Artificial Intelligence 195 (2013) 470–495

Contents lists available at SciVerse ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

The increasing cost tree search for optimal multi-agent pathfinding

Guni Sharon, Roni Stern *, Meir Goldenberg, Ariel Felner

Information Systems Engineering, Ben Gurion University, Beer-Sheva, 85104, Israel

ARTICLE INFO

Article history:
Received 11 December 2011
Received in revised form 13 November 2012
Accepted 14 November 2012
Available online 20 November 2012

Keywords:
Heuristic search
Multi-agent pathfinding

ABSTRACT

We address the problem of optimal pathfinding for multiple agents. Given a start state and a goal state for each of the agents, the task is to find minimal paths for the different agents while avoiding collisions. Previous work on solving this problem optimally, 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, called the *increasing cost tree search* (ICTS) that finds optimal solutions. ICTS is a two-level search algorithm. The high-level phase of ICTS searches the increasing cost tree for a set of costs (cost per agent). The low-level phase of ICTS searches for a valid path for every agent that is constrained to have the same cost as given by the high-level phase.

We analyze this new formalization, compare it to the A* search formalization and provide the pros and cons of each. Following, we show how the unique formalization of ICTS allows even further pruning of the state space by grouping small sets of agents and identifying unsolvable combinations of costs. Experimental results on various domains show the benefits and limitations of our new approach. A speedup of up to 3 orders of magnitude was obtained in some cases.

1. Introduction

Single-agent pathfinding is the problem of finding a path between two vertices in a graph. It is a fundamental and important field in AI that has been researched extensively. This problem can be found in GPS navigation [30], robot routing [33], planning [2,9], network routing, and many combinatorial problems (e.g., puzzles) as well [14,12].

Solving pathfinding problems optimally is commonly done with search algorithms that are based on the A* algorithm [8]. Such algorithms use a cost function of $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the shortest known path from the start state to state n and $h(n)$ is a heuristic function estimating the cost from n to the nearest goal state. If the heuristic function h is *admissible*, meaning that it never overestimates the shortest path from n to the goal, then A* (and other algorithms that are guided by the same cost function) are guaranteed to find an optimal path from the start state to a goal state, if one exists [3].

The *multi-agent pathfinding* (MAPF) problem is a generalization of the single pathfinding problem for $k > 1$ agents. It consists of a graph and a number of agents. For each agent, a unique start state and a unique goal state are given, and the task is to find paths for all agents from their start states to their goal states, under the constraint that agents cannot collide during their movements. In many cases, the task is to minimize a cumulative cost function such as the sum of the time steps required for every agent to reach its goal.

* Corresponding author.
E-mail addresses: gunisharon@gmail.com (G. Sharon), roni.stern@gmail.com (R. Stern), mgoldenbe@gmail.com (M. Goldenberg), felner@bgu.ac.il (A. Felner).

0004-3702/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.artint.2012.11.006

MAPF has practical applications in video games, traffic control [23,4], robotics [1] and aviation [18]. Optimally solving MAPF in its general form is NP-complete [32].

Approaches for solving MAPF can be divided to two classes: *decoupled approaches* and *coupled approaches*. In a *decoupled approach* a path is found for each agent individually. These approaches differ by the techniques used to resolve colliding paths. Usually, a decoupled approach is used when the number of agents is large. In such cases, the aim is to quickly find a path for the different agents, and it is often intractable to guarantee that a given solution is optimal.

By contrast, the problem addressed in this paper is to find an optimal solution to a MAPF problem. As such, the algorithm presented in this paper is part of the *coupled approaches* where MAPF is formalized as a global, single-agent search problem. Coupled approaches are usually applied when the number of agents is relatively small and the task is to find a solution with minimal cost.[1]

The traditional approach for solving MAPF optimally, is in a coupled manner by using an A*-based search [19,25]. A node in the search tree consists of the location of every agent at time t. The start state and goal state include all the initial locations and the goal locations of the different agents, respectively. For a graph with branching factor b (e.g., $b = 4$ in a 4-connected grid), there are $O(b+1)$ possible moves for any single agent: b moves to the neighboring locations and one *wait* move where an agent stays idle at its current location. In a *coupled* global search that includes k agents the number of operators per state is the cross product of all k single-agent moves, which is $O((b+1)^k)$. Thus, the branching factor for any A*-based search in the MAPF problem is exponential in the number of agents. Naturally, coupled search algorithms that are based on A* can solve this problem optimally but they may run for a very long time (or exhaust the available memory).

In this paper, we introduce a novel formalization and a corresponding algorithm for finding optimal solutions to the MAPF problem. The new formalization is based on the understanding that a *complete solution* for the entire problem is built from a set of *individual paths* for the different agents. The search algorithm that is based on this new formalization consists of two levels.

The high-level phase performs a search on a new search tree called the *increasing cost tree* (ICT). Each node in the ICT consists of a k-vector $\{C_1, C_2, \ldots, C_k\}$, where k is the number of agents in the problem. An ICT node represents *all* possible solutions in which the cost of the individual path of each agent a_i is exactly C_i. The ICT is structured in such a way that there is a unique node in the tree for each possible combination of costs. The high-level phase searches the tree in an order that guarantees that the first solution found (i.e., ICT node whose k-vector corresponds to a valid solution) is optimal.

For each ICT node visited, the high-level phase invokes the low-level phase to check if there is a valid solution that is represented by this ICT node. The low-level phase itself consists of a search in the space of possible solutions where the costs of the different agents are given by the specification of the high-level ICT node. For this we introduce a problem-specific data-structure that is a variant, adjusted to our problem, of the *multi-value decision diagram* (MDD) [24]. The MDD data-structure stores all possible paths for a given cost and a given agent. The low-level phase searches for a valid (non-conflicting) solution amongst all the possible single-agent paths, represented by the MDDs. We denote our 2-level algorithm as *ICT-search* (ICTS).

Unlike an A*-search, both levels of ICTS are not directly exploiting information from admissible heuristic. Nevertheless, we also introduce efficient pruning techniques that can quickly identify ICT nodes that do not represent any valid solution. These pruning techniques are based on examining small groups of agents (such as pairs or triples) and identifying internal conflicts that preclude the given ICT node from representing a valid solution. When such an ICT node is identified, there is no need to activate the low-level search and the high-level search can proceed to the next ICT node.

We study the behavior of our ICTS formalization and discuss its advantages and drawbacks when compared to the A*-based approaches. Based on characteristics of the problem we show cases where ICTS will be very efficient compared to the A*-based approaches. We also discuss the limitations of ICTS and show circumstances where it is inferior to the A*-based approaches.

Substantial experimental results are provided, confirming our theoretical findings. While, in some extreme cases, ICTS is ineffective, there are many cases where ICTS outperforms the state-of-the-art A*-based approach [25] by up to three orders of magnitude. Specifically, we experimented on open grids as well as on a number of benchmark game maps from Sturtevant's pathfinding database [28]. Results show the superiority of ICTS over the A*-based approaches in these domains.

The paper is organized as follows. In Section 2 the MAPF problem is defined along with basic terminology. Next, Sections 3 and 4 discuss previous work regarding the MAPF problem and elaborate on the state-of-the-art A*-based optimal solver [25]. In Section 5 we introduce and formulate the ICTS algorithm. Section 6 theoretically compares ICTS to the A*-based search algorithms and show its advantages and drawbacks. Section 7 provides initial experimental results. Section 8 introduces enhancements to the basic ICTS in the form of various pruning techniques. Sections 9 and 10 provide further experimental results and Section 11 concludes this paper and describes future and ongoing work.

Preliminary versions of this paper appeared in previous publications [21,22]. In this paper we provide a complete report on the ICTS algorithm and its different variants. In addition, this paper describes the ICTS algorithm in greater details, and provides more thorough experimental evaluation of its performance. Also, a broader theoretical analysis of ICTS is provided, which includes theoretical comparison to other MAPF algorithms.

[1] We note that it is also possible to solve a MAPF problem in a *coupled approach* without any guarantee on the solution cost [26].

2. Problem definition and terminology

In this section we define the multi-agent pathfinding problem. There are many variants for this problem. We mostly focus on one variant, which is commonly used [25,26]. However, we also mention other variants and briefly discuss ICTS for other variants in Section 7.6. It is important to note that many of the algorithms previously proposed for the multi-agent pathfinding problem, including our own, are robust across the different variants. Simple adaptations to the algorithms or to the searched graph might be required.

2.1. Problem input

The *input* to the *multi-agent pathfinding problem* (MAPF) is:

- **(1)** A graph $G(V, E)$ where $|V| = N$. The vertices of the graph are possible locations for the agents, and the edges are the possible transitions between locations. We refer to vertices of the graph as states of a single agent.
- **(2)** k agents labeled $a_1, a_2, \dots, a_k$. Every agent a_i has a start state $start_i \in V$ and a goal state $goal_i \in V$.[2]

Time is discretized into time points and at time point t_0 agent a_i is located in location $start_i$. Another input parameter that is given implicitly by the graph is the branching factor of a single agent. This is the number of locations that a single agent can move to in one time-step. We use the term b_{base} to denote the branching factor of a single agent. This paper focuses on 4-connected grids, where $b_{base} = 5$, since every agent can move to the four cardinal directions or wait at its current location.

