is an essential part of an implementation with good performance. Chapter 3 will describe how we build MDDs and the methods of abstraction for them we explore in this work. In Chapter 4 we will see an empirical evaluation of an implementation with these methods and Chapter 5 will conclude and discuss the findings.

Related Work

ICTS was introduced by Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner in 2011[1]. Later that year they introduced pruning techniques[3] which aim reduce the number of possible joint paths by first testing if they work in smaller subgroups and otherwise discarding them for the full problem.

The Independence Detection framework[4] was introduced shortly before in conjunction with A*-based approaches. It is a general framework for MAPF solvers in which agents are subdivided into groups which can be planned for in isolation without conflicting with the other groups.
For a different approach to Optimal Multi-Agent Pathfinding by many of the same authors as ICTS see Conflict Based Search[5]. Unlike ICTS, CBS never plans for groups of multiple agents at once. Instead it plans each agent individually, and, if the resulting joint path has conflicts, replans individual agents with additional constraints to avoid the conflicts as they come up.

2 Background

In this chapter, we define the Multi-Agent Path Finding problem as well as the variant of Multi-valued Decision Diagrams we use to encode sets of paths. Furthermore, we describe the Increasing Cost Tree Search and the Independence Detection Framework.

2.1 Muti-Agent Path Finding

Definition 2.1.1. (MAPF Problem Instance). An Instance of the Multi-Agent Path Finding problem is a tuple $\mathcal{I} = \langle (V, E), A, \mathit{start}, \mathit{goal}, \mathit{conflicts} \rangle$, where

- (V, E) is a directed graph with a set of vertices V and a set of edges $E \subseteq \{(u, v) \mid u, v \in V\}$. The graph includes self loops for all vertives $\forall v \in V : (v, v) \in E$.
- A is a finite set of agents.
- $\mathit{start} : A \to V$ and $\mathit{goal} : A \to V$ are injective functions which assign each agent a start and goal position.
- $\mathit{conflicts} \subseteq \{\{e_1, e_2\} \mid e_1, e_2 \in E\}$ is the subset of unordered edge pairs that are considered conflicting.

The agents are traversing the graph concurrently. At any given time step t each agent $a \in A$ is positioned at one respective vertex $v_t \in V$. Between time steps all agents travel along an edge of the graph to their next position v_{t+1} such that $(v_t, v_{t+1}) \in E$. Since we demand that the graph has self loops for every vertex, waiting in the current position is always a possible move. Once an agent has reached its final position and does not move again, it is considered resting.

A path π of length n for a single agent a_i is a sequence of $n + 1$ vertices $\pi = (v_0, v_1, \ldots, v_n)$ where $\mathit{start}(a_i) = v_0$ and $\mathit{goal}(a_i) = v_n$ and all consecutive vertices (v_t, v_{t+1}) have a transition in E. Furthermore we demand that $v_{n-1} \neq v_n$ meaning that the last transition of the path is not a self loop. We define $\pi(t)$ to denote the vertex v_t of the path if $t \leq n$ or the final resting position v_n of the path if $t > n$. That means that for the length of a path we only consider transitions until the final position is reached, but we expect $\pi(t)$ to give the position of the agent traversing the path even after it has come to rest. This is important to ensure that even resting agents are taken into account when avoiding collisions between agents.

We define two paths π_1, π_2 to be conflicting if $\exists t : \{(\pi_1(t), \pi_1(t+1)), (\pi_2(t), \pi_2(t+1))\} \in$ *conflicts*. That means two paths are conflicting if at any point in time two conflicting transitions occur simultaneously. A tuple of paths is considered conflict-free if no pair of paths in it is conflicting.
For k agents $a_1, a_2, \ldots, a_k$, a solution to an instance of the MAPF problem is a conflict-free k-tuple $\Pi = \langle \pi_1, \ldots, \pi_k \rangle$ of paths for all agents.

While the methods used support a directed Graph, in this thesis we only use undirected graphs, meaning $(u, v) \in E \Leftrightarrow (v, u) \in E$. Specifically, we work with both 4-connected and 8-connected grids whose conflict cases are illustrated in Figure 2.1. In any scenario $\forall (u, v), (u', v) \in E : \{(u, v), (u', v)\} \in$ *conflicts*, meaning that transitions, which place agents on the same position, are always conflicting. For 4-connected grids the only other type of transitions we treat as conflicting are head-on collisions of agents swapping positions; e.g. $\forall (u, v) \in E : \{(u, v), (v, u)\} \in$ *conflicts*. Conversely, moving an agent to a state that is being vacated in the same time step is allowed, even in the case of agents following each other in a cycle. For 8-adjacent grids we additionally consider the possibility that two agents collide by moving over each other diagonally as visualized in Figure 2.1.

While not necessarily true for all definitions of *conflicts*, we notice that in our conflict definition $\{(u, v), (u', v')\} \in$ *conflicts* $\Rightarrow (v, v') \in E$ holds. This means that in order for two transitions to conflict, they must move their agents on neighbouring positions. It should be noted that semantically this is the property that the destination positions are at most one move away from each other, meaning neighbours or themselves. But since we demand self-loops everywhere the criterion of neighbouring is sufficient. We can make use of that property to more efficiently implement conflict detection. Specifically, we can use it to quickly ensure that a set of transitions with destination positions $U \subseteq V$ can not conflict with another set of transitions with destination positions $U' \subseteq V$ if the neighbours of U do not intersect U'. Formally speaking this is $\{n \in V \mid \exists u \in U : (u, n) \in E\} \cap U' = \emptyset$.

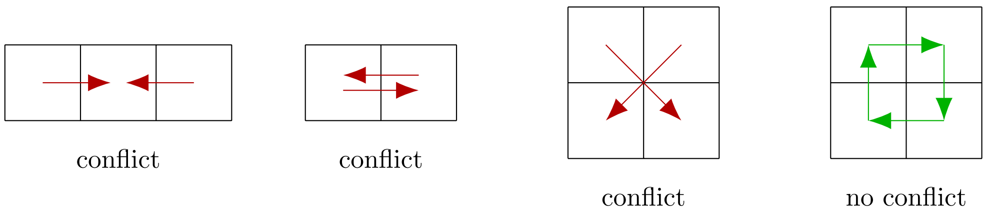


Figure 2.1: Visualisation of which transition pairs are considered conflicting in this thesis.

A solution is considered optimal with respect to a cost function if no solution with a lower cost exists. This cost function can be split in two parts, *path_cost* : $\pi \mapsto c \in \mathbb{R}$ which assigns

costs to individual paths and $joint\_cost : \Pi \mapsto c \in \mathbb{R}$ which assigns costs to joint paths based on the costs of its individual paths.
In this thesis we only work with a unit cost model for transition costs, which gives a cost of one for every transitions in the path. Effectively that means that for a path π of length n, $path\_cost(\pi) = n$. A common alternative is a cost function that only counts transitions (u, v) with $u \neq v$, effectively making it a free action for an agent to wait in place. Rather than counting them as zero or one, in theory we could assign costs based on any arbitrary transition cost function. ICTS is, however, not suited for any cost function in which the length of a path does not equal its cost.

A very commonly used cost function for joint paths is makespan[6–9]. It defines the cost of a joint path as the cost of the most expensive path therein. So for a joint path $\Pi = \langle \pi_1, \ldots, \pi_k \rangle$ the cost is $joint\_cost(\Pi) = \max_{1 \leq i \leq k}(path\_cost(\pi_i))$. With our unit cost model, optimising for makespan effectively means minimising the number of time steps until the entire problem is solved. The joint path cost function used in this thesis however is the sum-of-costs. In this cost function we simply sum up the costs of all individual paths as $joint\_cost(\Pi) = \sum_{i=1}^{k}(path\_cost(\pi_i))$. In our unit cost model, optimising for sum-of-costs means minimising the number of transitions taken across all agents.
While sum-of-costs is most straightforward to use, ICTS can work with makespan as well. And while we work with unweighted agents in this thesis, ICTS could be adjusted to use a cost function which assigns weights to the costs incurred by different agents with relative ease, as elaborate in section 2.2.1.

2.2 Increasing Cost Tree Search

Increasing Cost Tree Search (ICTS) is an optimal multi-agent path finding algorithm for unit-cost cost functions first published in 2011[1, 2]. The general idea in ICTS is to divide the space of potential MAPF solutions (joint paths) into subsets defined by how much cost each individual agent incurs in them. We can then explore the joint path space subset by subset.

Whatever cost function we use in our joint pathfinding problem, be it makespan or a sum-of-cost model, will be solely determined by the costs of the paths that the agents take. Thus, all joint paths contained in a subset have the same cost. A joint path that does not include any conflicts would be a valid solution.

2.2.1 High-Level Search

In order to ensure optimality we must explore these subsets in monotonically ascending order of cost. In other words, when we search through a subset and end up finding a valid solution in it, we want to be sure that no valid solution with a lower cost exists. This process of systematically searching through subsets of joint paths until a valid solution is found is called the **high-level search** of the algorithm.

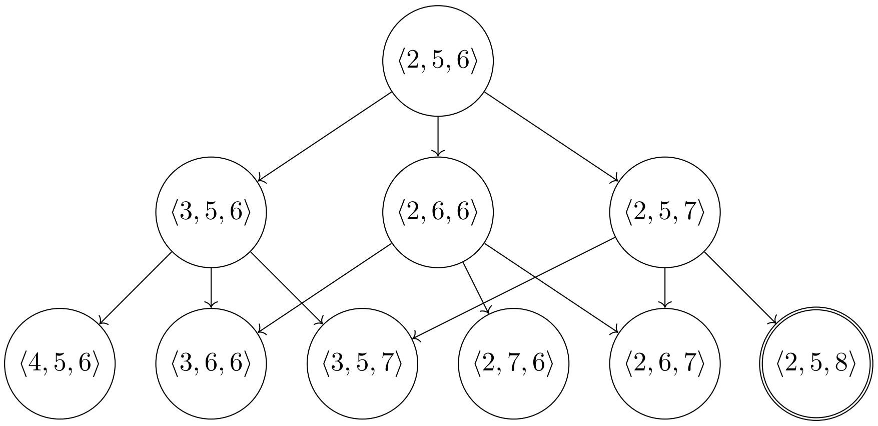


Figure 2.2: Compact representation of an example high-level search tree in which equivalent nodes of the tree are displayed as a single node with multiple parents

In practice this high-level search takes the form of traversing a search tree where each subset is a node. The nodes are identified by a vector of individual agent costs $\langle c_1, \ldots, c_k\rangle$ and represents the set of all joint paths in which every agent a_i reaches the goal in exactly c_i time steps. The children of a node are all nodes which allot exactly one unit cost more for exactly one of their agents. As many different nodes will produce equivalent child nodes, duplicate detection is a necessity in any efficient implementation. For simplicity we consider two nodes with the same cost vector to be identical. This means such a node can have multiple parent nodes, which technically makes the search tree just an acyclic directed graph. We will however continue to call it a search tree in deference to the underlying tree structure and the original authors' decision to name it such.

