The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding

Guni Sharon  Roni Stern  Meir Goldenberg  Ariel Felner
Information Systems Engineering Departement
Deutsche Telekom Laboratories
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. We present a novel formalization for this problem which includes a search tree called the increasing cost tree (ICT) and a corresponding search algorithm that finds optimal solutions. We analyze this new formalization and compare it to the previous state-of-the-art A*-based approach. Experimental results on various domains show the benefits and drawbacks of this approach. A speedup of up to 3 orders of magnitude was obtained in a number of cases.

1 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 travel time). 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. We introduce a novel two-level formalization 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 $[C_1, C_2, \ldots, C_k]$ 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 ICTS. We compare ICTS to the traditional A*-based search formalization and show its advantages and drawbacks. Experimental results on a number of domains show that ICTS outperforms the previous state-of-the-art A* approach by up to three orders of magnitude in many cases.

2 Problem definition and terminology

We define our variant of the problem 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 $a_1, a_2, \ldots, a_k$. Every agent $a_i$ is coupled with a start and goal vertices - $start_i$ and $goal_i$.

Initially, (at time point $t_0$) 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, two 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 have a 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.\(^1\)

3 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 \( s_{start_i}(goal_i) \). 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 should be 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, cliques etc.) 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 such that 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).

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(ii) is \( 2 + 2 = 4 \).

Standley compared his algorithm (A*+OD+ID) to a basic implementation of A* and showed spectacular speedups. We therefore compare our approach to A*+OD+ID. 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.

\(^1\)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.

4 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 \( \{opt_1, opt_2, \ldots, opt_k\} \), 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 \( [C_1, \ldots, C_k] \) 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., \( SIC(start) = opt_1 + opt_2 + \ldots + opt_k \). 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 \( \Delta \). \( \Delta \) equals the difference between the cost of the optimal complete solution (\( C^* \)) and the cost of the root (i.e., \( \Delta = C^* - (opt_1 + opt_2 + \ldots + opt_k) \)). 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 exponential in \( \Delta \) 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.

5 Low-level: goal test on an ICT node

A general approach to check whether an ICT node \( s = [C_1, C_2, \ldots, C_k] \) 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.

\(^2\)More accurately, the exact number of nodes at level \( i \) in the ICT is the number of ways to distribute \( i \) balls (actions) to \( k \) ordered buckets (agents). For the entire ICT this is \( \sum_{i=0}^{\Delta} \binom{k+i-1}{k-1} \).
5.1 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 $MDD^i_x$ be the MDD for agent $a_i$ which stores all possible paths of cost $c$. $MDD^i_x$ has a single source node at level 0 and a single sink node at level $C_i$. Every node at depth $t$ of $MDD^i_x$ 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 $MDD^2_1$ and $MDD^3_1$ 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 size of $MDD^i_x$ is at most $|V|^c$. For example, $MDD^3_1$ includes 5 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, $MDD^i_x$ can be reused to build $MDD^i_{x+1}$. We use the term $MDD^i_x(x, t)$ to denote the node in $MDD^i_x$ that corresponds to location $x$ at time $t$. We use the term $MDD^i_x$ 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 $MDD^3_2$ 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 the 2-agent case and then generalize to $k > 2$.

5.2 2-agent MDD and its search space

Consider two agents $a_1$ and $a_2$ 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, $MDD^1_i$ and $MDD^2_i$, which correspond to a given ICT node $[c, d]$.

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 $MDD_i(x_i, t+1)$ and $MDD_j(x_j, t+1)$ such that $x_i$ and $x_j$ are children of $x_i$ and $x_j$, respectively. If $x_i$ and $x_j$ do not conflict then $MDD_{ij}([x_i, x_j], t+1)$ becomes a child of $MDD_{ij}([x_i, x_j], t)$. $MDD_{ij}$ has at most $|V|^2$ 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 $MDD^1_{12}$ and $MDD^2_{12}$ were merged into a 2-agent-MDD, $MDD^3_{12}$.

Low-level search. Only one node is possible at level $c$ (the height of the MDDs) - $MDD^c_{ij}([goal_i, goal_j], c)$. A path to it represents a solution to the 2-agent problem. A goal test for an ICT node therefore performs a search on the search space associated with $MDD^c_{ij}$. 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 $MDD^c_{ij}$ 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 $MDD^c_{ij}$ before arriving at level $c$.