2.2. Actions

Between successive time points, each agent can perform a *move* action to an empty neighboring location or can *wait* (stay idle) at its current location. In this paper we assume that both moving and staying have unit cost. Extensions to arbitrary costs of *move* and *wait* actions can be performed.

We also make the following two assumptions.

1. Once the agent arrives at its goal state it *waits* and occupies this state as long as other agents are still traveling.
2. Wait actions at the goal cost zero.

Note that other definitions might not include these assumptions, e.g., (1) an agent can be assumed to disappear once it arrives to its goal location, and (2) *wait* actions at the goal can have a non-zero cost.

2.3. Constraints

The main constraint is that each state can be occupied by at most one agent at a given time. In addition, if a and b are neighboring states, 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 these two constraints is violated.

Note that in our definition agents are allowed to *follow* each other. That is, agent a_i could move from x to y if at the same time, agent a_j moves from y to z. Alternative definitions of the problem might not allow agents to follow each other. For example, the sliding-tile puzzle can be formalized as a MAPF where each tile is an agent and agents are not allowed to follow each other. Thus, since there is only one blank location, only one agent is allowed to move at any given time. Similarly, the traditional definition of the sliding-tile puzzle problem assumes that *wait* actions cost 0.

2.4. MAPF task

A solution for the MAPF problem is a sequence of $\{move, wait\}$ actions for each agent such that each agent will be located at its goal position after performing this sequence of actions. The MAPF problem discussed in this paper is to find a solution of a MAPF problem that minimizes a global cumulative cost function.

The specific global cumulative cost function used in this paper is the commonly-used cost function called the *sum of costs* [4,25]. The *sum of costs* is the summation (over all agents) of the number of time steps required to reach the goal location. The cost of the optimal solution is denoted by C^*.

Note that other global cumulative cost functions are possible. For example, another common MAPF cost function is the *makespan*, which is the total time until the last agent reaches its destination (i.e., the maximum of the individual costs). In other possible cost functions, only *move* actions are counted but *wait* actions are free (corresponding to a notion of fuel usage). Giving different weights to different agents is also possible.

[2] Another variant is to have a set of goal locations for a set of agents. Each agent should arrive at one of its associated goal locations. This is an important variant of this problem. Clearly, it deserves a full treatment on its own but this is beyond the scope of this paper.

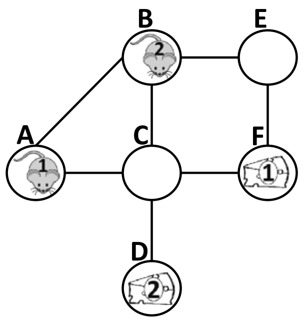

Fig. 1. An example of a MAPF problem with two agents.

Some variants of MAPF do not have a global cost function to optimize, but a set of individual cost functions, one for every agent [15]. In such variants a solution is a vector of costs, one per agent. This type of cost function is part of the broader field of *multi-objective optimization*, and is beyond the scope of this paper.

2.5. An example of a MAPF problem

Fig. 1 shows an example 2-agent MAPF problem that will be used throughout the paper. Agent a_1 has to go from A to F while agent a_2 has to go from B to D. Both agents have individual paths of length 2: $\{A–C–F\}$ and $\{B–C–D\}$, respectively. However, these paths have a conflict, as they both include state C at time point t_1. One of these agents must wait one time step or take a detour. Therefore, the optimal solution C^* is 5 in this example. Note that in this example, the optimal solution with respect to the *total time elapsed* (*makespan*) cost function has a total time elapsed of 3.

2.6. Classes of MAPF computing methods

Computing methods for MAPF can be categorized into two settings: (1) the *distributed* setting, and (2) the *centralized* setting. In a distributed setting, each agent has its own computing power and/or interests and different communication paradigms may be assumed (e.g., message passing, broadcasting etc.). A large body of work has addressed the distributed setting [6,7]. The distributed setting is beyond the scope of this paper. The *centralized* setting assumes a single central computing power which needs to find a solution for all agents. Equivalently, it is also regarded as a *centralized* setting if we have distributed computing power, such as a separate CPU for each of the agents, and a full knowledge sharing where a centralized problem solver controls all the agent. This paper assumes the centralized approach.

3. Decoupled approaches for MAPF

Work assuming a centralized approach can be divided into two classes. The first is called the *decoupled approach* where paths are planned for each agent separately. Such algorithms can run very fast but optimality and even completeness are often not guaranteed in the general case. The decoupled approaches differ by the way they treat conflicts between paths of agents. The second approach for solving the MAPF problem is the *coupled approach* where the MAPF problem is formalized as a global, single-agent search problem, where paths are planned for all of the agents simultaneously.

A large body of work on MAPF was done using the decoupled approach. Usually, such approach is used when the number of agents is large. In such cases, researchers usually focus on providing a valid solution fast. Most of the algorithms from the decoupled approach forsake optimality in order to obtain practical running time.

A prominent example of a decoupled approach algorithm is *Hierarchical cooperative A** (HCA*) [23]. In HCA* the agents are planned one at a time according to some predefined order. Once the first agent found a path to its goal, that path is written (*reserved*) into a global *reservation table*. More specifically, if the path found for agent a_i is $v_i^0 = start_i, v_i^1, v_i^2, \ldots, v_i^l = goal_i$, then the algorithm records that state v_i^j is occupied by agent a_i at time point t_j.[3] When searching a path for a given agent, paths chosen by previous agents are blocked, i.e., the agent may not traverse through locations that are in conflict with previous agents.

A number of variants for HCA* (simple HCA* and Window-HCA*) were introduced [23]. While the HCA* idea is appealing, it has a few drawbacks. First, when too many agents exist, deadlocks may occur and HCA* is not guaranteed to be complete. Second, HCA* does not provide any guarantee on the quality of its solution, and thus the solutions may be far from optimal. HCA* is one of the few decoupled approaches that explicitly considers using admissible (lower bound) heuristics to solve the problem. The heuristic used is the individual cost of the agent in question calculated by solving that agent optimally while

[3] Reservation tables can be implemented as a matrix of $\#vertices \times \#timesteps$, or in a more compact representation such as a hash table for the items that have been reserved.

ignoring all other agents. This idea was further extended by abstracting the size of the state space so that this heuristic could be built more quickly [29].

A similar approach of using reservation tables was used in a system that is devoted to traffic junctions where cars (agents) arrive at the junction and need to cross it [4]. That system solves an online version of the problem where cars arrive at a junction and once they cross it they disappear.

Other decoupled approaches establish flow restriction, similar to traffic laws [34,10]. Each position on the grid is assigned a direction (one outgoing edge). Agents are forced to move in the designated direction and the chance of conflicts is reduced significantly. These approaches prioritize collision avoidance over short paths and work well in state spaces with large open areas. As with many decoupled algorithms, these approaches are not complete for the general case, as deadlocks may occur, and the returned solution may not be optimal.

Some decoupled algorithms are proved to be complete but only in special types of graphs. A new decoupled approach was presented by Wang and Botea [35]. The basic idea is to pre-compute alternative paths for each successive move of all agents. If the original computed path of agent a_i is blocked by another agent a_j, we redirect agent a_i to its alternative path. The main limitations of that approach is that it is not optimal and that it only works, and proved to be complete, on special grids which have the special *slidable* attribute defined in the paper. It is not clear how to generalize this algorithm to graphs that are not *slidable*.

Recently two decoupled algorithms that run in polynomial time were introduced [16,11]. Both algorithms use a set of "macro" operators. For instance, the "swap" macro is a set of operators that swaps location between two adjacent agents [16]. Both algorithms do not return the optimal solution and do not guarantee completeness for the general case. The algorithm provided by Khorshid, Holte and Sturtevant [11] is complete only for tree-like graphs while the algorithm provided by Luna and Bekris [16] is complete only for graphs where at least two vertices are always unoccupied, i.e., $k \leqslant |V| - 2$.

4. Coupled approaches for MAPF

The focus of this paper is on finding optimal solutions for MAPF. This is usually done by *coupled approaches* where MAPF becomes a *single-agent search* problem as follows. The states are the different ways to place k agents into N vertices, one agent per vertex. In the start and goal states agent a_i is located at vertices $start_i$ and $goal_i$, respectively. Operators between states are all the non-conflicting actions (including *wait*) that all agents have.

4.1. A*-based search

Recall that b_{base} is the branching factor for a single agent. In a 4-connected grid $b_{base} = 5$ since every agent can move to the four cardinal directions or wait.