For a unit cost sum-of-cost cost function with unweighted agents, as we use it, this tree is simply traversed in breadth-first order. In order to optimise for makespan or weighted agents instead we have to explicitly explore the nodes in order of least cost. For that we can simply add the children of each newly explored node to a priority queue sorting by cost and pull the next node to be explored from that queue.

A good starting point to use as the root is the node where all agents incur exactly as much cost as they would in their respective single-agent pathfinding problem. We can be certain that there exists no solution to the joint problem in which an agent takes a shorter path than this, because that shorter path would then have been a better solution to the single-agent pathfinding problem. A node is considered a goal node if its subset of joint paths contains at least one joint path without any conflicts, which is then a solution to the problem instance. In Figure 2.2 we have an example of a high-level search tree for three agents whose respective optimal cost for single-agent pathfinding is 2, 5 and 6 for a sum of 13. In order to be able to reach the goal concurrently without conflict we need to at least incur two additional points for a sum of 15. Specifically, giving both point to the third agent yields a valid solution which is then also optimal with respect to the sum-of-costs.

2.2.2 Low-Level Search

The process of searching through the subset of a high-level node and trying to find a valid solution in it is called the **low-level search** of the algorithm. The low-level search, in the original implementation by Sharon et al.[2] as in the one made for this thesis, is done using a variant of Multi-valued Decision Diagrams (MDDs). MDDs were originally used to encode logical formulas [10], as an extension of the Reduced Ordered Binary Decision Diagrams[11], but for this thesis we simply use its convenient graph structure to encode our sets of paths. Since in this thesis we focus exclusively on unit-cost models we will from this point on consider the cost and length of a path to be the same; the number of time steps it takes until the agent comes to rest at the goal.

These MDDs encode every possible way for a specific agent to reach its goal node with a path of a specific cost (without paying attention to any other agents). This cost has to be met exactly, which means that reaching the destination too soon is just as disqualifying as reaching it too late. As stated in Definition 2.1.1 we assert that a path does not end in a self loop. So it is important to note that waiting in the goal position at the end of the path does not count towards reaching the specified cost.

Definition 2.2.1. (MDD). A Multi-valued Decision Diagram as used here is a tuple $\mathcal{M} = \langle (V, E), P, \mathit{position}, \mathit{layer}, \mathit{depth}, \mathit{root} \rangle$, where

- (V, E) is a directed acyclic graph with vertices V and edges $E \subseteq \{(u, v) \mid u, v \in V; \mathit{layer}(v) = \mathit{layer}(u) + 1\}$.
- P is a set of positions.
- $\mathit{position} : V \to L$ is a function assigning each vertex a position unique for its layer, with the property that for $u \neq v$, $\mathit{layer}(v) = \mathit{layer}(u) \Rightarrow \mathit{position}(v) \neq \mathit{position}(u)$.
- $\mathit{layer} : V \to \{i \in \mathbb{N} \mid 0 \leq i \leq \mathit{depth}\}$ is a function assigning each vertex a layer.
- $\mathit{depth} \in \mathbb{N}$, the number of steps the deepest vertex is away from the root and therefore $\nexists v \in V$ such that $\mathit{layer}(v) > \mathit{depth}$.
- $\mathit{root} \in V$ is the root node, with the property that $\mathit{layer}(v) = 0 \Leftrightarrow v = \mathit{root}$.

When we have a MAPF instance $\mathcal{I} = \langle (V, E), A, \mathit{start}, \mathit{goal}, \mathit{conflicts} \rangle$ for which we wish to encode all the paths an agent a can take with costs c, we build a Multi-valued Decision Diagram $\mathcal{M} = \langle (V', E'), V, \mathit{position}, \mathit{layer}, c, \mathit{root} \rangle$.
The positions assigned to the nodes V' of the MDD are the vertices of the graph V in the MAPF instance, and $\mathit{position}(\mathit{root}) = \mathit{start}(a)$.
The MDD nodes of each layer t represent all positions that the agent can occupy at time step t when it is on any path from $\mathit{start}(a)$ to $\mathit{goal}(a)$ of length c. The transitions to and

from such an MDD node mark all the MAPF vertices from which the agent can come and to which it can go while staying on a path with the required cost. Therefore a transition $(u', v') \in E'$ in the MDD requires a transition $(\textit{position}(u), \textit{position}(v)) \in E$.

In effect that means that a path $\pi = (v_0, v_1, \ldots, v_c)$, $v_i \in V$ with $\textit{start}(a) = v_0$ and $\textit{goal}(a) = v_c$ exists if and only if a respective path $(v'_0, v'_1, \ldots, v'_c)$, $v'_i \in V'$ exists such that $\forall v'_i : \textit{position}(v'_i) = v_i$.

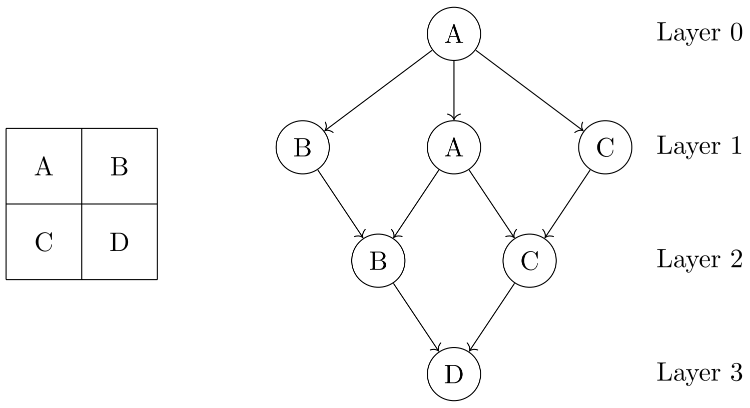


Figure 2.3: MDD (right) of how to get from tile A to tile D in a 2x2 grid (left) in exactly 3 time steps.

An example is given in Figure 2.3 where an agent on a 2x2 map with 4-adjacency tries to reach a diagonal tile in exactly 3 time steps. The tth layer of that MDD contains all tiles that the agent can be in after t time steps have elapsed while following a path that ends at the goal at $t = 3$. For the 0th layer that is only the starting tile and for the 3rd layer that is only the goal tile. Every way to traverse this MDD from top to bottom is a valid path from A to D with cost 3 and conversely every valid path from A to D with cost 3 corresponds to a path through this MDD.

The precise methods we used to construct MDDs to such specifications are elaborated in Section 3.1.

2.2.2.1 Using MDDs

The MDDs only include valid paths and all valid paths are included. However, if two agents try to reach their goals concurrently they might collide, effectively eliminating possible paths. So we define a joint MDD to encode all possible ways for k agents to reach their respective goals with their respective costs without any conflicts between them.

If there were no conflicts between two agents the layers of their joint MDD would be the Cartesian product of the respective layer of the individual MDDs with connections between joint nodes when there was a connection between the nodes of all the respective single agent MDDs. See Figure 2.4.

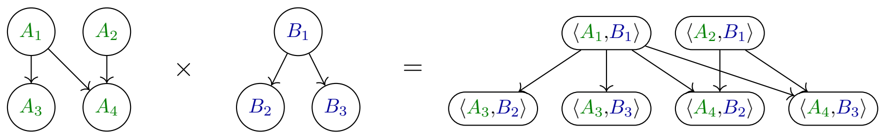


Figure 2.4: Two adjacent layers taken from an example joint MDD with 2 agents (right) and the MDDs of its agents (left) assuming there are no conflicts between the agents.

By taking into account conflicts between agents we eliminate transitions in the joint MDD. Specifically, every joint transition $\langle u, u'\rangle \rightarrow \langle v, v'\rangle$ requires not only that the transitions $u \rightarrow v$ and $u' \rightarrow v'$ are possible on their own ($(u, v), (u', v') \in E$) but also that they do not conflict ($\{(u, v), (u', v')\} \notin$ *conflicts*). Regarding the example in Figure 2.4, if we assume that the transition $B_1 \rightarrow B_2$ conflicts with both $A_1 \rightarrow A_3$ and $A_1 \rightarrow A_4$ we get a sparser MDD as seen in Figure 2.5.
Note that by dropping these transitions the node $\langle A_3,B_2\rangle$ becomes unreachable and disappears from the joint MDD entirely while $\langle A_4,B_2\rangle$ remains reachable through a different transition.

Removing transitions can sever the goal completely from the start. In this case the joint MDD encodes no joint paths which means there is no valid joint paths, in which the individual paths have the exact costs specified in the MDD. This very succinctly answers the question asked in the low-level search of whether or not a valid path for a cost vector $\langle c_1, \ldots, c_k\rangle$ exists. A solution exists if and only if the goal node is reachable in the respective joint MDD.
Joining more than two MDDs works analogously. A set of transitions is conflicting if any pair of conflicts therein is conflicting.