Generalization for $k > 2$ agents is straightforward. A node in a $k$-agent-MDD, $n = MDD_k[k][x[k], t]$, includes $k$ loca-

\footnote{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 $MDD^c_{ij}$ to get an equivalent MDD, $MDD^c_{ij}$. Figure 2(iii) also shows $MDD^2_{ij}$ where a dummy edge (with node $d$) was added to the sink node of $MDD^2_{ij}$.}

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

\Figure 3: (i) MDD$_{12}$ (ii) unfolded MDD$_{12}$.}
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. **foreach** pair of agents \(a_i, a_j\) do
      7. perform pairwise search //optional
      8. if pairwise search failed then
         9. break //Conflict found. Next ICT node
      10. end
   11. end
   12. search the \(k\)-agent MDD //low-level search
      13. if goal node was found then
         14. return Solution
   15. end
   16. end

Theoretical analysis

This section briefly compares the amount of effort done by ICTS to that of A* with the SIC heuristic. We note again that the cost of the root node of the ICT tree is exactly the SIC heuristic of the initial state of A*.

Let \(X\) be the number of nodes expanded by A*, i.e., \(X\) is the number of nodes with \(f \leq C^*\). We would like to measure the extra work of both algorithms with respect to X.

We begin with A*. While A* expands \(X\) nodes, it generates (= adds to the open list) many more. When expanding a non-goal node, A* will also generate all its \(b\) children.\(^5\) Therefore, the total number of nodes generated by A* is \(X \times b\). Recall that \(b = O((b_{base})^k)\). Therefore, A* will visit more nodes than \(X\) by a factor which is exponential in \(k\).\(^6\) The main extra work of A* with respect to \(X\) is that nodes with \(f > C^*\) might be generated as children of nodes with \(f = C^*\) which are expanded. A* will add these generated nodes to the open list but will never expand them.

Now, consider the two-level ICTS algorithm.

**Proposition:** Let \(n = [C_1, \ldots, C_k]\) be an ICT node such that \(C_1 + \ldots + C_k \leq C^*\). Then, the low-level search on the relevant \(k\)-MDD search space will visit at most \(X\) nodes.

**Proof:** The \(k\)-MDD search space of \(n\) contains only nodes where the cost (\(f\)-value) of agent \(a_i\) is at most \(C_i\). Thus, every node in the \(k\)-MDD search space has \(f\)-value \(\leq C^*\). There are at most \(X\) such nodes. Nodes outside the \(k\)-MDD search space are never visited. This means that no node with cost larger than \(C^*\) will ever be considered by ICTS. This is a strong advantage over A*. However, the extra work (over \(X\)) of ICTS comes from the fact that there are many ICT nodes and each of them can visit up to \(X\) nodes. Recall that the number of ICT nodes is bounded by \(k^\Delta\). Therefore, the number of nodes visited by ICTS is \(O(\Delta \times (b_{base})^k)\) while the number of nodes visited by A* is \(O(X \times (b_{base})^k)\).

Consequently, when \(k\) increases it hurts the performance of A* while when \(\Delta\) increases it hurts the performance of ICTS. This is backed up in our experimental results below.

Experimental results

We compared ICTS to the state-of-the-art A* variant of [Standley, 2010], i.e., to A*+OD+ID with SIC (denoted hereafter as A*). The low-level search on the \(k\)-MDD search space was performed by DFS with transpositions table for pruning duplicates. Similar to A*, ICTS was also built on top of the ID framework. That is, the general framework of ID was activated (see section 3). When a group of conflicting agents was formed, either A* or ICTS was activated.

Our first experiment is on a 4-connected 3x3 grid with no obstacles where we varied the number of agents from 2 to 8. Table 1(top) presents the results averaged over 50 random instances. \(b_{base}\) was set to 5 to account for four cardinal moves plus wait. The \(k\) column presents the number of agents. The \(k^\Delta\) column presents the average effective number of agents, i.e., the number of agents in the largest independent subgroup found by ID. In practice, both A* and ICTS were activated on \(k\) agents and not on \(k\). The A* and ICTS columns present the average runtime in milliseconds. The results confirm our theoretical analysis above. For \(k \leq 6\) we see that \(k^\Delta > k^\Delta\),

\[^5\]To be precise, A* has more overhead. It first considers all the potential children and selects only the legal ones. If duplicate detection is performed, duplicate legal nodes are also discarded.

\[^6\]This is true even if OD was applied on top of A*. OD reduced \(b\) (of regular states) but it is still exponential because each of the \(k\) agents can have many moves that do not increase the \(f\)-value.
and ICTS is faster than A*. For \( k \geq 7, b^{k'} < k^\Delta \) and the performance shifts. A* clearly outperforms ICTS for 8 agents.