Potentially, the branching factor for k agents is b_{base}^k (denoted as $b_{potential}$). When expanding a state in a k-agent state space, all the $b_{potential}$ combinations should be considered, but only those that have no conflicts (with other agents or with obstacles) are *legal* neighbors. The number of legal neighbors is denoted by b_{legal}, or simply by b in short. When the number of agents is very small $b_{legal} = O(b_{base}^k)$, and thus for worst-case analysis one can consider b_{legal} to be in the same order as $b_{potential}$, i.e., exponential in the number of agents (k). On the other hand, in a dense graph (with many agents and with a small number of empty states), b_{legal} can be much smaller than $b_{potential}$. Note that for extremely dense graphs, it might be possible to devise an efficient implementation that directly considers only the legal moves. For example, in the case of the sliding-tile puzzle where there is only one blank location one usually looks at the location of the blank and the 4 adjacent tiles only. In general, identifying the b_{legal} neighbors from the possible $b_{potential}$ neighbors is a Constraint Satisfaction Problem (CSP), where the variables are the agents, the values are the actions they take, and the constraints are to avoid conflicts.

Given the above definitions of start state, goal states and operators, any A*-based algorithm can be used to solve the MAPF problem optimally. To solve this problem more efficiently with A*, one requires a non-trivial (i.e., non-zero) admissible heuristic. Many advanced heuristic search techniques exist for single-agent search. However, only basic search algorithms and simple heuristics have been used for the MAPF problem.

4.2. Admissible heuristics for MAPF

A simple admissible heuristic is to sum the individual heuristics of the single agents such as Manhattan distance for a 4-connected grid or air line distance for general graphs [19]. A more informed heuristic for MAPF in the context of A* was used by Standley [25]. We denote it as the *sum of individual costs* heuristic (SIC). The SIC heuristic is calculated as follows. For each agent a_i we assume that no other agent exists and *precalculate* its optimal individual path cost from $start_i$ to $goal_i$. Then, the SIC heuristic is the sum of these costs over all agents.[4] For the example problem in Fig. 1, the SIC heuristic is $2 + 2 = 4$. For input graphs of reasonable size, the SIC heuristic can be stored as a lookup table by precalculating the *all-pairs*

[4] Note that the *maximum* of the individual costs is an admissible heuristic for the makespan variant described above, in which the task is to minimize the *total time elapsed*.

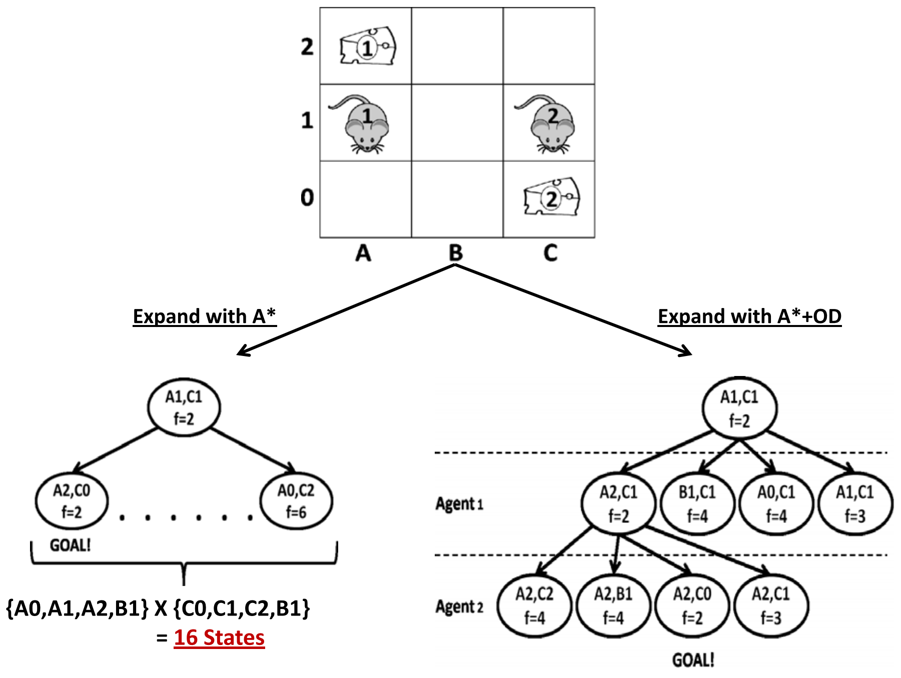

Fig. 2. Example of the benefit of $A^* + OD$.

shortest-path matrix for the input graph. In cases where there is no sufficient memory to store the *all-pairs shortest-path* one can precalculate and store the distances from all vertices to all goal positions in a given problem. This time and memory requirements for this will be $O(N \times k)$. This is negligible compared to the time required to solve the given problem, which is exponential in the number of agents. Otherwise, the SIC can also be computed on the fly.

4.3. Subgraph decomposition

Ryan introduced a coupled, A*-based approach for solving MPAF which uses abstraction to reduce the size of the state space [19]. The underlying map is partitioned into special structures such as cliques, halls and rings. Each structure represents a certain topology (e.g., a hall is a singly-linked chain of vertices with any number of entrances and exits). This partition can be used to search for a solution in an abstracted map that contains these special structures and the connections between them. A general way to use this partition to special subgraphs is for solving the entire MAPF problem as a Constraint Satisfaction Problem (CSP). Each special subgraph adds constraints to the CSP solver, making the CSP solver faster [20].

The efficiency of subgraph decomposition (in terms of runtime) depends on the partitioning of the map. Finding the optimal partitioning is a hard problem and not always feasible. Open spaces are not suitable for partitioning by the defined structures making this algorithm less effective on maps with open spaces.

4.4. Standley's approach

Standley [25] used the A* approach and introduced two important improvements for solving MAPF: *Operator Decomposition* (OD) and *Independence Detection* (ID). The combination of these enhancements, presented by Standley, is to date the strongest A*-based solver for optimally solving MAPF problems. We use this A*-based approach in our experiments and theoretical analysis, and therefore will next explain it in detail.

4.4.1. Operator Decomposition (OD)

The first improvement for A* given by Standley aims at reducing the number of nodes generated by A*. This is done by introducing *intermediate states* between the regular *full states*. *Intermediate states* are generated by applying an operator to a single agent only, one agent at a time. After an operator was applied to all k single agents, a new *full state* is reached. Adding these intermediate states helps in pruning misleading directions at the intermediate stage without considering moves of all of the agents (in a full state).

As an example, consider the MAPF problem shown in Fig. 2. There are two agents, a_1 and a_2. Cells $A1$ and $C1$ are the start states of agents a_1 and a_2 respectively, and cells $A2$ and $C0$ are their goal states, respectively. Clearly, every agent has four legal moves. Therefore, expanding this state in the regular A* formalization will result in generating $4 \times 4 = 16$ children with different f-values, as shown in the lower-left part of Fig. 2. Only fifteen of them are valid as node $\{B_1, B_1\}$ is illegal. Note that if the cost of the optimal solution is C^* (2 in our case), there is no point in generating any state with $f > C^*$. A* on full states does not detect this and generates all possible 15 children. Only one of these children is the goal node (with $f = 2$). It will be expanded next and the algorithm halts.

By contrast, the lower-right part of Fig. 2 shows the tree of intermediate states for the same problem. The root is the full start state (the same root state expanded by A*). When this full state is expanded using A* + OD, only the 4 optional moves of agent a_1 are applied (while agent a_2 remains at state $C1$) and intermediate states are generated for every such operator. Then, the intermediate state $\{A2, C1\}$ is expanded, as it has the lowest f-value of 2. Again, only 4 optional moves (this time for agent a_2) are applied, generating only 4 additional moves. Following, the goal node is expanded ($\{A2, C0\}$) and the search halts. In this example, 15 nodes are generated by regular A* while only 8 nodes are generated by A* + OD. A deeper analysis of the benefits of OD is given in Section 6.3.

4.4.2. *Independence Detection (ID)*

While OD is restricted to an A*-based approach, Standley also introduced another enhancement called the *Independence Detection* (ID) framework. This framework runs as a base level (i.e., is called by the user). An A* solver for specific groups of agents is invoked from the ID level.

Algorithm 1: The ID framework

```
   Input: A MAPF instance
1  Assign each agent to a singleton group
2  Plan a path for each group
3  repeat
4      Simulate execution of all paths until a conflict occurs
5      if Conflict occurred then
6          Try to resolve the conflict [optional]
7          if Conflict was not resolved then
8              Merge two conflicting groups into a single group
9              Plan a path for the merged group
10 until No conflicts occur;
11 return paths of all groups combined
```

ID works as follows. 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, and solve these groups separately. Algorithm 1 lists the pseudocode for ID. First, every agent is placed in its own group (line 1). Each group is solved separately using A* (line 2). The solution returned by the A* algorithm is optimal with respect to the group of agents at hand. The resulting paths of all groups are simultaneously performed until a conflict occurs between two (or more) groups (line 4). Then, ID tries to resolve the conflict between the conflicting groups (line 6). Line 6 is optional and can be skipped. Conflicts are resolved in an ad-hoc manner by trying to replan one group to avoid the plan of the other group, and vice versa.[5] If the conflict between the groups was not resolved, the groups are merged into one group and solved optimally using A* (lines 8–9). This process of replanning and merging groups is repeated until there are no conflicts between the plans of all groups.

Since the complexity of a MAPF problem in general is exponential in the number of agents, the runtime of solving a MAPF problem with ID is dominated by the running time of solving the largest independent problem [25]. ID may identify that a solution to a k-agent MAPF problem can be composed from solutions of several independent subproblems. We denote the size of the largest subproblem k' ($k' \leqslant k$). As the problem is exponential in the number of agents, ID yields in a speedup which is exponential in $k - k'$.