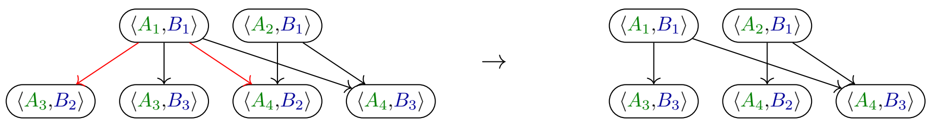


Figure 2.5: By assuming the transition $B_1 \rightarrow B_2$ conflicts with both $A_1 \rightarrow A_3$ and $A_1 \rightarrow A_4$ we lose two transitions in the joint MDD and drop a node in the process.

While it would not be hard to build such a joint MDD explicitly in memory, their size grows exponentially in the number of agents and may become prohibitively large. Sharon et al.[2] therefore decided to employ state-space search methods to search through this space of joint MDD nodes by going from successor to successor.

Every node of the joint MDD search space consists of a tuple of nodes from the individual MDDs, one from each respective MDD and all from the same layer. We can therefore think of the nodes of our joint MDD search space as tuple of pointers to nodes of the individual MDDs. So the start node of the search space is the tuple which points to all the start nodes of the MDDs. The children of a joint node point to all combinations of children of the individual MDDs in the next layer, except for those removed by conflicts.

The joined MDDs are not necessarily of the same length. Once a shorter MDD has reached its goal node it can treat that goal as its own child for subsequent time steps, effectively resting at (and, as far as conflicts are concerned, still occupying) the goal.

We can continue this way, generating children as needed until we reach the joint goal node (where our joint node points at all the goal nodes of the individual MDDs) or we run out of children to generate.

Finding a heuristic to guide the search through the joint MDD search space would be difficult, as the distance from the goal, assuming it is reachable from the current node, is simply the remaining depth of the diagram. So a heuristic would have to be able to accurately predict when a conflict will occur.
As long as we have no reason to prefer one solution over the other the best way to traverse the the search space is a simple depth-first search. In Section 2.3 we will introduce a conflict avoidance table which gives us a reason to prefer certain solutions, in which case we can use a best-first search instead.

As long as we work with regular ICTS without abstraction, the goal reconstruction after a successful search is very easy. During the search we just have to record, for every joint node we traverse, through which predecessor joint node we first reached it. Once we arrive at the joint goal node, we simply follow the chain of predecessors back to the beginning, recording the position of each agent at each time step as we go. These recorded paths then need to be reversed (since we went from goal to start position) and truncated, meaning that shorter paths should end when the agent comes to rest. For path reconstruction with abstract nodes see Section 3.4.

2.3 Independence Detection

The search time of a MAPF task grows exponentially in the number k of agents. Independence Detection is a framework that aims to reduce the effective value of k by splitting agents in independent subgroups. The total search time is then dominated by the largest subgroup of k' agents ($k' \leq k$) While it was first introduced by Trevor Standley in a short paper[4] about solving MAPF tasks optimally with A* it is a general framework that can be implemented on top of any MAPF solver while retaining optimality.

The general idea is that as long as agents in a group do not have conflicts with agents outside their group, the groups can be solved independently and their solutions just run concurrently for a joint solution of the whole problem.

Algorithm 1: Independence Detection (by Standley[4])

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

As seen in Algorithm 1 this is achieved by first assuming that no agents have conflicts. Since in this case they can all take the optimal path they would assume in a single-agent pathfinding problem, we place them into a singleton group each. If the paths of two groups can not be executed concurrently without conflicts, we can first see if there is not another path of equal cost for one of the groups which avoids the conflict. This is not strictly necessary but it avoids unnecessary merging of groups.

We do this by trying to replan for one of the groups while forbidding any moves that would put it in conflict with the other. Only if replanning with the same costs while avoiding collisions with the other group fails for both groups are the two groups merged into one. In order to avoid endless cycles of a group switching back and forth between paths to avoid alternating other groups we never do that reconciliation step for a single pair of groups more than once (line 6). We can furthermore avoid many conflicts in the first place by using a conflict avoidance table to tell the MAPF solver to prefer solutions which put a group in conflict with the fewest other groups.

It depends on the MAPF solver how to make use of the conflict avoidance table. For the A* search, for which it was originally proposed, this takes the form of breaking ties between states with equal f value by taking the state which introduces the least new conflicts.
For ICTS we can can switch the low-level search from depth-first to a best-first search that again looks at states with the least new conflicts first.

It can be said that there are three levels of refinement with the ID framework. The most basic implementation, if we neglect the optional merge-free conflict resolution in lines 6 to 13 as well as conflict avoidance tables this framework easily fits any solver. In order to keep unnecessary merging low by checking for paths of equal cost before merging the solver has to support respecting a set of illegal moves while planning, which should still be very straightforward in any algorithm. And finally, to further refine the framework by avoiding many such unnecessary conflicts in the first place, using a conflict avoidance table, the solver needs to come with an approach custom tailored to the underlying algorithm. In any case the order in which conflicts are addressed will influence the size of the subgroups. We can not guarantee that the k' found will be the smallest possible.

3 MDDs with Value Abstraction

In joint MDDs as described in Section 2.2.2.1 all valid joint paths are encoded, and all joint paths which are encoded in them are also valid. This is certainly a nice property but also a bit excessive. At the end of the algorithm, when a goal is found, we would like to have the actual joint path associated with it. Up until that point, however, we only need the low-level search to be able to differentiate two cases: Either no valid joint path exists or at least one valid joint path exists. The only thing we must guarantee before we allow to high-level search to continue, in order to retain optimality, is that we do not miss the existence of a solution in our currently examined subset of possible solutions. Since the low-level search will only return a found solution exactly once while it may return that no solution was found a potentially very large number of times, it is best to assume that most high-level nodes are not goal nodes. After all, if this assumption turns out to be false, the problem instance was very small anyway.

With that in mind, we could certainly make use of a quick and faulty goal test which will never return false negatives though it might return false positives. If this goal test returns positive we can further refine the test until it returns negative or we have a certain positive. If it returns negative, however, we can cancel the low-level search right there. If our assumption holds that most high-level nodes are not goals and our faulty goal test does a decent job of recognizing these early we can see a significant performance increase.

A way to implement such a false negative free goal test in practice would be via abstract nodes in MDDs. By introducing an over-approximation of some kind, merging together nodes in the joint MDD, we lose the property that all paths encoded in the MDDs are valid, though we would retain the property that all valid paths are encoded. So if we are unable to find a joint path in the abstract joint MDD, there surely is no path. If we do find a path in the abstract joint MDD we can not yet be sure it is actually valid. Using such an abstraction could significantly cut into the size of the joint MDD, which is the largest part of the algorithm that scales exponentially with the number of agents and is thus the likely bottleneck.

As for what nodes of the joint MDD are eligible for joining, merging across layers is not a viable option because of the importance of keeping to the exact costs demanded. Losing the certainty of *where* an agent is is not much of a problem, but losing certainty of *when* an agent is loses all coordination with the other agents.
Merging across agents is more reasonable, but still ill advised, because the algorithm makes heavy use of reusing previous MDDs in new combinations with other MDDs. Every child node of a node in the high-level search tree differs only for one agent, which means that as long as merging happens only on the level of single agents, all other MDDs can be reused.

This leaves us with only one good option. We can use value abstraction of the position of agents by assigning them not just one label, but a disjunction of labels, allowing the agent to be in any of the positions. Since the joint MDD is formed by a cross product of the single-agent MDDs, smaller single-agent MDDs translate nicely into a smaller joint MDD. Halving the number of nodes in one of the single-agent MDDs effectively halves the number of joint MDD nodes.

We have found that the most effective way to make use of these abstract MDDs during a low-level search is to refine the abstractions for that specific search as soon we encounter it during the search, but only if working with it would lead to an unreliable result. That way we sort of roll both the faulty and the reliable tests into one, using the over-approximations as long as we can and only refining them if they yield no satisfactory results.

3.1 Building MDDs

The way we use to build MDDs in this thesis is by first building a distance map for the goal vertex. For every vertex of the graph we calculate the optimal distance from that vertex to the goal vertex. We do this by assigning cost 0 to the goal vertex itself and exploring the graph in a breadth-first manner. It is important to note that for a directed graph we have to use transitions in reverse directions, as we explore paths from the destination to the start. As we in this thesis use undirected graphs this is no particular concern. In Figure 3.1 building a distance map is demonstrated for the toy example from Figure 2.3.

We do not technically have to fill in the entire graph. In order to build an MDD we only need to know the optimal goal distances in places where they are smaller than or equal to the cost of the MDD we want to create. It is, however, easier in practice to just fill a distance map of the entire graph for the goal vertices of every agent and then reuse the complete distance maps for every MDD we build throughout the search.

Once we have the distance map we begin building the MDD. The 0th layer of the MDD just contains a single node for the start vertex of the agent. For every successive layer we go through the previous layer and expand the nodes present therein. We expand such a node by taking its vertex and then consider making a child node in the next layer for every neighbouring vertex it has in the graph. We do, however, only actually include this child if

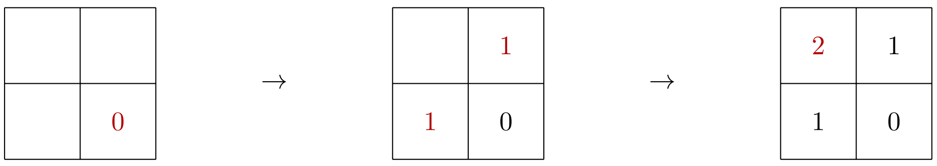


Figure 3.1: Generating a distance map for the toy example from Figure 2.3

the distance map reports that it is possible to reach the goal vertex from the vertex of that child in the remaining number of time steps.

So, in our example we generate the 1st layer of the MDD by taking the only node in the 0th layer, corresponding to vertex A, and looking at all neighbours of A: A, B, and C. Note that A is its own neighbour because we have self loops everywhere in our graph, representing wait actions. Since we want a total cost of 3 and we are already making our first move for 1 cost, the neighbours we accept must be able to reach the goal in at most 2 more steps. According to the distance map in Figure 3.1 this is the case for all three of them, so all three are made nodes in the 1st layer.

