Pruning Techniques for the Increasing Cost Tree Search for Optimal Multi-Agent Pathfinding

Guni Sharon **Roni Stern** **Meir Goldenberg** **Ariel Felner**
Information Systems Engineering, Ben-Gurion University
Be'er-Sheva, Israel
{gunisharon,roni.stern}@gmail.com, mgoldenbe@yahoo.ca, felner@bgu.ac.il

Abstract

We address the problem of optimal path finding for multiple agents where agents must not collide and their total travel cost should be minimized. Previous work used traditional single-agent search variants of the A* algorithm. In (Sharon *et al.* 2011) we introduced a novel two-level search algorithm framework for this problem. The high-level searches a novel search tree called *increasing cost tree* (ICT). The low-level performs a goal test on each ICT node. The new framework, called ICT search (ICTS), showed to run faster than the previous state-of-the-art A* approach by up to three orders of magnitude in many cases. In this paper we focus on the low-level of ICTS which performs the goal test. We introduce a number of optional pruning techniques that can significantly speed up the goal test. We discuss these pruning techniques and provide supporting experimental results.

Introduction

The *multi-agent path finding* (MAPF) problem consists of a graph and a number of agents. For each agent, a path is needed from its initial location to its destination without colliding into obstacles or other moving agents. The task is to minimize a cumulative cost function (e.g., total time steps). MAPF has practical applications in robotics, video games, vehicle routing etc. (Silver 2005; Dresner and Stone 2008). In its general form, MAPF is NP-complete, because it is a generalization of the sliding tile puzzle which is known to be NP-complete (Ratner and Warrnuth 1986).

Previous work on MAPF falls into two classes. The first is called the *decoupled approach* where paths are planned for each agent separately. A prominent example is HCA* (Silver 2005). Agents are ordered in some order. The path found for agent a_i (location and time) is written (*reserved*) into a global *reservation table*. To resolve conflicts, search for successive agents must avoid locations and time points that were reserved by previous agents. A similar approach was used for guiding cars that need to cross traffic junctions (Dresner and Stone 2008). Other decoupled approaches establish flow restrictions similar to traffic laws, directing agents at a given location to move only in a designated direction (Wang and Botea 2008; Jansen and Sturtevant 2008). Decoupled approaches run relatively fast, but optimality and even completeness are not always guaranteed.

The focus of this paper is on the second class of methods for solving MAPF called the *global search approach*, where MAPF is formalized as a global, single-agent search problem (Ryan 2008; Standley 2010) and is solved by an A*-based search. Global searches usually return the optimal solution but may run for a long time.

In (Sharon *et al.* 2011) we introduced a two-level framework that optimally solves MAPF. The high-level performs a search on a new search tree called *increasing cost tree* (ICT). Each node in the ICT consists of a k-vector $\{C_1, C_2, \dots C_k\}$ which represents *all* possible solutions in which the cost of the individual path of each agent a_i is exactly C_i. The low-level performs a goal test on each of these tree nodes. We denote our 2-level algorithm as *ICT-search* (ICTS). Experimental results on a number of domains showed that ICTS outperforms the previous state-of-the-art A* approach by up to three orders of magnitude in many cases.

In this paper we continue this line of research and focus on a number of *pruning techniques* that can be optionally activated before the low-level. These techniques were very briefly discussed in (Sharon *et al.* 2011). A *successful* pruning proves that a given ICT node n is not the goal, In such case n is immediately declared as non-goal and the high-level moves to the next ICT node without the need to perform the low-level search on n. If all pruning *failed*, we must activate the more heavy low-level search on n in order to confirm whether n is goal or not. The tradeoffs between the different pruning techniques are discussed in this paper and supporting experimental results are provided.

Problem definition and terminology

We define our variant of MAPF which is commonly used and some basic terminology. Nevertheless, most algorithms (including our own) work for other existing variants too.

Input: The *input* to MAPF is: **(1)** A graph $G(V, E)$. **(2)** k agents labeled $a_1, a_2 \dots a_k$. Every agent a_i is coupled with a start and goal vertices - $start_i$ and $goal_i$.

Initially, (at time t_0) every agent a_i is located in location $start_i$. Between successive time points, each agent can perform a *move* action to a neighboring location or can *wait* (stay idle) at its current location. The main constraint is that each vertex can be occupied by at most one agent at a given time. In addition, if a and b are neighboring vertices, different agents cannot simultaneously traverse the connecting edge in opposite directions (from a to b and from b to a). A *conflict* is a case where one of the constraints is violated. We allow agents to *follow* each other, i.e., agent a_i could move from x to y if at the same time, agent a_j moves from y to z.

The task is to find a sequence of $\{move, wait\}$ actions for each agent such that each agent will be located in its goal position while aiming to minimize a global cost function.

Cost function: We use the common cost function which is the summation (over all agents) of the number of time steps required to reach the goal location (Dresner and Stone 2008; Standley 2010). Therefore, both *move* and *wait* actions cost 1.0. We denote the cost of the optimal solution by C^*. Figure 2(i) (on page 3) shows an example 2-agent MAPF problem. Agent a_1 has to go from a to f while agent a_2 has to go from b to d. Both agents have a path of length 2. However, these paths conflict, as both of them have state c at the same time point. One of these agents must wait one time step or take a detour. Therefore, $C^* = 5$ in our case.[1]

Previous work on optimal solution

Previous work on optimal MAPF formalized the problem as a global single-agent search as follows. The *states* are the different ways to place k agents into $|V|$ vertices without conflicts. At the start (goal) state agent a_i is located at vertex $start_i$ ($goal_i$). *Operators* between states are all the non-conflicting actions (including *wait*) that all agents have. Let b_{base} be the branching factor for a single agent. The global branching factor is $b = O((b_{base})^k)$. All $(b_{base})^k$ combinations of actions are considered and only those with no conflicts are *legal* neighbors. Any A*-based algorithm can then be used to solve the problem. (Ryan 2008; 2010) exploited special structures of local neighborhoods (such as stacks, halls and cliques) to reduce the search space.