In order to improve the chance of identifying independent groups of agents, Standley proposed a slight modification to the A* algorithm called *conflict avoidance* as follows. The paths that were found for agents are stored in a table. In ID, when a merged group is solved with A* (line 9), then the A* search breaks ties in favor of states that will create the least amount of conflicts with existing planned paths of other agents (agents that are not part of the merged group). The outcome of this improvement is that the solution found by A* using the *conflict avoidance* tie-breaking is less likely to cause a conflict with the other agents. As a result, more independence between groups of agents is detected, resulting in substantial speedup.

We give an illustration of A* + ID for our example problem of Fig. 1 but with two more edges and one more agent as shown in Fig. 3. Here, state E also has an outgoing path E–G–H and a third agent a_3 located at state E needs to go to state H. ID will work as follows. Individual optimal paths of cost 2 are found for each of the three agents. Path $[A$–C–$F]$ for agent 1, $[B$–C–$D]$ for agent 2 and $[E$–G–$H]$ for agent 3. When trying to perform the paths of agents a_1 and a_2, a conflict occurs at state C. There is no way to resolve this conflict and agents a_1 and a_2 are merged into one group. A* is called on this group and returns a solution of cost 5. This solution is now simultaneously performed with the solution of agent a_3. No

[5] To maintain optimality, the cost of the plan found during the replanning must be exactly the same as the original optimal solution for that group.

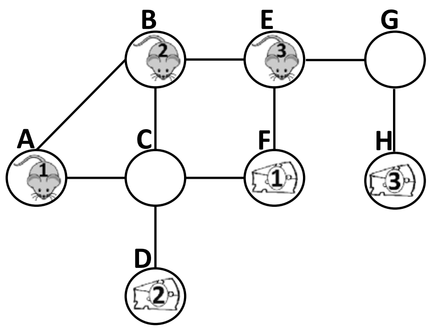

Fig. 3. Extending our MAPF example to three agents.

conflict is found and the algorithm halts. The largest group invoked by A* was of size 2. Without ID, A* would have to solve a problem with three agents. Recall that, on a four-connected grid, a problem with two agents has a potential branching factor of $5^2 = 25$ while a problem with three agent has a potential branching factor of $5^3 = 125$.

It is important to note that the ID framework can be implemented on top of any optimal MAPF solver (i.e., a solver that guarantees to return optimal solutions) used in line 9. Therefore, ID can be viewed as a general framework that utilizes a MAPF solver. Hence, ID is also applicable with the algorithm proposed in this paper (ICTS) instead of with A*. Indeed, in the experimental evaluation of ICTS we ran ICTS on top of the ID framework.

5. ICTS formalization

In this section we describe our new *increasing cost search* formalization for the MAPF problem. This formalization is then used to construct an efficient optimal search algorithm, called the *increasing cost tree search* algorithm (ICTS).

As described in the previous sections, the classic coupled global-search approach spans an A*-based search tree where states correspond to the possible locations of each of the agents. This search is coupled with an admissible heuristic that guides the search. ICTS is based on a conceptually different formalization and unlike A*, ICTS is not guided by a heuristic. Based on the understanding that a *complete solution* for the entire problem is built from *individual paths* (one for each agent), ICTS divides the MAPF problem into two problems:

1. What is the cost of the path of each individual agent in the optimal solution?
2. How to find a set of non-conflicting paths for all the agents given their individual costs?

ICTS answers these two questions in the different levels of the algorithm.

- **High-level search:** searches for a minimal cost solution in a search space that spans combinations of individual agent costs (one for each agent).
- **Low-level search:** searches for a valid solution under the costs constraints (given by the *high-level search*). The low-level search can be viewed as the goal test of the high-level search.

Next, we cover these levels in detail.

5.1. High-level search

The high-level search is performed on a new search tree called the *increasing cost tree* (ICT). The ICT is built as follows.

- **Nodes:** In ICT, every node s consists of a k-vector of individual path costs, $[C_1, C_2, \dots, C_k]$ with 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 total cost of node s is $C_1 + C_2 + \dots + C_k$. Nodes of the same level of ICT have the same total cost.
- **Root of the tree:** The root of ICT is $[opt_1, opt_2, \dots, opt_k]$, where opt_i is the cost of the optimal individual path for agent i assuming no other agents exist. This vector for the root of ICT is equivalent to the SIC heuristic of the start state if used with A*-based searches as described above.[6]
- **Successor function:** Each node $[C_1, C_2, \dots, C_k]$ will generate k successors. Successor $succ_i$ increments the costs of agent a_i resulting in a cost vector of $succ_i = [C_1, C_2, \dots, C_{i-1}, \mathbf{C_i + 1}, C_{i+1}, \dots, C_k]$, thus increasing the total cost function by one.

[6] We note, however, that any A*-based search is not restricted to SIC and might use any possible admissible heuristic. By contrast, ICTS is built exclusively on information of paths of individual agents which is logically equivalent to the SIC heuristic.

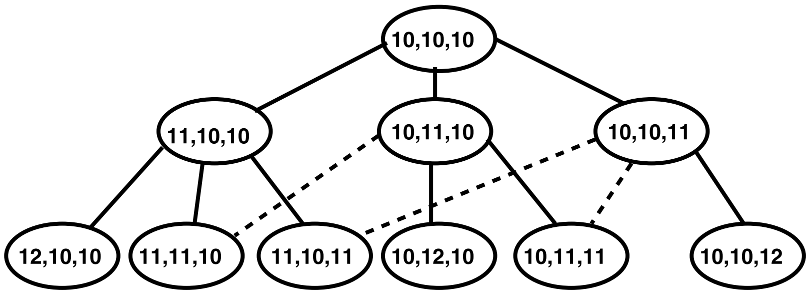

Fig. 4. ICT for three agents.

- **Goal test:** An ICT node $[C_1, \dots, C_k]$ 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.

Fig. 4 shows an example of an ICT with three agents, all with individual optimal path costs of 10. Dashed lines mark duplicate children which can be pruned. Since the costs of nodes in the ICT increase by one at each successive level, it is easy to see that a breadth-first search of ICT will find the optimal solution.

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., $\Delta = C^* - (opt_1 + opt_2 + \cdots + opt_k)$). The branching factor of ICT is exactly k (before pruning duplicates) and therefore the number of nodes in ICT is $O(k^\Delta)$.[7] Thus, the size of ICT is exponential in Δ but not in k. In the extreme case where the agents can reach their goal without conflicts, we will have $\Delta = 0$ and an ICT with a single node, regardless of the number of agents.

The value of Δ is constant and but can only be revealed a posteriori, after the optimal solution cost is found. There are many factors that affect the value of Δ for a given problem instance. For example, the number of agents k, the topology of the searched map, the ratio of the number of agents and the number of vertices of the graph – all affect the value of Δ. Section 7.3 discusses the effect of k on Δ. In general, the experimental results (Section 7) show that while Δ is affected by k, it can be significantly smaller than k in some cases, while in other cases the opposite it true. For example, in large open maps with a small number of agents, we expect Δ to grow slower than k, while in narrow corridors where many conflicts occur, we expect Δ to grow faster than k even for small values of k.

The high-level phase searches the ICT. For each ICT node, the low-level search determines whether it is a goal node, i.e., whether its cost vector corresponds to non-conflicting individual paths of the given costs. This is discussed next.

5.2. Low-level search

A straightforward approach to check whether an ICT node $[C_1, C_2, \dots, C_k]$ 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 combinations of individual paths of the different agents until a complete non-conflicting solution is found.

The main problem with this approach is that for every agent a_i, there may be an exponential number of paths of cost C_i. Moreover, the number of possible ways to combine paths of different agents is the cross product of the number of paths for every agent. Next, we introduce an effective algorithm for doing this.

5.2.1. 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) [24]. MDDs are DAGs which generalize *Binary Decision Diagrams* (BDDs) by allowing more than two choices for every decision node.

An MDD for our purpose is structured as follows. Let MDD_i^c be the MDD for agent a_i which stores all the possible paths of cost c. It has a single *source node*. Nodes of the MDD can be distinguished by their depth below the source node. 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 $goal_i$. MDD_i^c has a single *source node* at level 0, corresponding to agent a_i located at $start_i$ at time t_0, and a single *sink node* at level c, which corresponds to agent a_i located at $goal_i$ at time t_c.

Consider again our example problem instance from Fig. 1. For this problem Fig. 5 illustrates MDD_1^2 and MDD_1^3 for agent a_1, and MDD_2^2 for agent a_2. Agent a_1 only has a single path of length 2 and thus MDD_1^2 stores only one path. Now, consider MDD_1^3. At time 0, the agent is at location A. Next, it has three options for time 1: move to B, *wait* at A (which causes A to also appear at level 1 of this MDD), or, move to C. Thus, it can be in either of these locations at time step 1. These are all prefixes of paths of length 3 to the goal. Note that while the number of paths of cost c might be exponential in c, the

[7] 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 $\sum_{i=0}^{\Delta} \binom{k+i-1}{k-1}$.