We then build the 2nd layer by expanding the 1st. The B node in the 1st layer has 3 neighbours as well: A, B, and D. But since we are already 2 steps in, we must be able to reach the goal in just 1 more step from there. Since node A would take 2 steps it is thus disqualified while B with a goal distance of only 1 is made a child node. A special case is, however, encountered for D. Its goal distance of 0 is below 1, which would normally qualify it. However, it becomes important that a path ends as soon as the agent comes to rest at the goal. If we were to go to D in time step 2 we would not have enough time steps left to go away and come back again (which needs 2 but we only have 1 left). So all we could do then is to stay in node D which makes the effective cost of reaching the goal 2, not 3. An easy way to avoid this scenario is to just categorically forbid the goal vertex to be represented in the second-to-last layer of the MDD.
Then we continue following these expansion rules until we reach the last layer, which will only contain a single node representing the goal vertex.

3.2 Heat Map

The primary guidance used in this thesis to decide which nodes of an MDD to merge is the heat map. The general idea is to rate the importance of specific nodes of the traversed graph for conflict avoidance. Nodes which are on an optimal or near-optimal path for many agents are likely to be the sites of collisions.

The heat map represents that property by assigning a heat value $heat(s)$ to every node s of the graph. Every agent a_i has its own heat signature $heat_i$ contributing to to the total heat value $heat(s) = \sum_i heat_i(s)$ in the heat map. Visualization of a heat signature is given in

Figure 3.2 with the exact values given in Figure 3.3.

If a graph node is on an optimal path for an agent, the heat signature of that agent assigns the node a value of 1. For all graph nodes who, failing that, are on a suboptimal path which is only 1 unit cost more expensive that the optimal one, a heat value of $\frac{1}{2}$ is assigned. This can be broadened further by assigning every graph node for which the best path passing through it is k unit costs above optimal a heat value of 2^{-k}. For the sake of practicality only paths 3 unit costs above optimal are considered by our heat map.

An efficient algorithm for calculating a heat signature for an agent can be implemented by re-using the distance maps calculated for the generation of MDDs as introduced in Section 3.1. We call the start node s_I and the goal node s_G. From the distance map for s_G we get the optimal distance to the goal for every graph node s. In the style of classical planning we call this optimal goal distance $h^*(s)$. We can then start iterating through the graph in a breadth-first manner, starting with s_I. In doing so we keep track of each nodes distance from the start, called $g(s)$. Since we are working with unit cost transitions, the first time any node s is reached, it is reached with optimal cost $g(s)$.

For every new node encountered we calculate the heat value. For that we use that for every node s on an optimal path from s_I to s_G the property $g(s) + h^*(s) = h^*(s_I)$ holds. More generally, $k = g(s) + h^*(s) - h^*(s_I)$ gives us a measure of the suboptimality of a node s in the sense that the best path from s_I through s to s_G is k steps more expensive than the optimal path from s_I to s_G. In following our definition from before this gives s a heat value of 2^{-k}.

Finally, we can define a cut-off point for k, above which we do not consider a graph node noteworthy. We can avoid exploring the area of the graph with k-values above our threshold by simply ignoring neighbours with inadequate k-values when we first reach them during our breadth-first traversal. After all, all nodes most optimally reached through a node with k-value k_1 must have a k-value $k_2 \geq k_1$. Or in other terms, when the shortest path from s_I to s_G through a node s takes c steps, adding another detour through a neighbour of s can not result in a path with less than c steps.
The primary way we make use of the heat map values is to treat all MDD nodes in a layer representing graph nodes with low heat as a single abstract MDD node, trusting the guidance of the heat values that this node is comparatively unlikely to be involved in a conflict. Such a heat threshold should grow with the number of agents, as more agents mean more total heat.

3.3 Refinement

With the merging of nodes in an MDD comes an over-approximation of the represented paths. That means in effect that when plotting a path through an MDD that passes through

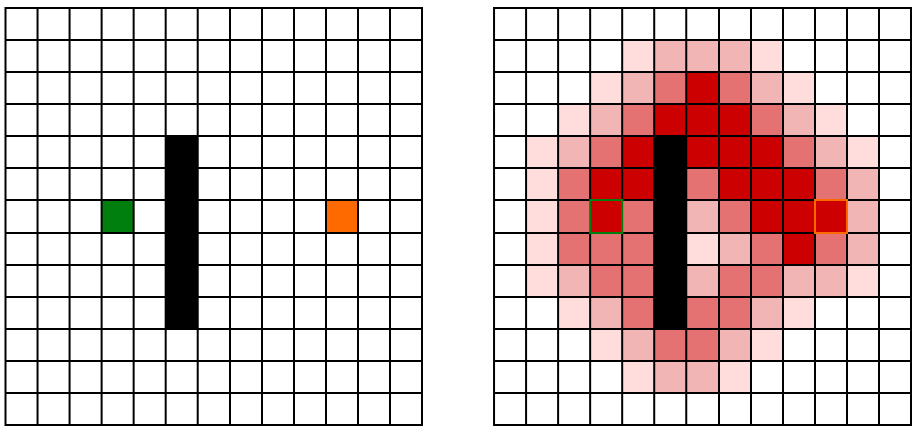


Figure 3.2: Visualization for the heat signature of an agent with starting position marked in green and goal position marked in orange, going around a black obstacle. Exact values given in Figure 3.3

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{8}$	0	0	0	0
0	0	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	**1**	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	0	0	0
0	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	**1**	**1**	**1**	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	0	0
0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	**1**	0	**1**	**1**	**1**	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	0
0	$\frac{1}{8}$	$\frac{1}{2}$	**1**	**1**	0	$\frac{1}{2}$	**1**	**1**	**1**	$\frac{1}{2}$	$\frac{1}{4}$	0
0	$\frac{1}{8}$	$\frac{1}{2}$	**1**	$\frac{1}{2}$	0	$\frac{1}{4}$	$\frac{1}{2}$	**1**	**1**	**1**	$\frac{1}{4}$	0
0	$\frac{1}{8}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	**1**	$\frac{1}{2}$	$\frac{1}{4}$	0
0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{2}$	0	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{8}$	0
0	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	0	0	0
0	0	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	0	0	0	0
0	0	0	0	$\frac{1}{8}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{8}$	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 3.3: The exact values of the heat map from Figure 3.2

an abstract node we can not easily be certain whether or not an actual valid path through the graph corresponds with that sequence of MDD nodes.
In order to determine if, for a given path through an abstract node, there exists a valid path in the graph, we can refine the node – undo the merging – and see if a path through the resulting unabstracted nodes can be found.

This can be done by taking the abstract MDD node $S = \{s_1, s_2, \ldots, s_n\}$ and splitting off a single graph node s_i represented in the abstract node. We then turn $s - I$ into a new concrete node, leaving us with two nodes $S_C = \{s_i\}$ and $S_A = \{s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n\}$. S_A and S_C will each retain a subset of the connections to nodes in the previous and following layer of the MDD that S originally had. If a path through S_C instead of S exists, the refinement was successful in verifying a valid path exists. Should S_C lack the connections needed we can try again, splitting of another graph node from S_A.

A good criterion for selecting the node s_i to split off, is the conflict avoidance table of the Independence Detection framework. By choosing to split off nodes first which can be reached with fewer conflicts in other groups of the framework we can avoid some unnecessary merging of groups. This is, however, not as effective as the conflict avoidance we have without abstractions, and we can expect slightly larger groups to be formed by the independence Detection. Ties can be broken to prefer nodes with smaller heat map values.

This refinement can be repeated as necessary until only unabstracted nodes remain. If by that time no concrete node could fill the place of S in the path we found, we can conclude that the path was a false positive introduced by the over-approximation. Should every path through the abstracted MDD we can find break apart by refining its abstract nodes there is in fact no valid path represented in this MDD.

3.3.1 Refinement in Joint MDDs

When searching through the space of joint paths, represented by joint MDDs, we can reduce the problem to the single-agent MDD variant. A joint MDD node is considered abstract – and any path through it thus unreliable – as long as it contains at least one abstracted position for an agent. Since a joint MDD is really just the combination of multiple single-agent MDDs being traversed concurrently, with conflict rules pruning some of the connections, we can describe a joint MDD node $\mathcal{S} = \langle S_1, S_2, \dots, S_k \rangle$ as a tuple of single-agent MDD nodes. We can refine a joint MDD node by refining the abstract single-agent nodes contained therein.

This too can be done in steps by iteratively splitting off unabstracted nodes from the abstraction. We take one abstract single-agent node S_i in the joint node and split it into S_{i_C} and S_{i_A} as detailed above. We can then in much the same way split the joint node $\mathcal{S} = \langle S_1, \dots, S_i, \dots, S_k \rangle$ into $\mathcal{S}_C = \langle S_1, \dots, S_{i_C}, \dots, S_k \rangle$ and $\mathcal{S}_A = \langle S_1, \dots, S_{i_A}, \dots, S_k \rangle$, each having a subset of the connections $\mathcal{S}$ had to other joint nodes.
Should a joint node contain multiple abstract single-agent nodes this can be applied multiple times to get a completely unabstracted joint node.

3.3.2 Avoiding Unnecessary Refinement

If we just refine every abstract node we come across during our search we will have expended a great deal of computation and gained very little. It is therefore imperative to keep as many nodes as we can abstracted. Abstract nodes treat certain positions as interchangeable. In order to keep that over-approximation the positions have to actually be interchangeable for the purposes of the search.
For one thing, that means that either *all* transitions to this node are conflicting, or *none* are. The case where all transitions conflict is not very useful; detecting and handling it provides no real advantage over refining the node. The case where no conflicts occur, however, is

very interesting. It is very common, since even agents that potentially conflict somewhere will usually spend most of their paths far apart from each other. For this reason we use a simple heuristic to check if an abstract MDD node in a joint MDD node is too far away from the other MDD nodes to cause any conflicts.