The major cause for large \( \Delta \) is the existence of many conflicts. Increasing \( k \) can potentially increase the number of conflicts but this depends on the density of agents and of obstacles. When the density is low, adding another agent will add relatively few conflicts and \( \Delta \) will increase slightly. When the density is high adding another agent will have a much stronger effect. This is shown in the \( \Delta \) column. Moving from 7 to 8 agents increases \( \Delta \) much more than from 2 to 3 agents. The size of the graph has direct influence on the density. For a given \( k \), small graphs are denser and will have more conflicts (and thus larger \( \Delta \)) than large graphs.

To demonstrate this, our second experiment is on a larger grid of size 8x8. We set a time limit of 5 minutes. If an algorithm could not solve an instance within the time limit it was halted and \textit{fail} was returned. Figure 4 presents the number of instances (out of 50 random instances) solved by each algorithm within the 5-minutes limit. The curve “ICTS+P” will be described in Section 8. Clearly, as the number of agents increases, ICTS is able to solve more instances than A*. Table 1 (bottom) presents the average runtime in milliseconds for the instances that were solved by both algorithms within the limit. Here too, the tradeoff between \( k^\Delta \) and \( b^k \) can be observed. ICTS outperforms A* and for 12 agents (where \( k^\Delta = 3.8 \) but \( b^k = 57699 \)) ICTS is 25 times faster.

Limitations of ICTS: When \( k \) is very small and \( \Delta \) is very large, ICTS will be extremely inefficient compared to A*. Figure 5 presents such a pathological example. Agents \( a \) and \( b \) only need to swap their positions (linear conflict) and thus the SIC heuristic is 2. However, both agents must travel all the way to the end of the corridor to swap their relative positions. The cost of the optimal path is 74 (37 time steps for each agent). \( b_{base} \leq 3 \) along the corridor (left, right, wait) and thus \( (b_{base})^k \leq 9 \). A* expanded \( X = 892 \) nodes, generated 2,367 nodes (\( b \approx 4 \) due to illegal and duplicate nodes) and solved this problem relatively quickly in 51ms. \( \Delta = 72 \) and as a result, 2,665 ICT nodes were generated and ICTS solve the problem in 36.688ms.

8 Pairwise abstractions

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 \( s = \{C_1, C_2, \ldots, C_k\} \). 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 \( MDD_{ij} \).

8.1 The pairwise pruning enhancement

The pairwise pruning enhancement is optional and can be performed just before the low-level search. This is shown in lines 9-14 of Algorithm 1. Pairwise pruning iterates over all pairs \( MDD_i, MDD_j \) and searches the 2-agent search space of \( MDD_{ij} \). If a pair of MDDs with no pairwise solution is found, the given ICT node is immediately declared as a non-goal and the high-level search moves to the next ICT node.

Otherwise, if pairwise solutions were found for all pairs of MDDs, we activate the low-level search through the search space of the \( k \)-agent MDD.

There are \( O(k^2) \) prunings in the worst case where all pairwise searches resolved in a 2-agent solution and the \( k \)-agent low-level search must be activated. However, pairwise pruning produce valuable benefits even in this worst case. 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 \( MDD_s^3 \) after it was unfolded from \( MDD^3_{12} \). Light items correspond to parts of the original MDD that were pruned. Node \( c \) in the right path of \( MDD_s^3 \) is invalid as it was not unified with any node of \( MDD^3_{12} \). Thus, this node and its incident edges, as well as its only descendent (\( f \)) can be removed and are not included in \( MDD_s^2 \).

In practice, one can delete invalid nodes from each individual MDD while performing the pairwise pruning search through the search space of \( MDD_{ij} \). If the entire \( MDD_{ij} \) was searched (e.g., when a solution to the 2-agent problem was found), the sparser MDDs \( MDD_s^i \) and \( MDD_s^j \) are available. These sparser MDDs can be now used in the following two cases. (1) Further pairwise pruning. After \( MDD_s \) was obtained, it is used for the next pairwise check of agent \( a_i \). Sparser MDDs will perform more pruning as they have a smaller number of possible options for unifying nodes. (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.