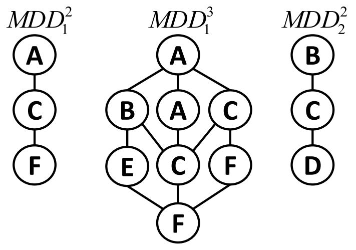

Fig. 5. 2 and 3 steps MDDs for agent a_1 and 2 steps MDD for agent a_2.

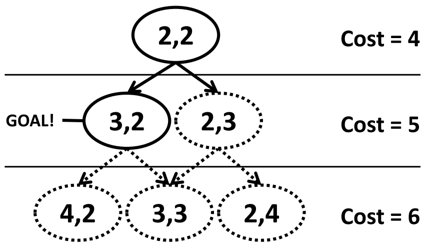

Fig. 6. ICT for the problem in Fig. 1.

size of MDD_i^c is at most $|V| \times c$ as each level in the MDD has no more than $|V|$ nodes and there are exactly c levels. 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}. Thus, one might use previously built MDDs when constructing a new MDD.

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. For example, the source node of MDD_1^2 shown in Fig. 5 is denoted by $MDD_1^2(A,0)$. We use the term MDD_i when the depth of the MDD is clear from the context.

5.2.2. Example of ICTS

The low-level search performs a goal test on nodes of the ICT 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. The ICT for our example problem from Fig. 1 is shown in Fig. 6. The high-level search starts with the root ICT node $[2,2]$. MDD_1^2 and MDD_2^2 (shown in Fig. 5) 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 search. Next, ICT node $[3,2]$ is verified by the high-level search. 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 search and the solution cost 5 is returned.

5.2.3. Goal test in the k-agent-MDD search space

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

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 agents simultaneously to all possible directions. This is the same search space that is searched by the standard A*-based search algorithms for this problem [25]. Now consider the MDDs of agents a_i and a_j that correspond to a given ICT node $[c,d]$: MDD_i^c and MDD_j^d. It is important to note that 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. Fig. 7 shows $MDD_2^{2'}$ where a dummy edge (with node D) was added to the sink node of MDD_2^2.

The cross product of MDD_i and MDD_j spans a subset of the *global 2-agent search space*, denoted as the *2-agent-MDD search space*. This *2-agent-MDD search space* 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. We now define a *2-agent-MDD* denoted as MDD_{ij} for agents a_i and a_j. A *2-agent-MDD* is a generalization of a single-agent MDD to two agents. MDD_{ij} is a *subset* of the *2-agent-MDD search space* spanned by the cross product of the two single-agent MDDs. Every node in MDD_{ij} corresponds to a valid (non-conflicting) pair of locations of the two agents. That is, nodes in the cross product that correspond to a conflict are part of the *2-agent-MDD search space* but are not part of the *2-agent-MDD*.

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 $MDD_j(x_j,t)$. Starting at the two source nodes of MDD_i and MDD_j, MDD_{ij} is built level by level. The source node $MDD_{ij}([x_i,x_j],0)$ is the unification of the two source

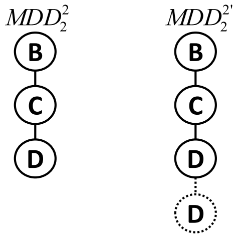

Fig. 7. MDDs of 2 steps for agent a_2 and its extension.

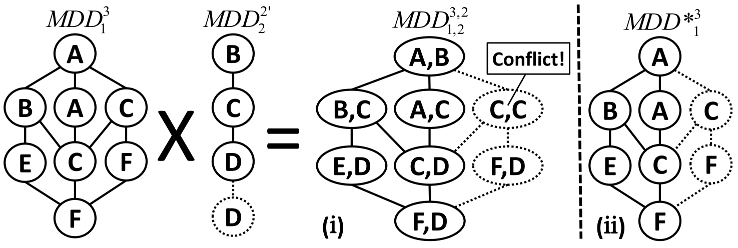

Fig. 8. (i) Merging MDD_1 and MDD_2 to MDD_{12}, (ii) unfolded MDD_1^{*3}.

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 (they are all in the 2-agent MDD search space) and only non-conflicting pairs are added as children of $MDD_{ij}([x_i, x_j], t)$. 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 $\bar{x}_j$ are children of x_i and x_j, respectively. If $\bar{x}_i$ and $\bar{x}_j$ do not conflict[8] then $MDD_{ij}([\bar{x}_i, \bar{x}_j], t+1)$ becomes a child of $MDD_{ij}([x_i, x_j], t)$ 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 $c \times |V|^2$.

In principle, one can actually build and store MDD_{ij} in memory by performing a 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 $t-1$. For example, Fig. 8(i) shows how MDD_1^3 and $MDD_2^{2'}$ (which has a dummy edge so as to have 3 levels) were merged into a 2-agent-MDD, MDD_{12}^3. Elements in **bold** represent the resulting 2-agent-MDD, MDD_{12}^3. Dotted elements represent parts of the *2-agent MDD search space* that are pruned. In the process of building MDD_{12}^3 a conflict (where both agents are assigned position C at time step 1) was encountered. The conflicting node from the 2-agent MDD search space will therefore not be added to MDD_{12}^3 together with all its outgoing edges. Note that node (F, D) at the second level will also not be added to MDD_{12}^3 despite the fact that it does not represent a conflict. This happens because all its legal predecessors (in this case only node (C, C)) are not part of MDD_{12}^3. There is only one possible node at level c (the height of the MDDs), which both agents arrive at their goal. This is the sink node of MDD_{ij}. Any path to it represents a valid solution to the 2-agent problem.

In practice one does not necessarily needs to build the entire structure of MDD_{ij} and store it in memory. In order to perform the goal test for a given ICT node, all that is needed is to systematically search through the nodes of MDD_{ij} and check if such a sink node exists in MDD_{ij} (in this case *true* is returned) or prove that such as node does not exist (in this case *false* is returned). The exact search choice we use is described below in Section 5.3.1.

5.2.4. Generalization to $k > 2$ agents

Generalization for $k > 2$ agents is straightforward. The *k-agent MDD search space* is the cross product of all k single-agent MDDs. Similarly, 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 single-agent MDD nodes of level t that **do not conflict**. The size of $MDD_{[k]}$ is $O(c \times |V|^k)$. The low-level search is performed on the k-agent MDD search space, which is the cross product of all the k single-agent MDDs associated with the given ICT node. In practice (as just explained above for 2 agents) visiting *all* nodes of $MDD_{[k]}$ is not mandatory in order to perform this goal test. Furthermore, as a combined MDD (for multiple agents) is potentially exponential, we would like to avoid building it entirely in memory. Instead, we perform a systematic exhaustive search through the k-agent-MDD search space in order to check whether the current ICT node is a goal node or not. This is done by first building a single-agent MDD for each of the k agents. Following, we run a search in the k-agent MDD search space constrained by the unification rules defined above. This results in visiting the nodes in the k-agent MDD. Once a node at level c is reached, *true* is returned. If the entire k-agent-MDD was scanned and no node at level c had been reached, *false* is returned. This means that there is no way to unify k paths from the k single-agent MDDs. In other words, dead-ends were

[8] 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.

reached in $MDD^c_{[k]}$ before arriving at level c. This search in the k-agent-MDD search space is called the *low-level search*. We note that any systematic exhaustive search will work here and discuss several alternatives in Section 5.3.1 below.