Specifically, we use the property of the conflict definition we use on our maps as addressed in Section 2.1: Transitions with destination positions $U \subseteq V$ can not conflict with transitions with destination positions $U' \subseteq V$ if the neighbours of U do not intersect U'. Formally speaking this is $\{n \in V \mid \exists u \in U : (u, n) \in E\} \cap U' = \emptyset$ implies that $\forall (v, u), (v', u')$ with $v, v' \in V$, $u \in U$ and $u' \in U'$ the property $\{(v, u), (v', u')\} \notin$ *conflicts* holds.
In order for a conflict to be possible, the destination positions must be at most one step away from each other. For two grid positions with coordinates (x_1, y_1) and (x_2, y_2) we can cheaply test that as $|x_1 - x_2| + |y_1 - y_2| \leq 1$ for 4-adjacency and $\max(|x_1 - x_2|, |y_1 - y_2|) \leq 1$ for 8-adjacency.
This is, of course, just a heuristic to determine if conflicts are *possible*, it does not mean they are guaranteed. This test does, however, strike an excellent balance of great accuracy while being very cheap to compute.

The other property we need to ensure before using an abstract node without refinement is, that if we use it to find a path through the MDD, this abstract path will include an actual path. See Figure 3.4 for a visualization. The ellipses represent MDD nodes while the numbered squares encode positions. The arrows indicate between which positions actual transitions exist while the MDD nodes have a transition so long as any tile within them is reachable from any tile of the predecessor. Note that while in all three cases an abstract path exists, the left case does not encode an actual path.

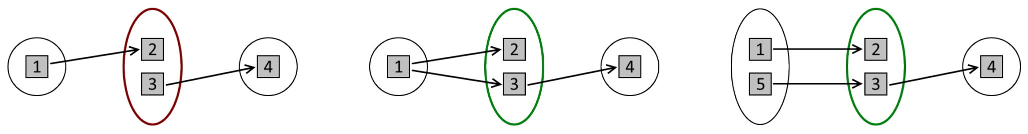


Figure 3.4: Three examples of a short abstract path. Note that the left path does not correspond with any non-abstract path.

We ensure this property be asserting that any abstract node which does not need to be refined has to have a predecessor position in the predecessor node for *every* position it holds. In the example in Figure 3.4 this does not hold for the left case, where the position 3 has no predecessor in the previous node, but it does hold for both the other cases.
This again is just a heuristic. There is no need for every node to be connected as long as at least one is part of an unbroken chain from start to goal. It is, however, very challenging to detect if at least one path exists in the entirety. Detecting if a predecessor node has a predecessor position for each position a node holds, on the other hand, is very easy.

For every MDD node on an abstract path we can observe the following. If at least one position in the node has a valid successor in the next node – which is given in any case since the next node has to have at least one transitions to justify the abstract path – and every position in the node has a valid predecessor in the previous node, then it is always possible to traverse the node with an non-abstract path. Specifically, we can always go from a predecessor position in the previous node to a position in our node, which in turn has a successor position in the next node.

Having this stricter requirement to ensure the existence of non-abstract paths also makes it easier to reconstruct a path at the end of the search, as described in Section 3.4.

3.4 Path Reconstruction with Abstract Nodes

When we attempt to reconstruct the path like we did for purely non-abstract nodes as described in Section 2.2.2.1, we run into the problem that we can now encounter abstract nodes encoding multiple positions. This makes it ambiguous which position to append to the path. If we just take an arbitrary one, we could get paths with jumps in them, where the agent moves between positions not connected by a transition.

But if the assertion holds, that if an abstract node can be used without refinement, every position it encodes has a predecessor position in the predecessor node, we can make use of that. We simply start retracing our steps from the joint goal node to the joint start node as we did before. Now, if we encounter an abstract node we choose one position from it, which is a predecessor position to the position we chose for the same agent in the last step. Since the node we now explore is the predecessor node of the node that supplied our previous pick, we know, thanks to our assertion, that it will always contain such a predecessor position.

This way, we build paths in which each position is guaranteed to be a neighbour of the previous position. Other than that we proceed exactly as we did without abstract nodes, recording the position of each agent at each step, reversing the paths at the end to go from start to goal and truncating paths which ended before the longest path did.

One additional point to consider is how to choose a position from an abstract node if we have multiple options. An arbitrary one will do and result in a valid path. But if we wish to incorporate Independence Detection, we should attempt to choose a path least likely to cause additional conflicts with other groups of the framework. In order to do this we can consult the conflict avoidance table and choose the position that causes the fewest conflicts with the paths of other groups. Ties can be broken in favour of taking the position with the least amount of heat in the heat map.

4 Evaluation

For an empirical evaluation of the concepts presented in this thesis we have implemented an ICTS OMAPF solver with optional value abstraction in C++. The solver was almost completely written from scratch, with the exception of a class to load scenario files (encoding MAPF problem instances) written by Nathan Sturtevant. This class was made available along with the benchmarks[12] for certain maps of the game *Dragon Age: Origins*, which are used in this thesis.

We test our solver on a variety of grid-based maps. Each instance is solved with varying levels of abstraction: No abstraction, light abstraction, medium abstraction, heavy abstraction and total abstraction. In addition we vary the mobility of the agents. We test each problem instance for both the case when agents are allowed to walk diagonally (8-adjacency), and the case when they are not (4-adjacency).

The levels of abstraction are realized by using different thresholds for the heat map. The exception to this is when we use no abstraction at all. For that we aim to implement normal ICTS with Independence Detection as faithfully to the original description as possible. It therefore does not generate a heat map, which would be unnecessary overhead. The other levels of abstraction treat positions as interchangeable if their value in the heat map fails to meet the threshold demanded.

For k agents the thresholds are defined as

- **light abstraction**: $\frac{k}{10}$
- **medium abstraction**: $\frac{k}{5}$
- **heavy abstraction**: $\frac{k}{3}$
- **total abstraction**: ∞

For comparison keep in mind that a position has a heat of 1 for each agent with an optimal path through it and additional fractions for each agent with near-optimal paths. See Section 3.2 for details.

4.1 Test Instances

We try to imitate the tests performed by Sharon et al.[2] for the original solver. The actual instances were unfortunately not available, so we generated our own test instances based on their descriptions. The instances can be grouped into three classes as follows.

8x8 grids with varying number of agents

The first set of tests takes place on a very small, unobstructed map of 8 by 8 tiles. For each problem instance we populate the map with a varying number of agents, ranging from 4 to 16. Each agents in the scenario has a random but unique start and goal position. There are 100 instances each for agent counts of 4, 6, 8, 10, 12, 14, and 16. All instances are solvable.

32x32 grids with varying number of obstacles

The second set of tests takes place on slightly larger map of 32 by 32 tiles. This map comes in six variants, in which a varying percentage of tiles, ranging from 0 to 25 percent, are impassable. The obstacles are randomly distributed.
The maps for 5% obstruction and 25% obstruction can be seen in Figure 4.1. Note how the obstacles in the 5% map are loosely scattered in a way that will require only slight detours from the agents while the 25% map routes agents through numerous small bottlenecks.

For each problem instance we populate the map with 40 agents. Each agents in the scenario has a random but unique start and goal position. There are 100 instances each for obstacle percentages of 0, 5, 10, 15, 20, and 25. All instances are solvable.
The solvability is ensured by solving the single-agent pathfinding problem with 4-adjacency for every agent. If a pathfinding problem is solvable for 4-adjacency it is also solvable for 8-adjacency. Unless the map is designed to deprive agents of any space to manouver around each other, which is not the case for my maps, this single-agent solvability for all agents leads to a solvable joint pathfinding problem.

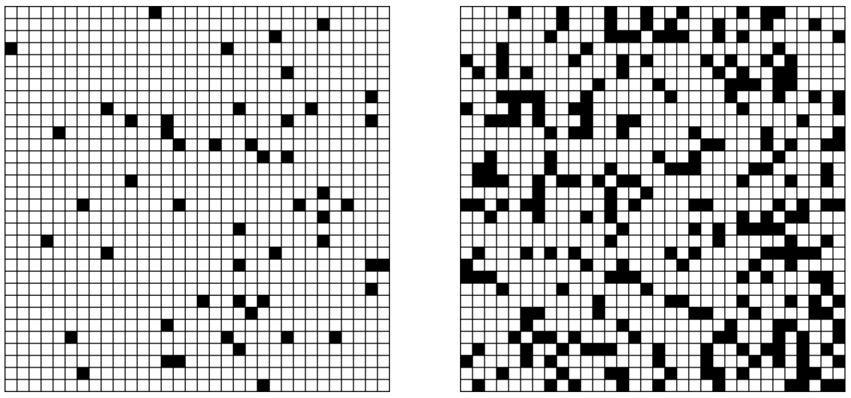


Figure 4.1: The maps used for 5% obstruction (left) and 25% obstruction (right).

Maps from Dragon Age: Origins

The third set takes place on grid representations of maps from the game Dragon Age: Origins. For this we do use the same maps as Sharon et al. made available by Nathan Sturtevant[12] with the permission of the developer BioWare. We do, however, generate our own problem instances on these maps. As for the other instances, we give agents random but unique start and goal positions and ensure solvable instances by performing single-agent pathfinding for all individual agents.

We use three maps, called den520d, ost003d, and brc202d. They can be seen visualized in Figure 4.2. Sharon et al. chose them for their varied topologies, with a maze-like structure in brc202d, multiple open sections connected by small passages in ost003d, and a broad, tube-like structure with multiple branches in den520d.

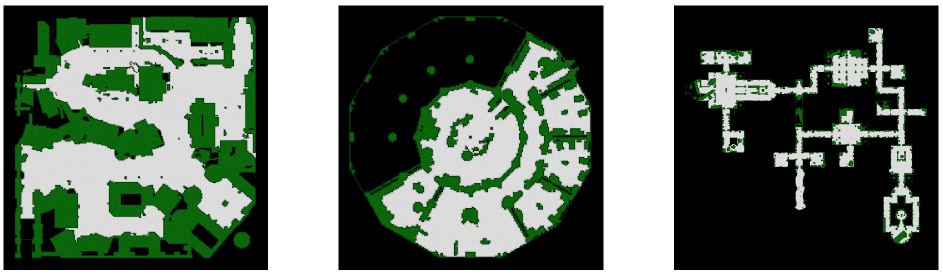


Figure 4.2: The dragon age maps used, den520d (left), ost003d (middle), and brc202d (right).