8.2 Experiments with ICTS+pairwise pruning

Figure 4 also presents the number of instances that were solved under 5 minutes for ICTS with the pairwise pruning (denoted ICTS+P). Clearly, ICTS+P solves more instances
than basic ICTS and many more than A*. Table 2 presents the average runtime over the instances that were solved by both ICTS and ICTS+P. This is a larger set of instances than that of Table 1 (bottom) which included all instances that were solved by all three algorithms. ICTS+P clearly outperforms basic ICTS by more than an order of magnitude when the number of agents exceeds 8. The runtime of A* is only presented here up to 12 agents. Above that, A* could not solve this set of instances under 5 minutes.

We also experimented with maps of the game Dragon Age: Origins from [Sturtevant, 2010]. Figure 6 shows three such maps (den520d (top), ost003d (middle) and brc202d (bottom)) and the success rate of solving 50 random instances of these maps within the 5 minutes limit. Curves have the same meaning as figure 4. Table 3 presents the running times of A* and ICTS+P on the instances that could be solved by A*. When the number of agents increases, only relatively easy problems (out of 50) were solved, hence the numbers do not necessarily increase. In all these maps, ICTS+P significantly outperforms A*. The differences are due to the topology of the maps and the amount of conflicts. brc202d has small corridors and is denser and shows relatively smaller advantage of ICTS (and ICTS+P) compared to A*.

9 Discussion and future work

We presented the ICTS algorithm for optimally solving MAPF. We also provided a pairwise pruning enhancement which further speeds up ICTS. We compared ICTS to A* theoretically and experimentally and observed that the performance of A* tends to degrade mostly when \( k \) increases while the performance of ICTS tends to degrade when \( \Delta \) increases. There is no universal winner. In very dense environments such as the 3x3 grid with 8 agents or the pathological case ICTS will be inefficient. However, we have demonstrated that in practical cases, when there are relatively many chunks of open areas, ICTS obtained significant speedup of up to 3 orders of magnitude over the state-of-the-art version of A*.

Future work will continue in a number of directions: (1) More insights about the influence of parameters on the difficulty of MAPF will better reveal which algorithm performs best under what circumstances. (2) Improved pruning methods, (e.g., triples, quadruples etc) or CSP-based approaches might speed-up the goal test. (3) A* might also benefit from heuristics that are more informed than SIC.

10 Acknowledgements

This research was supported by the Israeli Science Foundation (ISF) under grant no. 305/09 to Ariel Felner.

References


<table>
<thead>
<tr>
<th>( k )</th>
<th>( k' )</th>
<th>A*</th>
<th>ICTS</th>
<th>ICTS+P</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.0</td>
<td>0.5</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>4</td>
<td>1.1</td>
<td>0.7</td>
<td>0.2</td>
<td>0.4</td>
</tr>
<tr>
<td>6</td>
<td>1.5</td>
<td>2.9</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>8</td>
<td>2.7</td>
<td>108.0</td>
<td>2.7</td>
<td>3.3</td>
</tr>
<tr>
<td>10</td>
<td>3.5</td>
<td>23,560.4</td>
<td>542.1</td>
<td>14.0</td>
</tr>
<tr>
<td>12</td>
<td>5.2</td>
<td>50,371.4</td>
<td>2,594.6</td>
<td>69.8</td>
</tr>
<tr>
<td>14</td>
<td>7.1</td>
<td>&gt;300,000.0</td>
<td>20,203.1</td>
<td>707.7</td>
</tr>
<tr>
<td>16</td>
<td>9.6</td>
<td>&gt;300,000.0</td>
<td>29,634.2</td>
<td>833.7</td>
</tr>
</tbody>
</table>

Table 2: Runtime in ms. 8x8 grid. No obstacles.

<table>
<thead>
<tr>
<th>( k )</th>
<th>( k' )</th>
<th>A*</th>
<th>ICTS</th>
<th>ICTS+P</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>42</td>
<td>65</td>
<td>128</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>913</td>
<td>385</td>
<td>7,961</td>
<td>127</td>
</tr>
<tr>
<td>15</td>
<td>4,361</td>
<td>481</td>
<td>5,821</td>
<td>206</td>
</tr>
<tr>
<td>20</td>
<td>3,581</td>
<td>803</td>
<td>2,1453</td>
<td>344</td>
</tr>
<tr>
<td>25</td>
<td>16,162</td>
<td>2,667</td>
<td>16,756</td>
<td>295</td>
</tr>
<tr>
<td>30</td>
<td>47,277</td>
<td>4,919</td>
<td>26,203</td>
<td>850</td>
</tr>
</tbody>
</table>

Table 3: A* Vs. ICTS+P on the DAO maps

Figure 6: DAO maps (left). Their performance (right). The \( x \)-axis = number of agents. The \( y \)-axis = success rate.