5.3. *The ICTS algorithm*

Algorithm 2: 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      [        //optional
6      foreach pair (triple) of agents do
7          Perform node-pruning
8          if node-pruning failed then
9              Break   //Conflict found. Next ICT node
10     ]
11     Search the $k$-agent MDD search space        // low-level search
12     if goal node was found then
13         return Solution
```

ICTS is summarized in Algorithm 2. The high-level phase searches each node of the ICT (line 2). Then, the low-level phase searches the corresponding k-agent-MDD search space (line 11). The lines in the square brackets (lines 5–10) are optional and will be discussed in Section 8 as a possible enhancement for further pruning the search space before the low-level search is activated. The version without these pruning techniques is referred to as *basic ICTS*.

5.3.1. *Search choices*

To guarantee admissibility, the high-level search should be done with breadth-first search (or any of its variants). However, any systematic search algorithm can be activated by the low-level search on the k-agent MDD search space ((line 11). We found the best variant to be depth-first search. The advantage of DFS is that in the case that a solution exists (and the corresponding ICT node will be declared as the goal) it will find the solution fast, especially if many such solutions exist. In order to prune duplicates nodes we coupled the DFS with a transposition table that stores **all** the visited nodes. This assures that this low-level search for a given ICT node visits every node in the MDD search space at most once.

Note that as stated in Section 4.4.2, ICTS can be activated on top of the ID framework. In such case, ICTS is used to optimally solve subgroups of the k agents (see Algorithm 1 for description of ID). Standley has shown that when solving one subgroup of agents it is worthwhile to prefer solutions that will not create conflicts with agents from a different subgroup [25]. This is done using the *conflict avoidance table* as explained in Section 4.4.2. Therefore, when ICTS is used on top of ID we slightly modified the search strategy of the low-level search to prefer low-level nodes with minimal conflicts with other groups as follows.

Depth-first search can be implemented as a best-first search where the cost function is the number of steps away from the start states (i.e., the g-value). DFS expands the node with the highest g-value. We used such implementation for our low-level search. When ID was used, we changed the cost function of the low-level search to be the k-MDD node with the smallest number of conflicts with other groups as reported by the *conflict avoidance table*.

We therefore only report results for this variant in the experimental section. That is, we used DFS with a transposition table when ID was not used (experiments of type 1 below) and best-first search (which prefers nodes with small number of conflicts with other groups) when ID was used (experiments of type 2 below).

6. Theoretical analysis

This section compares the amount of effort done by ICTS to that of A* and A* + OD with the SIC heuristic. It is well known that A* will always expand all the nodes with $f < C^*$ and some of the nodes with $f = C^*$. In the worst case (depending on tie braking) A* expands all the nodes with $f \leqslant C^*$. Let X be the number of nodes expanded by A* with the SIC heuristic in the worst-case, i.e., X is the number of nodes with $f \leqslant C^*$.

A* is known to be “optimally effective”, which means that A* expands the minimal number of nodes necessary in order to ensure an optimal solution [3]. As such, any algorithm that is guaranteed to return the optimal solution has a computational complexity of at least $O(X)$. Therefore, we next analyze the complexity of ICTS, as well as A* and A* + OD, with respect to X. We show that the actual work done by A* and A* + OD (in terms of total numbers of expanded and generated nodes) is much larger than X and is substantially larger than ICTS in some cases.

6.1. ICTS

The time complexity of ICTS is composed from the complexity of applying the low-level search for every ICT node visited by the high-level search. As explained in Section 5.1, the number of ICT nodes visited by the high-level search until the optimal solution is found is $O(k^{\Delta})$. Therefore, the complexity of ICTS is the complexity of the low-level search on a single ICT node times $O(k^{\Delta})$.

The low-level search is a systematic search of the k-agent MDD search space. To search the k-agent MDD search space, one must first build the single-agent MDD for each of the k agents and then consider their cross-product. Therefore, the complexity of the low-level search is composed from the complexity of building k single-agent MDDs and searching the k-agent MDD search space.

Building an MDD for a single agent requires, in the worst case, time that is linear in the size of the single-agent state-space and the depth of the MDD. This is in general exponentially smaller than the number of states visited by A* (X) when searching the global k-agent state-space. Thus, it can be omitted. Next, we show that complexity of searching the k-agent MDD search space (i.e., the number of nodes in the k-agent MDD search space that are visited during the search) is bounded by X.

Lemma 1. *For every node m in the global k-agent search space that will be visited by a low-level search for ICT node $[C_1, \dots, C_k]$, it holds that $f(m) \leqslant \sum_{i=1}^{k} C_i$, where $f(m)$ is the cost given to node m by an A* search with the SIC heuristic.*

Proof. Node m represents a possible location for every agent, denoted by $loc_i(m)$, at time step t. Let $g_i(m)$ be the cost of agent a_i arriving at $loc_i(m)$. Similarly, let $h_i(m)$ be the heuristic estimate of the distance from $loc_i(m)$ to $goal_i$, and let $f_i(m) = g_i(m) + h_i(m)$.

The cost of node m in an A* search with the SIC heuristic is given by the sum of $f_i(m)$, since:

$$f(m) = g(m) + h(m) = \sum_{i=1}^{k} g_i(m) + \sum_{i=1}^{k} h_i(m) = \sum_{i=1}^{k} f_i(m)$$

By definition, the k-agent MDD search space is spanned by $MDD_1 \times MDD_2 \times \dots \times MDD_k$. MDD_i contains only nodes on a path of cost C_i from $start_i$ to $goal_i$. Consequently, $f_i(m) \leqslant C_i$, since $f_i(m)$ uses an admissible heuristic ($h_i(m)$). Therefore we can conclude that $f(m) \leqslant \sum_{i=1}^{k} C_i$ as required. □

Theorem 1. *For any ICT node visited by the high-level search, the low-level search on the relevant k-agent MDD search space will visit at most X states from the global k-agent search space.*

Proof. Since the high-level search is performed in a breadth-first search manner, the cost of every ICT node returned by the high-level search will never exceed C^*. This means that for any ICT node $[C_1, \dots, C_k]$ that is visited by the high-level search, it holds that $C_1 + \dots + C_k \leqslant C^*$.

Following Lemma 1, we have that any node m visited by the low-level search will have $f(m) \leqslant C^*$. Since X is the number of all the nodes with $f(m) \leqslant C^*$, we can conclude that any low-level search will visit at most X states. Note that since we implemented the low-level search with DFS plus a transpositions table, each of these X states is visited at most once. □

An important observation is that nodes in the global k-agent search space that are outside the k-MDD search space are never visited by the low-level search. This means that no node with cost larger than C^* will ever be considered by ICTS. As will be shown next (in Section 6.2), this is not the case with A*.

While a single low-level search of ICTS will not visit more than X states from the global k-agent search space, ICTS preforms many low-level searches, one for each ICT node. Since the number of ICT nodes is bounded by k^{Δ}, and the number of nodes visited for a single ICT node is bounded by X, then the number of nodes visited by ICTS is $O(X \times k^{\Delta})$.[9]

Next, we perform a similar analysis of the number of nodes visited by A*, with respect to X.

6.2. A*

While A* expands only X nodes, it *generates* (= adds to the open list) many more. Every time a non-goal node is expanded, A* generates all its $b_{legal} = O(b_{base}^{k})$ children.[10] Therefore, the total number of nodes generated by A* is $X \times b_{legal}$.[11]

[9] This is an upper bound. Searches in ICT nodes at level l will visit no more than the number of nodes with $f = SIC(root) + l$. Only for depth Δ this equals C^*. Furthermore, one can potentially reuse information from MDDs across ICT nodes.

[10] To be precise, A* has more overhead. It first considers all the $b_{potential} = b_{base}^{k}$ potential children and selects only the legal ones. If duplicate detection is performed, duplicate legal nodes are also discarded.

[11] An equivalent situation will occur in the last iteration of IDA* where the threshold is C^*. IDA* will generate all these b_{legal} nodes and backtrack.

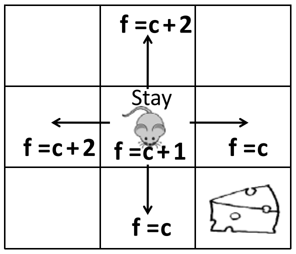

Fig. 9. Example of changes in the f-value on a 4-connected grid. The f-value of the cells is for time step 1.

Recall that $b_{legal} = O(b^k_{base})$. Therefore, A* will generate more nodes than X by a factor which is exponential in k. On a four-connected grid this would be a factor of 5^k.

The main extra work of A* with respect to X is that nodes with $f > C^*$ might be generated as children of nodes with $f \leqslant C^*$ which are expanded. A* will add these generated nodes to the open list but will never expand them.[12] Therefore, the number of nodes visited by A* is actually $O(X \times b^k_{base})$.