Standley's improvements: Recently, (Standley 2010) suggested two improvements for solving MAPF with A*:

(1) Operator Decomposition (OD): OD aims at reducing b. This is done by introducing *intermediate states* between the regular states. *Intermediate states* are generated by applying an operator to a single agent only. This helps in pruning misleading directions at the intermediate stage without considering moves of all of the agents (in a regular state).

(2) Independence Detection (ID): Two groups of agents are *independent* if there is an optimal solution for each group s.t. the two solutions do not conflict. The basic idea of ID is to divide the agents into *independent* groups. Initially each agent is placed in its own group. Shortest paths are found for each group separately. The resulting paths of all groups are simultaneously performed until a conflict occurs between two (or more) groups. Then, all agents in the conflicting groups are unified into a new group. Whenever a new group of $k \geq 1$ agents is formed, this new k-agent problem is solved optimally by an A*-based search. This process is repeated until no conflicts between groups occur. Standley observed that since the problem is exponential in k, the A*-search of the largest group dominates the running time of solving the entire problem, as all other searches involve smaller groups (see (Standley 2010) for more details on ID).

For the A* search Standley used a common heuristic function which we denote as the *sum of individual costs* heuristic (SIC). For each agent a_i we assume that no other agents exist and *precalculate* its optimal individual path cost. We then sum these costs. The SIC heuristic for the problem in Figure 2(i) is $2 + 2 = 4$.

Standley compared his algorithm (A*+OD+ID) to a basic implementation of A* and showed spectacular speedups. The ID framework might be relevant for other algorithms that solve MAPF. It is relevant for ICTS and we ran ICTS on top of the ID framework as detailed below.

We now turn to present our two-level ICTS algorithm.

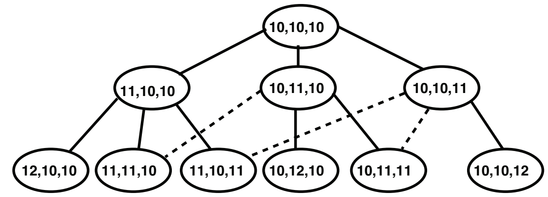

Figure 1: ICT for three agents.

High-level: *increasing cost tree* (ICT)

The classic global search approach spans a search tree based on the possible locations of each of the agents. Our new formalization is conceptually different. It is based on the understanding that a *complete solution* for the entire problem is built from *individual paths*, one for each agent. We introduce a new search tree called the *increasing cost tree* (ICT). In ICT, every node s consists of a k-vector of individual path costs, $s = [C_1, C_2, \ldots C_k]$ one cost per agent. Node s represents *all* possible complete solutions in which the cost of the individual path of agent a_i is exactly C_i.

The root of ICT is $[opt_1, opt_2, ..., opt_k]$, where opt_i is the cost of the optimal individual path for agent i which assumes that no other agents exist. A child is generated by adding a unit cost to one of the agents. An ICT node $[C_1, .., C_k]$ is a *goal* node if there is a non-conflicting complete solution such that the cost of the individual path for agent a_i is exactly C_i. Figure 1 shows an example of an ICT with three agents, all with individual optimal path costs of 10. Dashed lines lead to duplicate children which can be pruned. The total cost of node s is $C_1 + C_2 + \ldots + C_k$. For the root this is exactly the SIC heuristic of the start state, i.e., $SIC(start) = opt_i + opt_2 + \ldots opt_k$. Nodes of the same level of ICT have the same total cost. It is easy to see that a breadth-first search of ICT will find the optimal solution, given a goal test function.

The depth of the optimal goal node in ICT is denoted by Δ. Δ equals the difference between the cost of the optimal complete solution (C^*) and the cost of the root (i.e., $\Delta = C^* - (opt_i + opt_2 + \ldots opt_k)$). The branching factor of ICT is exactly k (before pruning duplicates) and therefore the number of nodes in ICT is $O(k^{\Delta})$.[2] Thus, the size of ICT is

[1] Another possible cost function is the total time elapsed until the last agent reaches its destination. This would be 3 in our case. Also, one might only consider *move* actions but not *wait* actions. The tile puzzles are an example for this.

[2] More accurately, the exact number of nodes at level i in the

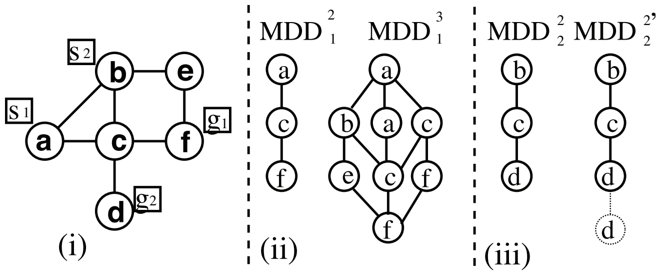

Figure 2: (i) 2-agent problem (ii) MDD_1 (iii) MDD_2

exponential in Δ but not in k. For example, problems where the agents can reach their goal without conflicts will have $\Delta = 0$, regardless of the number of agents.