4.2 Results

In this section we will present and discuss the empirical results of the solver, running on the Slurm Grid of the University of Basel.

4.2.1 8x8 grids with varying number of agents

For 8x8 grids we observe the following average search times in seconds. A table with the exact values is given in Table 4.1. k is the number of agents. The best values in each category are printed in bold. Figure 4.3 provides a visualization of the search times by agent count.

Discussion

The search times for these small but crowded test instances are all over the place. There is no meaningful distinction in the performance of the different abstraction levels. The wide spread of the results is most likely due to the fact that Independence Detection will divide the agents into groups of different sizes depending on what order conflicts are resolved in. That in turn depends on the exact paths found which is influenced by the levels of abstraction which represent paths differently.

k	8-adjacency					4-adjacency				
	none	**light**	**med**	**heavy**	**total**	**none**	**light**	**med**	**heavy**	**total**
4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
6	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.04	0.06	0.06	0.04	**0.03**	0.18	0.18	0.15	0.11	**0.10**
10	0.23	**0.01**	0.03	0.91	0.50	0.46	**0.45**	**0.45**	1.19	0.64
12	7.27	7.38	7.41	**6.68**	6.76	3.91	3.49	3.44	**1.49**	3.70
14	21.30	**16.34**	22.48	17.66	22.08	21.38	16.88	27.95	**15.77**	36.76
16	11.16	**7.80**	14.02	24.71	26.34	15.22	15.28	**11.19**	19.57	16.72

Table 4.1: Search time for 8x8 grids

4.2.2 32x32 grids with varying number of obstacles

For 32x32 obstructed grids we observe the following average search times in seconds. A table with the exact values is given in Table 4.2. *o* is the percentage of tiles which are impassable. The best values in each category are printed in bold. Figure 4.4 provides a visualization of the search times by obstruction percentage.

o	8-adjacency					4-adjacency				
	none	**light**	**med**	**heavy**	**total**	**none**	**light**	**med**	**heavy**	**total**
0	**7.17**	21.47	23.24	36.46	22.19	1.44	4.63	11.54	2.61	**0.82**
5	1.76	**1.66**	18.79	1.68	4.67	**1.47**	7.70	1.67	4.49	3.95
10	**0.62**	1.35	9.53	7.91	7.68	**1.47**	6.60	3.03	4.35	4.35
15	5.53	**4.66**	29.10	32.62	23.47	12.24	**8.20**	17.07	37.89	15.06
20	21.99	**1.43**	7.80	34.46	16.80	**61.06**	61.35	165.01	101.47	84.37
25	**4.02**	18.02	6.54	26.00	4.23	**10.29**	27.92	18.12	15.04	13.89

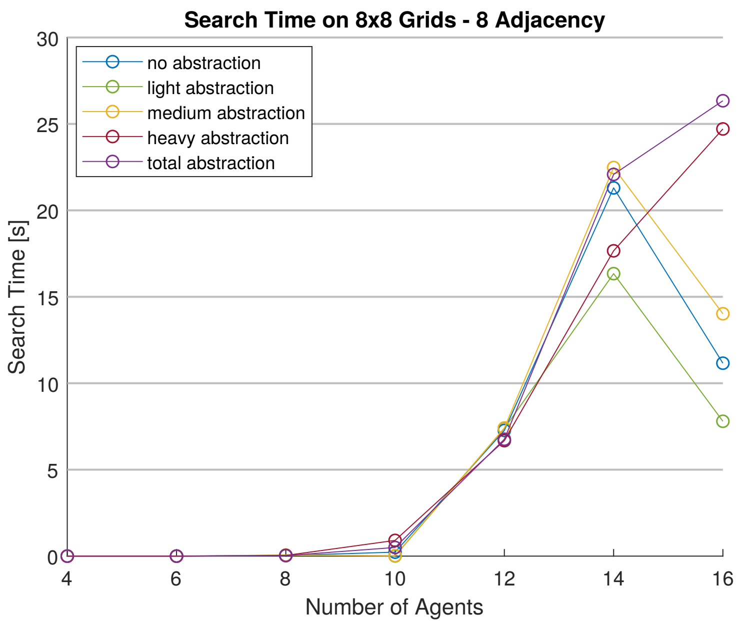
Table 4.2: Search time for obstructed 32x32 grids

Discussion

The search times still vary quite a bit but we can see a trend of abstractions negatively impacting performance. This makes sense given that the maps are crowded and full of bottlenecks, such that the number of positions that are effectively interchangeable is very limited.

We can see that 4-adjacency not only took noticeably longer for more obstructed maps, but the times are also much more strongly determined by the map. The longer search times can be explained by the fact that the random obstacles are much more permeable with 8-adjacency than 4-adjacency, as diagonally adjacent obstacles do not form a wall for 8-adjacency.

The heavy dependence on the map stems from the fact that all instances with the same obstruction percentage use the same map, so a difficult map will affect all instances. This effect is especially strongly pronounced for the map with 20% obstruction. This map in



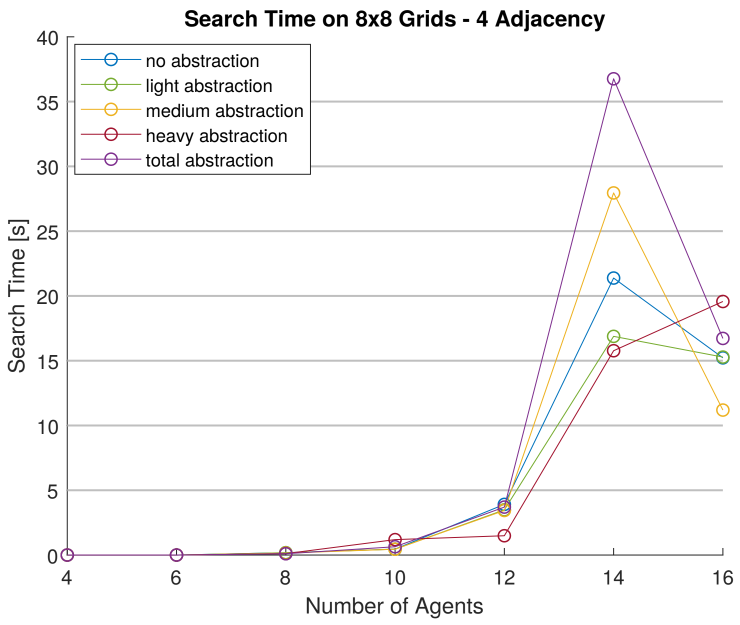
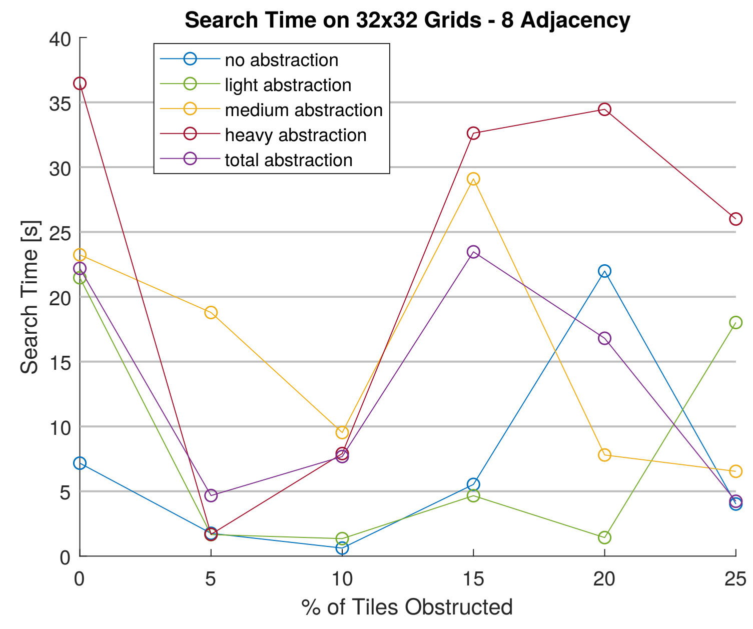


Figure 4.3: Average search time on 8x8 grids with 8-adjacency (top) and 4-adjacency (bottom) by agent count.

particular has many obstacles lining up diagonally in a way that forms a wall in 4-adjacent grids. The exact map used can be seen in Figure 4.5

4.2.3 Maps from Dragon Age: Origins

For the dragon age maps we observe the following average search times in seconds. A table with the exact values is given in Table 4.3. k is the number of agents. The best values in



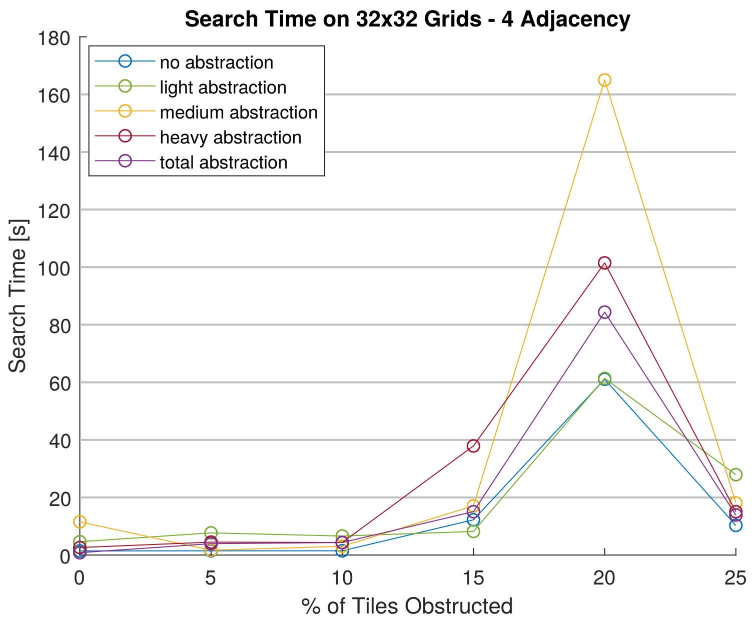


Figure 4.4: Average search time on obstructed 32x32 grids with 8-adjacency (top) and 4-adjacency (bottom) by obstruction percentage.

each category are printed in bold. Figures 4.6, 4.7 and 4.8 provide a visualization of the search times by agent count.