6.3. $A^* + OD$

As described in Section 4.4.1, Operator Decomposition (OD) is a recent improvement of A* where *intermediate states* are generated between the regular *full states* by applying an operator to a single agent only. Thus, there is a path of k intermediate states between any pair of neighboring full states. OD reduces the branching factor from $O(b^k_{base})$ to b_{base} (single-agent branching factor). However, since each operator only advances a single agent rather than every agent, the depth of a goal node increases by a factor of k.

While A* + OD expands more nodes then A*, it can generate substantially fewer nodes than A*. As explained above, when A* expands all the X nodes, it generates $O(b^k_{base} \times X)$ nodes. In OD, when any one of these X nodes is expanded, initially (for the first agent) only b_{base} nodes are generated. If the f-value of these b_{base} nodes is above C^*, they will not be further expanded. Thus, potentially, OD may reduce the number of nodes generated by A* from $O(b^k_{base} \times X)$ to $O(b_{base} \times X)$. This is indeed a significant saving.

However, this potential saving of OD is a "best-case" analysis. Following we show that in the worst-case expanding a full state with A* + OD with $f = C^*$ may incur expanding and generating a number of nodes that is exponential in the number of agents.

Consider a single agent a_i located at location loc, and assume that h_{SIC} is the heuristic of that individual agent (i.e., $h_{SIC}(loc)$ is the distance from loc to $goal_i$). Define b_{SIC} as the number of locations adjacent to loc for which $h_{SIC} = h_{SIC}(loc) - 1$. For an open 4-connected grid $b_{SIC} = 2$, as there are at most two neighbors whose general direction is towards the goal. These neighbors will have the same f-value as location loc. For example, consider Fig. 9 where and located in the middle cell with $f = c$ at time t_0. The agent only has two locations at time t_1 to reach the goal in the lower-right corner and keep the cost of $f = c$.

With A* + OD, every state (full or intermediate) will generate only b_{base} children. However, b_{SIC} of them will be expanded afterwards, since they have exactly the same f value as their parent.[13] The number of intermediate nodes that are expanded below a full node with f-value of C^* is therefore b^k_{SIC}. Each of these nodes will generate $b_{base} - b_{SIC}$ children with f-value that is larger than C^*. So the total number of nodes that are generated and not expanded below each full state is $b^k_{SIC} \times (b_{base} - b_{SIC}) = O(b^k_{SIC} \times b_{base})$.

In total, the number of generated nodes by A* + OD is $O(X \times b^k_{SIC} \times b_{base})$. Therefore, A* + OD will visit more nodes than X by a factor which is still exponential in k but the base of the exponent is reduced from b_{base} to b_{SIC}. In an open 4-connected grid, this means a reduction from 5^k to 2^k. Note that this analysis ignores the intermediate states expanded by nodes with f-value smaller than C^*. However, as the MAPF problem is an exponential domain, the number of nodes with f-value equal to C^* dominates the number of nodes with $f < C^*$. This is a well-known phenomenon in combinatorial spaces, which is exploited by search algorithms such as IDA* [36]. Thus, A* + OD expands $O(X \times b^k_{SIC} \times b_{base})$.

Table 1 summarizes the number of nodes visited by A*, A* + OD and ICTS compared to the number of nodes expanded by A* (X). Consider the difference between the number of nodes visited by A*, i.e., $X \times b^k$, and the number of nodes visited by ICTS , i.e., $X \times k^\Delta$. Clearly, ICTS visits less nodes than A* if k^Δ is smaller than b^k. As explained in Section 5.1, the exact value of Δ is affected by the value of k, as adding more agents will likely increase Δ. However, there are factors other than k that affect the value of Δ, such as the topology of the map and the ratio between the number of agents and the number of vertices of the graph.[14] Therefore, in some cases, $k^\Delta < b^k$ and ICTS will outperform A* while in other cases $b^k < k^\Delta$ and A* will outperform ICTS. In the experimental results we show both cases. For example in an open 8×8 grid with 10 agents

[12] These nodes are called *surplus nodes* in our new formalization which studies tradeoffs of expanded nodes vs. generated nodes [5].

[13] Naturally, b_{SIC} is different for every agent and location but we assume that b_{SIC} is a constant for simplicity.

[14] This is discussed below in Section 7.3.

Table 1
Summary of theoretical comparison: A*, A* + OD and ICTS.

Algorithm A* expanded	Nodes visited	
	General	Open 4-connected grid
	X	$f \leqslant C^*$
A*	$X \times b_{base}^k$	$X \times 5^k$
A* + OD	$X \times b_{SIC}^k \times b_{base}$	$X \times 2^k \times 5$
ICTS	$X \times k^{\Delta}$	$X \times k^{\Delta}$

that are located randomly, we have that $b_{base}^k = 9{,}765{,}625$ while $k^{\Delta} = 4105$. On the other hand, in an open 3×3 grid with 8 agents that are located randomly, we have that $b_{base}^k = 390{,}625$ while $k^{\Delta} = 11 \times 10^8$. These results for the 8×8 and 3×3 open grids can be seen in Table 3 and Table 2, respectively.

7. Experimental results: ICTS vs. A*

In this section we provide experimental results comparing ICTS to basic A* and to the state-of-the-art A* variant of Standley [25], i.e., A* + OD. Both versions of A* were guided by the SIC heuristic. For each problem instance encountered during our experiments a time limit of 5 minutes was set. If an algorithm was not able to solve a problem within the time limit it was halted. In such cases, the different measurements were accumulated (number of nodes generated/expanded, runtime, etc.) and treated as lower bounds.

7.1. Experiment types

As explained in Section 4.4.2, the ID framework can be used to enhance both A* and ICTS. When a group of conflicting agents is formed, any optimal and complete MAPF solver can be used (line 9 in Algorithm 1). Therefore, ICTS was also implemented on top of the ID framework. This version is called ICTS + ID and is a competitor to A* + OD + ID. When ID is activated, we distinguish between two different agent counts:

- **Total number of agents.** This number is labeled by k and represents the number of agents that exist in the problem instance to be solved.
- **Effective number of agents.** This number is labeled by k' and represents the number of agents in the largest independent subgroup found by ID. Thus, the solvers (A* or ICTS) were actually activated on at most k' agents instead of k agents.

Our experiments below are also classified into two types with regards to the usage of ID. In the first type of experiments (type 1) we aimed to show the overall performance of the evaluated algorithms. In such cases we activated ID and then executed both ICTS and A* on top of ID. The different algorithms are given a problem instance with a given number of k agents, where the start and goal locations are uniformly randomized. Since ID breaks these problems into subproblems we also report the average value of k'.

In the second type of experiments (type 2), our aim is to study the behavior of the A* or ICTS algorithm for a given number of agents. However, when the ID framework is applied on k agents (whose start and goal locations are randomized) the resulting *effective* number of agents, k', is noisy and its variance is very large. Therefore, in some of our experiments we did not activate ID and compared the algorithms (A* and ICTS) on a fixed number of agents k. In such cases $k' \equiv k$. However, to make the experiments meaningful we generated these groups of agents as follows. In a preprocessing phase, we randomized the start and goal locations for a large number of agents and activated ID on these agents. Whenever ID recognizes k agents that conflict with each other, we place the problem instance that corresponds to these k agent in a bucket labeled k. This process is repeated until the buckets for all values of k are filled with enough instances. We denote this process as the *coupling mechanism* that generates sets of conflicting agents of different sizes.

7.2. 3×3 grid

Our first experiment is on a 4-connected 3×3 grid with no obstacles where we varied the number of agents from 2 to 8. This was an experiment of type 2, i.e., the ID framework was not activated. This is due to the small search space and the fact that the density of the agents is high. Therefore, the activation of ID will not gain too much, as agents tend to be in conflict with each other.[15] Table 2 presents the results averaged over 100 instances generated with the *coupling mechanism* described above. The column "Cost" in Table 2 shows the average solution cost of the optimal solution. Naturally, since the cost function used is additive, adding more agents results in a longer solution cost.

[15] In fact, in some cases applying ID actually degraded the performance, due to the overhead of ID.

Table 2
Results on 3×3 grid averaged over 100 instances. ID was not activated (experiment of type 2). Comparison of the theoretical measures to the number of nodes and to the actual running time (in ms).

k	Cost	Δ	b_{base}^{k}	k^{Δ}	Nodes generated			Runtime		
					A*	A* + OD	ICTS	A*	A* + OD	ICTS
2	3.6	0.10	25	1	23	17	1	1	0	0
3	5.5	0.23	125	1	90	38	2	3	0	0
4	7.8	0.77	625	3	294	102	6	12	1	1
5	10.7	1.93	3125	22	980	425	31	71	8	7
6	13.8	3.27	15,625	350	2401	1383	204	327	44	42
7	18.6	5.83	78,125	84,513	6050	7105	2862	1547	611	1654
8	23.6	10.02	390,625	11×10^{8}	11,055	35,288	79,942	5311	13,474	248,407