The high-level searches the ICT with breadth-first search. For each node, the low-level determines whether it is a goal.

Low-level: goal test on an ICT node

A general approach to check whether an ICT node $s = [C_1, C_2, \dots, C_k]$ is a goal would be: **(1)** For every agent a_i, enumerate all the possible individual paths with cost C_i. **(2)** Iterate over all possible ways to combine individual paths with these costs until a complete solution is found. Next, we introduce an effective algorithm for doing this.

Compact paths representation with MDDs

The number of different paths of length C_i for agent a_i can be exponential. We suggest to store these paths in a special compact data structure called *multi-value decision diagram* (MDD) (Srinivasan *et al.* 1990). MDDs are DAGs which generalize *Binary Decision Diagrams* (BDDs) by allowing more than two choices for every decision node. Let MDD_i^c be the MDD for agent a_i which stores all possible paths of cost c. MDD_i^c has a single *source node* at level 0 and a single *sink node* at level c. Every node at depth t of MDD_i^c corresponds to a possible location of a_i at time t, that is on a path of cost c from $start_i$ to $goal_i$.

Figure 2(ii,iii) illustrates MDD_1^2 and MDD_1^3 for agent a_1, and MDD_2^2 for agent a_2. Note that while the number of paths of cost c might be exponential in c, the number of nodes of MDD_i^c is at most $|V| \times c$. For example, MDD_1^3 includes 5 possible different paths of cost 3. Building the MDD is very easy. We perform a breadth-first search from the start location of agent a_i down to depth c and only store the partial DAG which starts at $start(i)$ and ends at $goal(i)$ at depth c. Furthermore, MDD_i^c can be reused to build MDD_i^{c+1}. We use the term $MDD_i^c(x,t)$ to denote the node in MDD_i^c that corresponds to location x at time t. We use the term MDD_i when the depth of the MDD is not important for the discussion.

Goal test with MDDs. A goal test is now performed as follows. For every ICT node we build the corresponding MDD for each of the agents. Then, we need to find a set of paths, one from each MDD that do not conflict with each other. For our example, the high-level starts with the root ICT node $[2,2]$. MDD_1^2 and MDD_2^2 have a conflict as they both have state c at level 1. The ICT root node is therefore declared as non-goal by the low-level. Next, the high-level tries ICT node $[3,2]$. Now MDD_1^3 and MDD_2^2 have non-conflicting complete solutions. For example, $< a-b-c-f >$ for a_1 and $< b,c,d >$ for a_2. Therefore, this node is declared as a goal node by the low level and the solution cost 5 is returned.

Next, we present an efficient algorithm that iterates over the MDDs to find whether a non-conflicting set of paths exist. We begin with two agents and then generalize to $k > 2$.

ICT is the number of ways to distribute i balls (actions) to k ordered buckets (agents). For the entire ICT this is $\sum_{i=0}^{\Delta}\binom{k+i-1}{k-1}$.

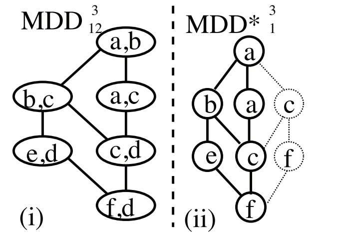

Figure 3: (i) MDD_{12} (ii) unfolded $MDD*_1^3$.

2-agent MDD and its search space

Consider two agents a_i and a_j located in their start positions. Define the *global 2-agent search space* as the state space spanned by moving these two agents simultaneously to all possible directions as in any centralized A*-based search. Now consider their MDDs, MDD_i^c and MDD_j^d, which correspond to a given ICT node $[c,d]$.[3]

The cross product of the MDDs spans a 2-agent search space or equivalently, a *2-agent*-MDD denoted as MDD_{ij} for agents a_i and a_j. MDD_{ij} is a *2-agent search space* which is a subset of the global 2-agent search space, because we are constrained to only consider moves according to edges of the single agent MDDs and cannot go in any possible direction.

MDD_{ij} is formally defined as follows. A node $n = MDD_{ij}([x_i,x_j],t)$ includes a pair of locations $[x_i,x_j]$ for a_i and a_j at time t. It is a unification of the two MDD nodes $MDD_i(x_i,t)$ and $MDD_j(x_j,t)$. The source node $MDD_{ij}([x_i,x_j],0)$ is the unification of the two source nodes $MDD_i(x_i,0)$ and $MDD_j(x_j,0)$. Consider node $MDD_{ij}([x_i,x_j],t)$. The cross product of the children of $MDD_i(x_i,t)$ and $MDD_j(x_j,t)$ should be examined and only non-conflicting pairs are added as its children. In other words, we look at all pair of nodes $MDD_i(\bar{x_i},t+1)$ and $MDD_j(\bar{x_j},t+1)$ such that $\bar{x_i}$ and $\bar{x_k}$ are children of x_i and x_j, respectively. If $\bar{x_i}$ and $\bar{x_j}$ do not conflict[4] then $MDD_{ij}([\bar{x_i},\bar{x_j}],t+1)$ becomes a child of

[3]Without loss of generality we can assume that $c = d$. Otherwise, if $c > d$ a path of $(c-d)$ *dummy goal nodes* can be added to the sink node of MDD_j^d to get an equivalent MDD, MDD_j^c. Figure 2(iii) also shows $MDD_2^{2'}$ where a dummy edge (with node d) was added to the sink node of MDD_2^2.