Discussion

For the larger, more expansive maps from the game Dragon Age: Origins seen in Figure 4.2 the abstractions finally begin to shine. With the exception of 8-adjacency on ost003d,

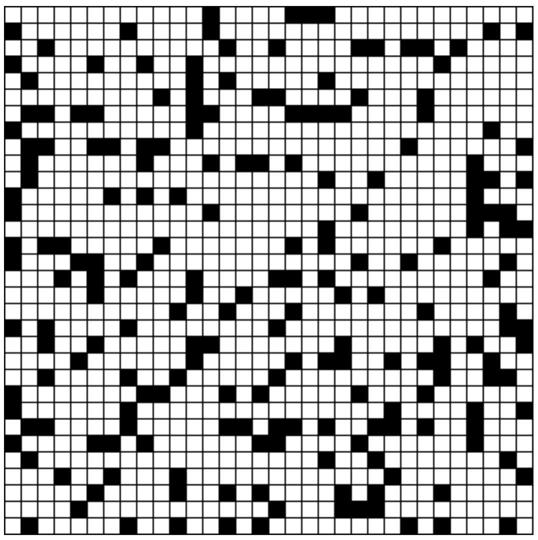


Figure 4.5: The map used for 32x32 grids with 20% obstruction.

heavier abstraction tend to outperform lighter ones, especially for 4-adjacency. This can be explained by the fact that on these large maps agents spend much larger portions of their paths not in direct proximity to other agents. In addition, crossing an empty space diagonally with 4-adjacency allows for huge amounts of possible paths of equal length, which can be efficiently abstracted. This helps explain why ost003 is very partial to abstractions for 4-adjacency but not so much for 8-adjacency.

4.3 Summary

In Table 4.4 we have the arithmetic mean of the search times across all test instances. These averages are heavily dominated by the large problems from the dragon age maps, as they are where the computation spent most of its time. Due to this fact, in the total average the heaviest abstractions outperformed the lighter ones by a large margin. It should also be noted that while we can see a direct correlation of abstraction strength to performance in all cases where abstraction was used, using no abstraction at all always gives a solid performance. From the experimental results given in this chapter it seems clear that using light abstractions is not a good idea in general, but that total abstraction can substantially speed up search time for suitable problem instances.

map	**k**	8-adjacency				
		none	**light**	**med**	**heavy**	**total**
den520d	10	**1.11**	1.18	1.39	1.42	1.40
den520d	20	40.15	46.08	42.76	25.18	**12.49**
den520d	30	87.67	100.47	53.86	45.81	**19.65**
den520d	40	257.27	259.10	187.11	144.07	**55.61**
den520d	50	200.34	208.29	249.73	**152.45**	172.89
ost003d	10	**0.51**	0.54	0.56	0.55	0.53
ost003d	20	3.86	3.63	3.88	6.49	**2.90**
ost003d	30	13.99	15.61	**13.20**	27.70	26.42
brc202d	5	**0.82**	0.89	0.89	0.89	0.89
brc202d	10	3.44	3.59	4.40	3.10	**2.31**
brc202d	15	**3.17**	4.20	4.29	4.17	3.94
brc202d	20	**4.89**	7.09	6.85	7.00	6.22
map	**k**	4-adjacency				
		none	**light**	**med**	**heavy**	**total**
den520d	10	4.21	4.38	4.66	**3.24**	3.62
den520d	20	3.95	4.21	3.89	2.78	**2.60**
den520d	30	**4.42**	5.19	5.38	6.85	5.63
den520d	40	20.57	20.01	19.00	**18.87**	19.14
den520d	50	84.20	74.23	55.17	**44.89**	47.89
ost003d	10	**0.85**	0.89	0.86	0.97	0.93
ost003d	20	32.03	32.09	27.29	35.10	**15.36**
ost003d	30	47.57	48.64	43.25	56.55	**21.70**
brc202d	5	**0.56**	0.61	0.62	0.67	0.68
brc202d	10	2.74	2.85	3.05	2.99	**2.33**
brc202d	15	38.82	28.94	23.29	20.84	**20.36**
brc202d	20	51.21	49.36	40.07	32.97	**22.08**

Table 4.3: Search time for dragon age maps.

In particular it appears that large maps sparsely populated by agents are well suited for abstraction while small, crowded maps with many bottlenecks are very ill-suited. And since possible overhead of even the total abstraction, which requires a lot of refinement steps, is not all that large, it seems that using value-abstracted MDDs is a solid improvement in general.

Adjacency	**none**	**light**	**med**	**heavy**	**total**
8-adjacency	27.93	29.23	28.32	24.32	**17.60**
4-adjacency	16.81	16.96	19.45	17.23	**13.71**

Table 4.4: Average search time across all instances.

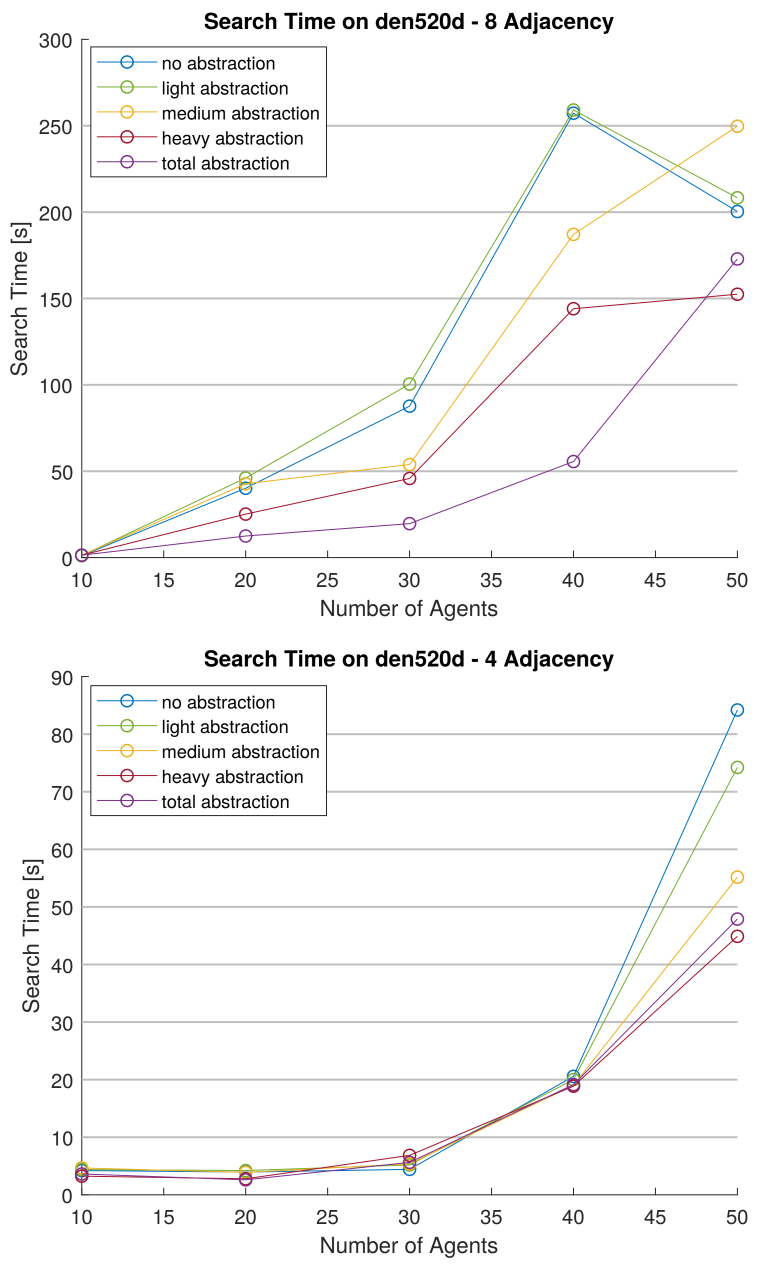
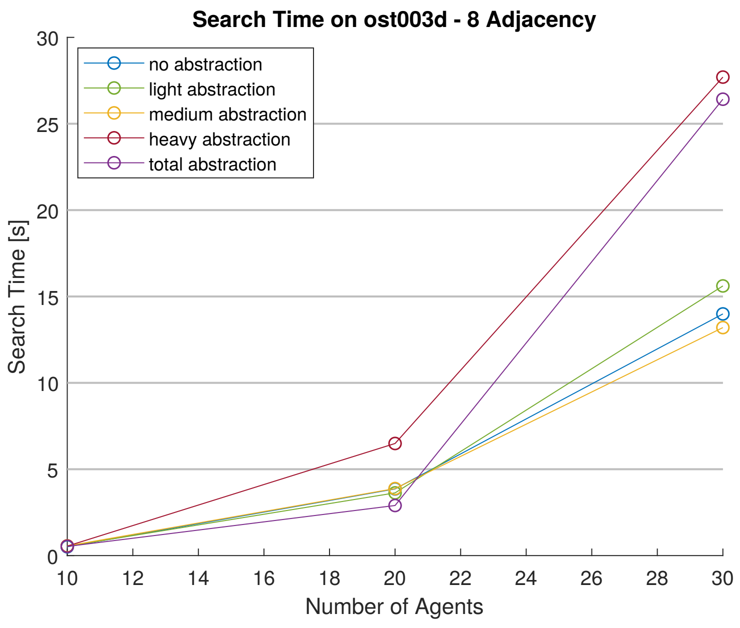


Figure 4.6: Average search time on Dragon Age: Origins map den520d with 8-adjacency (top) and 4-adjacency (bottom) by agent count.