[4]They conflict if $\bar{x_i} = \bar{x_j}$ or if ($x_i = \bar{x_j}$ and $x_j = \bar{x_i}$), in which case they are traversing the same edge in an opposite direction.

Algorithm 1: The ICT-search algorithm

Input: (k, n) MAPF
1 Build the root of the ICT
2 **foreach** *ICT node in a breadth-first manner* **do**
3 **foreach** *agent* a_i **do**
4 Build the corresponding MDD_i
5 **end**
6 **[** perform subset pruning //***optional***
7 **if** *pruning was successful* **then**
8 break *//Conflict found. Next ICT node*
9 **end**
10 **]** search the k-agent MDD *// low-level search*
11 **if** *goal node was found* **then**
12 **return** Solution
13 **end**
14 **end**

$MDD_{ij}([x_i, x_j], t)$ in MDD_{ij}. There are at most $|V|$ nodes for each level t in the single agent MDDs. Thus, the size of the 2-agent-MDD of height c is at most $c \times |V|^2$.

One can actually build and store MDD_{ij} by performing a search (e.g. breadth-first search) over the two single agent MDDs and unifying the relevant nodes. Duplicate nodes at level t can be merged into one copy but we must add an edge for each parent at level $t-1$. Figure 3(i) shows how MDD_1^3 and $MDD_2^{2'}$ were merged into a 2-agent-MDD, MDD_{12}^3.

Low-level search. Only one node exists at level c (the MDD height) - $MDD_{ij}^c([goal_i, goal_j], c)$. A path to it is a solution to the 2-agent problem. A goal test for an ICT node therefore performs a search on the search space associated with MDD_{ij}^c. This search is called the *low level search*. Once a node at level c is found, $true$ is returned. If the entire search space of MDD_{ij}^c was scanned and no node at level c exists, $false$ is returned. This means that there is no way to unify two paths from the two MDDs, and deadends were reached in MDD_{ij}^c before arriving at level c.

Generalization for $k > 2$ is straightforward. A node in a k-agent-MDD, $n = MDD_{[k]}(x[k], t)$, includes k locations of the k agents at time t in the vector $x[k]$. It is a unification of k non-conflicting single-agent MDD nodes of level t. The size of $MDD_{[k]}$ is $O(c \times |V|^k)$. The low-level search is performed on the search space associated with $MDD_{[k]}$.

ICTS is summarized in Algorithm 1. The high-level searches each node of the ICT (Line 2). Then, the low-level searches the corresponding k-agent MDD search space (Lines 10 -14). Lines in square brackets (6-9) are optional. These are the pruning techniques which are the main focus of this paper and are discussed below.

Search choices

The high-level search is done with breadth-first search. However, any complete search algorithm can be activated by the low-level search on the k-agent MDD search space. We tried many variants but only report results where the low level search was performed by DFS with transpositions table for pruning duplicates. In the case that a solution exists (and the corresponding ICT node will be declared as the goal) this DFS variant will find the solution fast, especially if many such solutions exist.

Similarly, like A*, ICTS in all our experiments was also built on top of the ID framework. That is, the general framework of ID was activated (see above). When a group of conflicting agents was formed, A* or ICTS was activated.

k	Ins	A*	ICTS
2	50	0	0
4	50	0	0
6	50	2	0
8	50	98	2
10	47	5,888	57
12	42	8,142	499

Table 1: Runtime in ms. in a 8x8 grid

Comparison between A* and ICTS

In (Sharon *et al.* 2011) we performed a systematic comparison between ICTS and the state-of-the-art A* variant of A*+OD+ID (Standley 2010). We provided theoretical explanation on the benefits of ICTS and explained why it runs faster. Let X be the number of nodes expanded by A*, i.e., X is the number of nodes with $f \leq C^*$. The main finding was that the number of nodes visited by the low-level of ICTS is $O(X \times k^{\Delta})$ while the number of nodes generated by A* is $O(X \times (b_{base})^k)$. ICTS clearly outperforms A* in cases where $k^{\Delta} < (b_{base})^k$. We provided worst-case examples where $k^{\Delta} > (b_{base})^k$. However, in most of our experiments ICTS outperformed A*.

Table 1 shows an example experiment from (Sharon *et al.* 2011) on a 4-connected 8x8 grid with no obstacles where we varied the number of agents from 2 to 12 (the k column). The *Ins* column shows the number of instances (out of 50) solved by both algorithm under 5 minutes. The next columns are the running times in ms for A*+ID+OD and for ICTS+ID. ICTS clearly outperforms A* in all these cases. More experimental results on other grids and on maps from (Sturtevant 2010) were presented in (Sharon *et al.* 2011) and similar tendencies were observed.

Since thorough comparison to A* was already provided in (Sharon *et al.* 2011) in the rest of this paper we explore further speedups for ICTS and only provide experimental results within the framework of ICTS.

Pruning techniques

We now turn to discuss a number of useful pruning methods that can be optionally activated before the low-level on an ICT node n. This is shown in lines 6-9 of Algorithm 1. If the pruning was *successful*, n can be immediately declared as non-goal and there is no need to activate the low-level on n. The high-level jumps to the next ICT node. We begin with the *simple pairwise pruning* and then describe enhancements as well as generalize these techniques to pruning techniques that consider groups with more than two agents. We then provide experimental results and discuss these methods.

Algorithm 2: Pairwise pruning in ICT node n

1 **foreach** *pair of agents* a_i *and* a_j **do**
2   Search MDD_{ij} with DFS
3   **if** *solution found* **then**
4   continue *// next pair*
5   **end**
6   **if** *solution not found* **then**
7   return SUCCESS *// Next ICT node*
8   **end**
9 **end**
10 return FAILURE *// Activate low level on n*

Simple pairwise pruning

As shown above, the low-level search for k agents is exponential in k. However, in many cases, we can avoid the low-level search by first considering subproblems of pairs of agents. Consider a k-agent MAPF and a corresponding ICT node $n = \{C_1, C_2, \ldots C_k\}$. Now, consider the abstract problem of only moving a pair of agents a_i and a_j from their start locations to their goal locations at costs C_i and C_j, while ignoring the existence of other agents. Solving this problem is actually searching the *2-agent search space* of MDD_{ij}. If no solution exists to this 2-agent problem (i.e., searching MDD_{ij} will not reach a goal node), then there is an immediate benefit for the original k-agent problem as this ICT node (n) can be declared as non-goal right away. There is no need to further perform the low-level search through the k-agent MDD search space. This variant is called *Simple pairwise pruning* (SPP).

SPP, presented in Algorithm 2 is optional and can be performed just before the low-level search. SPP iterates over all pairs (i, j) and searches the 2-agent search space of MDD_{ij}. If a pair of MDDs with no pairwise solution is found (Line 6), $Success$ is returned. The given ICT node is immediately declared as a non-goal and the high-level moves to the next ICT node. Otherwise, if pairwise solutions were found for all pairs of MDDs, $failure$ is returned (Line 10) and the low-level search through the search space of the k-agent MDD of the given ICT node n must be activated. SPP is performed with a DFS on MDD_{ij} (line 2). The reason is again that a solution for the 2-agent problem can be found rather fast, especially if many solutions exist. In this case, this particular pair of agents cannot prune the current ICT node n. The pruning phase quickly moves to the next pair of agents and tries to perform pruning for the new pair on node n. There are $O(k^2)$ prunings in the worst case where all pairwise searches resolved in a 2-agent solution and *failure* is returned.

Enhanced pairwise punning

With some modification, pairwise pruning can produce valuable benefits even in this worst case. This is done by changing the search strategy of the pairwise pruning from DFS to breadth-first search and adding a number of steps that modify the single-agent MDDs, MDD_i and MDD_j as follows. Assume that MDD_{ij} was built by unifying MDD_i and MDD_j. We can now unfold MDD_{ij} back into two single agent MDDs, $MDD{*}_i$ and $MDD{*}_j$ which are sparser than the original MDDs. $MDD{*}_i$ only includes paths that do not conflict with MDD_j (and vice versa). In other words, $MDD{*}_i$ only includes nodes that were actually unified with at least one node of MDD_j. Nodes from MDD_i that were not unified at all, are called *invalid* nodes and can be deleted.

Figure 3(ii) shows $MDD{*}_1^3$ after it was unfolded from MDD_{12}^3. Light items correspond to parts of the original MDD that were pruned. Node c in the right path of MDD_1^3 is invalid as it was not unified with any node of MDD_2^3. Thus, this node, its incident edges and its only descendent (f) can be removed and are not included in $MDD{*}_1^3$.

Enhanced pairwise pruning (EPP) deletes invalid nodes from each individual MDD while performing the pairwise pruning search through the entire search space of MDD_{ij}. Unlike SPP, EPP searches the MDD_{ij} in a breadth-first search manner. For each level t the following actions are performed. Each node $MDD_i(x, t)$ is kept in the sparser $MDD{*}_i$ if there exists at least one node $MDD_j(y, t)$ such that MDD_{ij} includes the node $MDD_{ij}([x, y], t)$. Otherwise, $MDD_i(x, t)$ is pruned and not included in $MDD{*}_i$. Similarly, $MDD{*}_j$ is generated in the same way.

The entire MDD_{ij} is searched with the breadth-first search even if a solution was found. This is done in order to have the the sparser MDDs $MDD{*}_i$ and $MDD{*}_j$ fully available. These sparser MDDs can be now used in the following two cases:

(1) Further pairwise pruning. After $MDD{*}_i$ was obtained, it is used for the next pairwise check of agent a_i. Sparser MDDs will perform more ICT node pruning with other MDDs as they have a smaller number of options for unifying nodes and higher chances of declaring an ICT node as a non-goal. Furthermore, when $MDD{*}_i$ is matched with MDD_k, it might prune more portions of MDD_k than if the original MDD_i was used. This has a *cascading effect* such that pruning of MDDs occurs through a chain of MDDs.

(2) The general k-agent low-level search. This has a great benefit as the sparse MDDs will span a smaller k-agent search space for the low-level than the original MDDs.

Repeated enhanced pairwise pruning

If all $O(k^2)$ pairs were matched and a solution was found for every pair then the ICT node cannot yet be declared as a non-goal and the low-level should be activated. Assume a pair of agents a_i and a_j such that a solution was found in MDD_{ij} when the agents were first matched. However, now, after all the mutual pruning of single-agent MDDs, the resulting $MDD{*}_i$ and $MDD{*}_j$ are much sparser. Repeating this process might reveal that now, the new sparser MDDs can no longer be unified and that the previous solution no longer exists. The *Repeated enhanced pairwise pruning* (REPP) repeatedly performs iterations of EPP. In each iteration, EPP matches all $O(k^2)$ pairs and repeatedly makes the single-agent MDDs sparser. This is continued until either the ICT node is pruned (because there exist a pair a_i and a_i such that there is no solution to $MDD{*}_{ij}$) or until no single-agent MDD can be made sparser by further pairwise pruning.