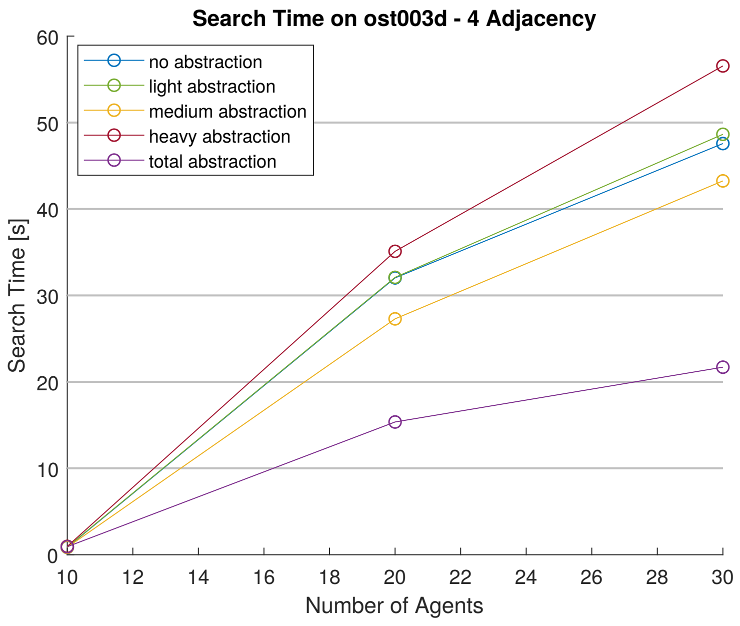


Figure 4.7: Average search time on Dragon Age: Origins map ost003d with 8-adjacency (top) and 4-adjacency (bottom) by agent count.

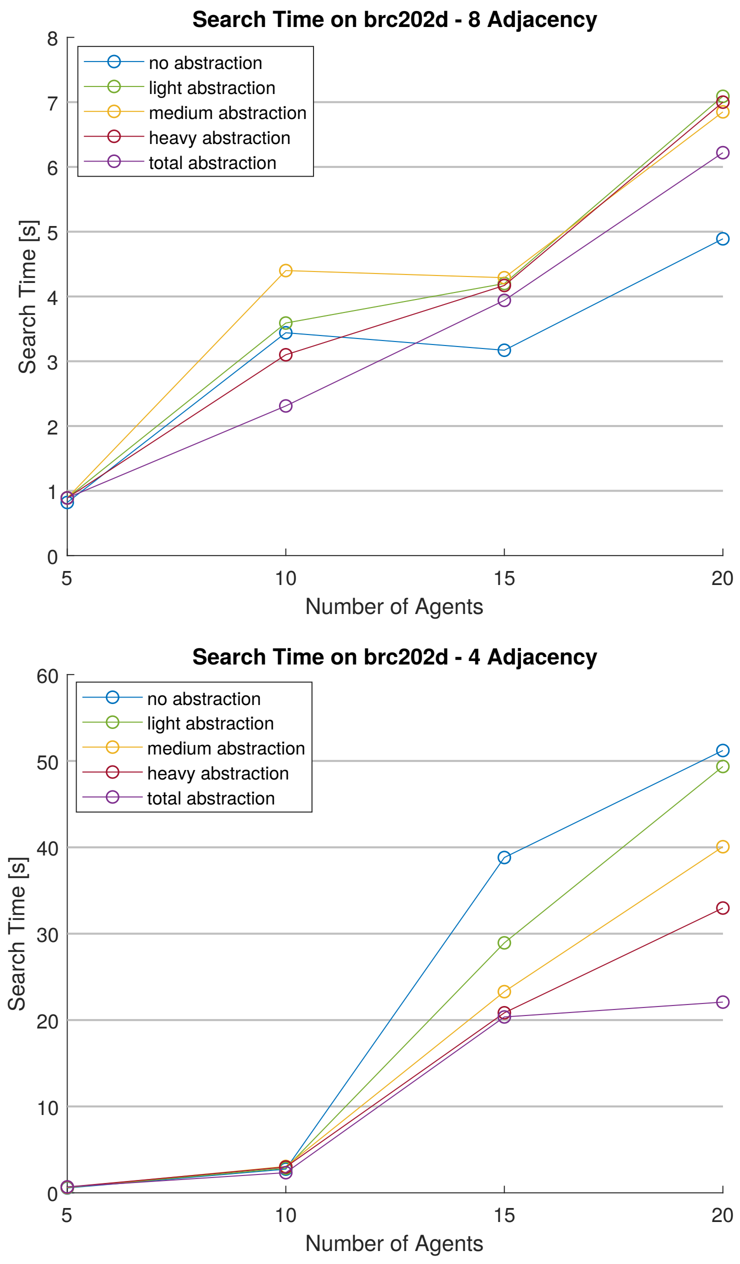


Figure 4.8: Average search time on Dragon Age: Origins map brc202d with 8-adjacency (top) and 4-adjacency (bottom) by agent count.

5 Conclusion

In this Thesis we introduced value abstraction to the Multi-valued Decision Diagrams used to encode sets of paths for the low-level search of the Increasing Cost Tree Search algorithm. We introduced the heat map as a way to heuristically judge how likely a position is to be the site of a collision by rating how many agents are expected to want to pass through it. We used this heat map to merge MDD nodes representing positions with a low heat value and also used it to guide our search by preferring tiles with low heat when given an otherwise equal choice between positions.
Furthermore, we presented an algorithm to refine abstractions again, allowing us to extract reliable information from over-approximated representations. Importantly we also gave a criterion used to determine when abstract nodes are allowed to remain abstract even if they are used in a path to the goal and how we can still reconstruct the paths then.

We implemented all these concepts in C++ and gathered empirical results on 4-adjacent and 8-adjacent grids with different maps and agent configurations. From these results we can judge that abstraction can speed up search time by a large margin, especially for expansive or sparsely populated maps. A total abstraction, where by default all nodes in the same layer of an MDD are merged into one, seems particularly promising. An implementation which does not explicitly build non-abstract MDDs but instead starts with total abstraction and refines as necessary to shape all MDDs seems like a solid improvement, on average, over using MDDs without any abstraction.

5.1 Future Work

Having worked with and personally implemented ICTS search, we judge that the two most important factors, for which there is still a lot of room for improvement, are reducing the number of refinements needed and ensuring smaller groups in Independence Detection.

Reducing the number of refinement steps could take multiple forms. Finding stronger heuristics to judge when a node is safe to leave unrefined is certainly one. We use cheap but pretty accurate heuristics, but even so we will sill refine some nodes for being too close to another

node, even though there was no actual conflict, or for not having a predecessor for every position even though in the end at least one path through the node would have worked. A less demanding criterion for position predecessors that does not require one for every position would also require a more involved path reconstruction. But since path reconstruction is only used once per solver call, a fairly expensive one would be easy to justify.
In addition one could try to pick which nodes to refine more effectively. If we could accurately predict which node best to refine into which smaller nodes, to achieve nodes that need no further refinement and bring us closer to the goal would help tremendously.

As for the Independence Detection, it can be difficult to influence just how agents will be grouped up. But making effective use of the conflict avoidance table is an easy way to start. In our implementation we used the table to decide between different positions for both refinement (splitting off positions with fewer conflicts first, breaking ties with heat map) and during path reconstruction (picking positions with fewer conflicts and heat to use for the path). But this is certainly not the full extend to which the the conflict avoidance table could be utilized, especially since execution time scales exponentially with the number of agents in a group. With that in mind we could even justify a rather substantial drop in performance (for fixed group size) if in exchange it gets us smaller groups.

And finally, one could very well try to use abstraction of things other than the positions. In Chapter 3 we have given reasons why that might not work too well, but there might well be ways to make them work.

Bibliography

[1] Sharon, G., Stern, R., Goldenberg, M., and Felner, A. The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI'11)*, pages 662–667 (2011).

[2] Sharon, G., Stern, R., Goldenberg, M., and Felner, A. The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding. *Artificial Intelligence*, 195:470–495 (2013).

[3] Sharon, G., Stern, R. T., Goldenberg, M., and Felner, A. Pruning Techniques for the Increasing Cost Tree Search for Optimal Multi-agent Pathfinding. In *Proceedings of the Fourth Annual Symposium on Combinatorial Search (SoCS'11)*, pages 150–157 (2011).

[4] Standley, T. S. Finding Optimal Solutions to Cooperative Pathfinding Problems. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI'10)*, volume 1, pages 28–29 (2010).

[5] Sharon, G., Stern, R., Felner, A., and Sturtevant, N. R. Conflict-based Search for Optimal Multi-agent Pathfinding. *Artificial Intelligence*, 219:40–66 (2015).

[6] Rajendran, C. and Ziegler, H. Ant-colony Algorithms for Permutation Flowshop Scheduling to Minimize Makespan/Total Flowtime of Jobs. *European Journal of Operational Research*, 155(2):426–438 (2004).

[7] Tasgetiren, M. F., Liang, Y.-C., Sevkli, M., and Gencyilmaz, G. A Particle Swarm Optimization Algorithm for Makespan and Total Flowtime Minimization in the Permutation Flowshop Sequencing Problem. *European journal of operational research*, 177(3):1930–1947 (2007).

[8] Surynek, P. Compact Representations of Cooperative Path-Dinding as SAT based on Matchings in Bipartite Graphs. In *Proceedings of Tools with Artificial Intelligence (ICTAI)*, pages 875–882 (2014).

[9] Surynek, P. Makespan Optimal Solving of Cooperative Path-Finding via Reductions to Propositional Satisfiability. *arXiv preprint arXiv:1610.05452* (2016).

[10] Miller, D. M. Multiple-valued Logic Design Tools. In *Proceedings of The Twenty-Third International Symposium on Multiple-Valued Logic*, pages 2–11 (1993).

[11] Bryant, R. E. Graph-based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 100(8):677–691 (1986).

[12] Sturtevant, N. Benchmarks for Grid-Based Pathfinding. *Transactions on Computational Intelligence and AI in Games*, 4(2):144 – 148 (2012).