Tradeoffs

Natural tradeoffs exist between the different pairwise pruning. Per pair of agents, SPP is the fastest because as soon

k	k'	Δ	Ins	NP	2S	2E	2RE	3S	3E	3RE
4x4 grid										
4	1.9	0.5	50	0.7	0.5	0.1	0.0	0.4	0.0	0.0
5	2.1	0.5	50	1.2	0.9	0.5	0.0	0.3	0.0	0.0
6	3.2	1.1	50	5.4	2.7	0.2	0.0	0.3	0.1	0.0
7	4.4	1.8	50	18.9	9.1	2.0	0.0	2.8	0.6	0.0
8	6.1	3.3	50	579.7	299.1	121.7	0.2	56.6	5.1	0.1
9	7.0	4.3	47	2,098.5	741.4	66.2	0.0	264.3	5.8	0.0
10	7.8	5.2	39	3,500.9	2,200.5	116.5	0.3	488.7	11.4	0.0
11	9.4	6.1	18	6,573.6	1,959.4	186.5	0.0	407.1	6.4	0.0
8x8 grid										
10	3.5	0.5	50	1.2	0.6	0.0	0.0	0.1	0.0	0.0
12	5.5	1.1	50	9.4	2.7	0.9	0.0	1.2	0.1	0.0
14	7.0	1.7	39	84.5	25.6	3.6	0.1	13.2	0.3	0.1
16	9.8	1.5	26	64.3	40.1	13.3	0.0	9.8	0.1	0.0
18	11.0	2.0	21	57.4	17.1	2.9	0.0	2.5	1.5	0.0

Table 2: Number of (non-goal) ICTS nodes where the low level was activated for 4x4 grid (top) and 8x8 grid (bottom)

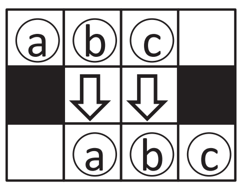

Figure 4: Bottleneck for three agents.

as the first two-agent solution is found we stop and move to the next pair. EPP is slower per pair of agents because the pairwise search is performed until the entire MDD_{ij} was searched and the single-agent MDDs were made as sparse as possible. However, EPP might cause speedup in future pruning and in the low-level search as descried above. REPP is even slower than EPP per ICT node but can cause further pruning of ICT nodes and of single-agent MDDs.

m-agent pruning

All variants of pairwise pruning can easily be generalized to include groups of $m > 2$ agents. Given a group of m agents (where $2 < m < k$), one can actually search through the m-agent MDD search space. Again, if no solution is found for a given set of m agents, the corresponding ICT node can be pruned and declared as a non-goal. The low-level search on the k-agent MDD search space will not be activated and the high-level moves to the next ICT node. In the *simple* m-agent pruning, DFS will be activated. In the *enhanced* and *repeated-enhanced* m-agent pruning breadth-first search will be activated on the m-agent MDD search space and the different single-agent MDDs can be made sparser according to the same reasoning provided above.

Given m agents, m-agent pruning is more effective than all $\binom{m}{m-1}$ $(m-1)$-agent pruning. To illustrate this, consider a bottleneck of 2 cells where 3 agents need to pass it at the same time as shown in Figure 4. Each pair of agents can pass without conflicts (at a cost of 3 moves per agent) but the three agents cannot pass it at the same time with a total of 9 moves.

A single-agent MDD contains up to $|V| \times C^*$ nodes where C^* is the length of the shortest solution. Therefore, an m-agent MDD contains at most $(|V| \times C^*)^m$ nodes. If a solution is found to the m-agent MDD in the pruning phase, the corresponding ICT node is pruned and this is much cheaper than performing the k-agent low-level search which takes $O((|V| \times C^*)^k)$. However in the worst case, there are $\binom{k}{m} = O(k^m)$ groups of m agents so the total time for all m-agent pruning in the worst case is $O((|V| \times c^*)^m \times k^m)$. In general, asymptotically, it should always be beneficial to activate the m-agent pruning when $(|V| \times C^*)^m \times k^m) < (|V| \times C^*)^k$, i.e., when $k^m < (|V| \times C^*)^{k-m}$. In such cases, the total time of the pruning is negligible when compared to that of the low-level search. In other cases, timing results might vary as we detail below.

Experimental results

In our experiments we compared the following 7 different variants of pruning techniques for ICTS. Each of these was activated before the low-level search. If a pruning was *successful* the low level was not activated.

1. No pruning (NP). This is pure ICTS.
2. Simple pairwise pruning (2S).
3. Enhanced pairwise pruning (2E).
4. Repeated enhanced pairwise pruning (2RE).
5. Simple triples pruning (3S).
6. Enhanced triples pruning (3E).
7. Repeated enhanced triples pruning (3RE).

We set a time limit of 5 minutes. If a variant could not solve an instance within the time limit it was halted and *fail* was returned. Our first experiments are on 4-connected 4x4 and 8x8 grids with no obstacles. For each of theses grids and for each given number of agents, we randomized 50 problem instances. The numbers in the tables are averages over the instances that were solved by *all* variants out of the 50 instances. This number of solved instances is shown in the *Ins* columns of the tables discussed below.

Pruning effectiveness

Table 2 (top of this page) presents the effectiveness of the pruning of the different variants for a given number of agents

k	k'	Δ	Ins	NP	2S	2E	2RE	3S	3E	3RE
4x4 grid										
4	1.9	0.5	50	0.1	0.2	**0.1**	0.2	0.3	**0.1**	0.2
5	2.1	0.5	50	**0.2**	0.4	**0.2**	0.3	0.5	**0.2**	0.3
6	3.2	1.1	50	3.1	3.6	**2.2**	2.6	3.3	2.3	2.7
7	4.4	1.8	50	30.5	25.7	13.8	14.9	19.7	**13.5**	15.4
8	6.1	3.3	50	6,354.4	5,968.9	1,672.7	681.1	1,817.2	**646.6**	761.9
9	7.0	4.3	47	14,991.1	12,223.4	3,700.2	3,689.6	10,135.7	**3,533.7**	4,314.6
10	7.8	5.2	39	43,637.9	31,104.3	6,137.7	6,319.2	18,928.7	**6,018.5**	7,041.0
11	9.75	6.5	24	N/A	N/A	24,436.0	19,915.1	60,003.9	**19,527.5**	23,088.6
8x8 grid										
6	1.5	0.1	50	0.2	0.2	**0.1**	0.3	0.2	0.2	0.4
8	2.6	0.5	50	1.8	2.2	**1.7**	2.3	2.4	1.7	2.6
10	3.5	0.5	50	298.9	287.4	13.3	8.6	56.0	**8.0**	10.6
12	5.5	1.1	50	3,076.5	594.9	136.2	**85.2**	274.5	93.3	106.3
14	7.0	1.7	39	11,280.6	3,467.5	844.9	**307.3**	2,562.7	343.2	348.9
16	9.8	1.5	26	17,226.5	12,768.1	1,534.9	**493.3**	4,966.3	859.4	605.5
18	11.0	2.0	21	20,650.7	14,123.8	825.1	**390.3**	9,640.9	505.4	445.1

Table 3: Runtime for 4x4 grid (top) and 8x8 grid (bottom).

(indicated by the k column). The k' column presents the average *effective number of agents*, i.e., the number of agents in the largest independent subgroup found by the ID framework. In practice, due to the activation of ID, ICTS was executed on at most k' agents and not on k.

For each of the pruning variants the table presents the number of non-goal ICT nodes where the pruning failed and the low-level was activated. This number for pure ICTS, given in the NP column, is the total number of such non-goal ICT nodes. For example, consider the line for $k = 8$ (the last number where all 50 instances could be solved by all variants) for the 4x4 grid in the top of the table. There were 579.7 non-goal ICT nodes. For all of these nodes the NP variant (pure ICTS) activated the low-level search. When 2S was activated almost half of them were pruned and the low-level was only activated for 299.1 non-goal ICT nodes. This number decreases with the more sophisticated technique and for 2RE almost *all* nodes were pruned and only 0.2 nodes needed the low-level. Triple pruning show the same tendency and it is not surprising that triples always outperformed the similar pairwise pruning.

It is important to note the correlation between k, k', Δ and the NP column (the number of ICT nodes). When more agents exist, k' and $Delta$ increase linearly but the number of ICT nodes increases exponentially with Δ. This phenomenon was studied in (Sharon *et al.* 2011).

Similar tendencies can be observed for the 8x8 grid (bottom). However, the 8x8 grid is less dense with agents and there are less conflicts. This leads to small values for Δ and therefore for small numbers of ICT nodes.

Runtime

Table 3 shows the runtime in ms. for the same set of experiments. The best variant for each line is given in **bold**. As explained above, there is a time tradeoff per ICT node between the different variants ; the enhanced variants incur more overhead. Therefore, while the enhanced variants always prune more ICT nodes (as shown in Table 2) this is not necessarily reflected in the running time. When the number of agents increases, only relatively easy problems (out of 50) were solved, hence the numbers do not necessarily increase.

Clearly, one can observe the following trend. As a given problem becomes denser with more agents it pays off to use the enhanced variants. For example, for the 4x4 grid, clearly, 7 agent is the point where the winner shifts from 2E to 3E. Similarly, for 8x8 2RE start to win at 12 agents. Note that the best variant outperformed the basic NP variant in up to a factor of 50.

It is interesting to note from both tables that, for the cases of 4x4 and 8x8 grids that we tested, 2RE, 3E and 3RE perform very similar. They all managed to prune almost all non-goal ICT nodes and their time performance is very similar. For 4x4, 3E was slightly faster while for 8x8 2RE was slightly faster. This is explained below.

Dragon age maps

We also experimented with maps of the game *Dragon Age: Origins* from (Sturtevant 2010). Figure 5 shows two such maps (`den520d` (top), and `ost003d` (bottom)) and the *success rate*, i.e., the number of instances (out of 50 random instances) solved by each variant within the 5-minutes limit. Clearly all the pruning techniques significantly solves more instances than NP. Table 4 presents the running times on the instances that could be solved by all of the reported variants. In the DAO maps 2E was the fastest, although other variants were not too far behind.

Discussion

The major difference in the results is that in DAO the simple 2E variant was the fastest while for the grid domains the advanced variants 2RE and 3E were faster than 2E. The explanation is as follows. We note that a major factor that determines the ”hardness” of the problem is the ratio between the number of agents (k) and the size of the graph (n) - i.e., the *density* of the agents. Larger graphs reduce the internal conflicts (more ’blanks’) and the problem becomes easier. This is reflected in the fact that the number of states in the DAO maps is very large but yet, we could solve problems of tens of agents. When the problem is more dense, more conflicts exist and the enhanced pruning techniques tend to

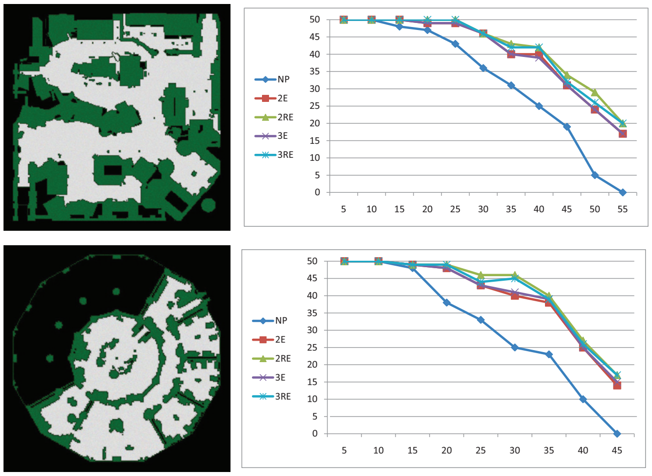

Figure 5: DAO maps (left). Their performance (right). The x-axis = number of agents. The y-axis = success rate.

k	k'	Ins	NP	2E	2RE	3E	3RE
Den520d							
5	1.0	50	58	**52**	61	**52**	67
10	1.3	50	439	**231**	359	275	442
15	1.9	50	16,375	1,251	**679**	1,296	768
20	2.1	49	13,240	**853**	1,282	1,070	1,689
30	4.6	46	67,424	**5,061**	9,042	8,647	14,891
40	5.7	39	N/A	**20,568**	23,455	28,359	35,724
50	6.9	24	N/A	**26,716**	34,283	36,611	51,925
ost003d							
5	1.1	50	592	**62**	82	65	91
10	1.5	50	205	**149**	223	154	238
15	1.8	49	11,577	**396**	523	422	564
20	2.9	48	70,267	**6,053**	8,238	7,600	10,331
30	4.1	39	N/A	**13,920**	14,699	15,853	17,574
40	6.0	23	N/A	**25,784**	33,457	31,493	42,190
45	6.8	14	N/A	**39,278**	51,420	49,023	63,858

Table 4: Runtime on the DAO maps

pay off. The DAO maps are not dense. Therefore, 2E captured most/all the conflicts. There was no point to use more advanced techniques. The 8x8 and 4x4 grids are more dense and the extra time to activate the more advanced techniques was worth it. The results of the 4x4 are interesting. Table 2 shows that 2RE was the most efficient in pruning. By contrast, 3E was the fastest on this gird, although 2RE was only slightly behind. The reason is that since there are only 16 states in this domain, the low-level runs very fast and 3E did not lose too much by activating a few extra low-level searches. Additionally, 3E was faster per node than 2RE due to the fact that the number of agents is small and the number of triples is not large. 2RE performed more checks.

Conclusion and future work

In this paper we discussed a number of techniques that are able to prune non-goal ICT nodes without the need to activate the low-level phase of ICTS. All of these techniques significantly outperform simple ICTS where no pruning is performed and the low-level is activated for all ICT nodes.

There is a tradeoff between the different techniques. The advanced pruning techniques incur larger overhead but are able to prune a larger portion of the ICT nodes. There is no universal winner as the different problem instances differ in many attributes. However when the problem is dense with agents, more conflicts exist and it was beneficial to apply more advanced pruning techniques.

We believe that in this line of work we only touched the surface of tackling optimal MAPF and ICTS. Considerable work remains and will continue in the following directions.

(1) More insights about the influence of the different parameters of the problems on the difficulty of MAPF will better reveal when the ICTS framework is valuable and what pruning technique performs best under what circumstances.

(2) Improved pruning techniques. In this paper we only provided a comparison between a number of pairwise and triple variants. Groups larger than 3 agents can be used. Furthermore, more variants exist. For example, one can devise a pruning technique based on the ID framewrok. That is, try to prune a given pair. Then, add a third agent. Then forth etc. until the entire low-level is activated.

(3) Sophisticated usages of the MDDs might provide further speedup. We are currently applying a CSP-based approach to speed-up the goal test.

Acknowledgements

This research was supported by the Israeli Science Foundation (ISF) grant 305/09 to Ariel Felner.

References

K. Dresner and P. Stone. A multiagent approach to autonomous intersection management. *JAIR*, 31:591–656, March 2008.

M. Jansen and N. Sturtevant. Direction maps for cooperative pathfinding. In *AIIDE*, 2008.

D. Ratner and M. Warrnuth. Finding a shortest solution for the N $\times$ N extension of the 15-puzzle is intractable. In *AAAI-86*, pages 168–172, 1986.

M. Ryan. Exploiting subgraph structure in multi-robot path planning. *JAIR*, 31:497–542, 2008.

M. Ryan. Constraint-based multi-robot path planning. In *ICRA*, pages 922–928, 2010.

G. Sharon, R. Stern, M. Goldenberg, and A. Felner. The increasing cost tree search for optimal multi-agent pathfinding. In *IJCAI*, 2011. To appear.

D. Silver. Cooperative pathfinding. In *AIIDE*, pages 117–122, 2005.

A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *ICCAD*, pages 92–95, 1990.

T. Standley. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*, pages 173–178, 2010.

N. Sturtevant. Pathfinding benchmarks. *Available at http://movingai.com/benchmarks*, 2010.

K. C. Wang and A. Botea. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008.