Comparing the number of nodes is problematic as the different algorithms have different phases with different types and sizes of nodes. In such cases, researchers sometimes only report running times [25,13]. We chose to report both the number of generated nodes (middle portion of the table) and running times (right portion of the table). However, we note that the number of generated nodes is calculated differently for each algorithm. For A* it is the traditional number of generated nodes. For A* + OD it includes both full states and intermediate states. For ICTS this number corresponds to the summation of the number of k-agent MDD nodes visited by all the calls to the low-level search.

The results confirm our theoretical analysis (Section 6) about the correlation between the performance of the algorithms and the relation between b_{base}^{k} and k^{Δ}. In a 4-connected grid $b_{base} = 5$ in the worst case, to account for four cardinal moves plus *wait*. For $k \leqslant 6$ we see that $b_{base}^{k} > k^{\Delta}$, and ICTS is indeed superior to the A* variants. It generates a smaller number of nodes and needs the same or slightly less time to solve the problem. For $k \geqslant 7$, we have that $b_{base}^{k} < k^{\Delta}$. Correspondingly, the relative performance shifts. For $k = 7$ ICTS still generated a slightly smaller number of nodes than the A* versions, but the A* versions were faster in time due to a smaller constant time per node. Both A* and A* + OD clearly outperform ICTS for 8 agents in both nodes and time.

As a side note, we observed an interesting phenomenon regarding the performance of A* + OD with respect to A*. Standley reported that A* + OD is superior to A* [25], and we supported this claim theoretically in Section 6. However, in an extreme case where the graph is dense with agents A* + OD may be weaker than A*. Such an extreme case is given in Table 2, where for $k \geqslant 7$, A* outperforms A* + OD in terms of generated nodes, and for $k = 8$, A* even outperforms A* + OD in terms of runtime. This occurs because in such dense cases, the branching factor of both A* and A* + OD plays a smaller role than the depth of the search tree, which is larger for A* + OD because it has intermediate states.

7.3. *The growth of k vs. Δ*

Recall that according to the theoretical analysis described in Section 6, the performance of ICTS with respect to A* is affected by the values of Δ. The major cause for large values of Δ is the existence of many conflicts. Increasing k can potentially increase the number of conflicts but this depends on the *density* of the problem at hand, which is defined to be the ratio between k and N. When the density is low, adding another agent will add relatively few conflicts and Δ will increase slightly. When the density is high, adding another agent can increase Δ substantially. This is shown in the Δ column of Table 2. Moving from 7 to 8 agents increases Δ much more than moving from 2 to 3 agents. Naturally, 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 values for Δ) than large graphs.

Fig. 10 shows the relation between Δ, k and the density, for the 3×3 grid. The X-axis corresponds to growing values of density ($d = \frac{\#agents}{\#cells}$). The table presents two curves. The k curve corresponds to the number of agents that procure the given density. For our case of a 3×3 grid, the relation between the density and the number of agents is linear, as there are 9 cells and therefore $k = 9 \times d$. The second curve presents the Δ obtained. As can be seen in Fig. 10, for small density values Δ is smaller than k. However, we can see that in general, Δ increases in a manner which is super-linear in the density. Thus, from a certain value of density, Δ will be higher than k. In the results shown in Fig. 10, we can see that Δ is larger than k for density of 0.82 and larger. Of course, the exact density value from which Δ is larger than k is greatly influenced by the topology of the map. Hence, while this value was 0.82 for our example, this number may vary for other maps.

7.4. 8×8 *grid*

Next, we compared A*, A* + OD and ICTS on a larger grid of size 8×8. Two sets of experiments were performed. The first experiment, reported next, is of type 2. Instances were generated with the coupling mechanism (explained above) to verify that there are k conflicting agents in every instance, and the ID framework was not activated. A type 1 experiment was also performed, where we generated purely random instances of k agents and the ID framework was activated. The results for this experiment are reported in Section 10.

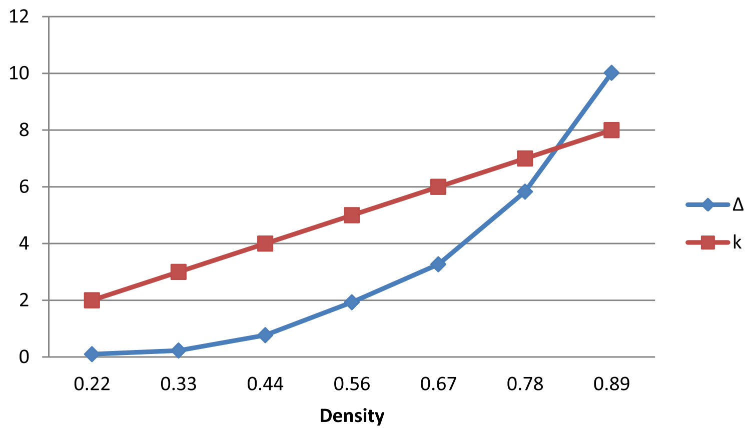

Fig. 10. Δ and k growth on a 3×3 grid with no obstacles.

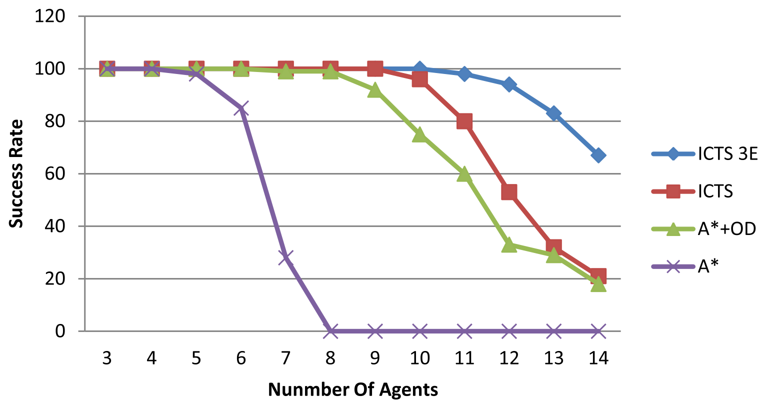

Fig. 11. Success rate on an 8×8 grid with no obstacles. Experiment of type 2 where ID not activated.

Table 3
k conflicting agents (experiment of type 2, where ID is not activated) on an 8×8 grid. Running time in ms.

k	Cost	Δ	b_{base}^{k}	k^{Δ}	Nodes generated			Runtime		
					A^*	$A^* + OD$	ICTS	A^*	$A^* + OD$	ICTS
3	14.7	0.5	125	2	409	90	16	14	1	1
4	20.3	0.9	625	3	2756	303	31	401	5	1
5	26.1	1.4	3125	9	>19,631	933	94	>12,826	43	7
6	29.9	1.9	15,625	30	>78,432	2287	143	>84,689	193	19
7	36.2	2.2	78,125	67	>176,182	4762	372	>239,411	380	81
8	41.0	2.5	390,625	187	NA	12,935	645	NA	2792	282
9	46.7	3.4	1,953,125	1642	NA	46,565	3826	NA	18,516	3048
10	52.3	3.6	9,765,625	4,105	NA	>106,181	24,320	NA	>78,999	24,784

Fig. 11 presents the number of instances (out of 100 random instances) solved by each of the evaluated algorithms within the 5 minutes time limit. Clearly, as the number of agents increases, ICTS is able to solve more instances than $A^* + OD$, and both are stronger than A^*. The line in Fig. 11 labeled by ICTS + 3E denotes results for ICTS with the *enhanced triple pruning* technique that will be described in Section 8. ICTS with this enhancement solves more instances than basic ICTS and many more than $A^* + OD$.

Table 3 presents the average number of states visited and the runtime in milliseconds for the instances that were solved by both $A^* + OD$ and ICTS algorithms within the time limit. Since A^* could not solve all these instances, we use the accumulated measures (nodes and time) of A^* until the timeout as lower bounds. Note that for $k = 10$, $A^* + OD$ could solve less then 80% of the instances (as can be seen in Fig. 11). Thus, for this case, we also report instances not solved by $A^* + OD$ and we report accumulated measures (nodes and time) as lower bounds for $A^* + OD$. It is important to point out that there were no instances solved by $A^* + OD$ that were not solved by ICTS. It is clear that in this setting ICTS significantly outperforms both A^* variants (A^* and $A^* + OD$). For example for 5 agents ICTS is more than 1422 times faster than A^* and 18 times faster than $A^* + OD$. The superior performance of ICTS over the A^* variants corresponds to the theoretical analysis

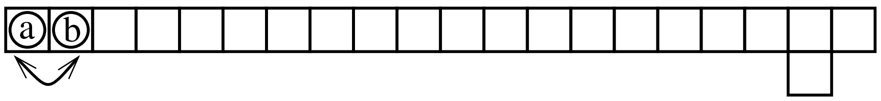

Fig. 12. ICTS pathology.

presented in Section 6, where ICTS is expected to outperform the A* variants when $k^{\Delta} < b_{base}^{k}$. Indeed in this setting (8×8 grid, no obstacles), b_{base}^{k} grows very fast while k^{Δ} grows relatively slowly. These values can be seen in the relevant columns of Table 3. Note that theoretically, if we continue to add agents to an 8×8 grid we may reach a point where $k^{\Delta} > b_{base}^{k}$. However, solving instances with such a large value of k was not feasible with our computing resources.

7.5. Limitations of ICTS

In the vast majority of our experiments and settings the ICTS approach was superior to the A* approach. However, there are cases where A* is significantly faster than ICTS. We now concentrate on such cases.

Recall that A* is exponential in k and ICTS is exponential in Δ. Therefore, when k is very small and Δ is very large, ICTS will be extremely inefficient compared to A*. Fig. 12 presents such a pathological example. Agents a and b are on the left side of the corridor and only need to swap their positions (linear conflict). 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). $b_{base} \leqslant 3$ along the corridor (left, right, wait) and thus $b_{base}^{k} \leqslant 9$. A* expanded $X = 852$ nodes, generated 2367 nodes ($b_{legal} \approx 4$ due to illegal- and duplicate nodes) and solved this problem relatively quickly in 51 ms. By contrast, $\Delta = 72$ and as a result, 2665 ICT nodes were visited and ICTS solve the problem in 36,688 ms.

Similarly, as shown in Table 2, in the 3×3 grid when k was large and thus the density was very high, again, Δ was very large and ICTS was inferior to A*. For those two cases, though for different reasons, Δ can be very large. While for the corridor case, the density was low and there was only a single conflict, solving this conflict caused many extra moves over the SIC heuristic. Thus, this case is pathological from the ICTS perspective due to its topological structure. For the 3×3 grid, the density was high and caused a large number of conflicts. Resolving these conflicts results in a large value for Δ.

7.6. ICTS on other MAPF variants

In this paper we focus on the commonly used variant of MAPF (described in Section 2). Other variants might have different time analysis for ICTS. To demonstrate this, two extreme examples are provided next.

- **Time-elapsed variant** – In this variant the task is to minimize the number of time steps elapsed until all agents reach their final positions (this is also known as the *makespan*). For this case, there is no meaning to the individual cost of a single agent. All agents virtually use the same amount of time steps. Thus, the size of the ICT will be linear in Δ instead of exponential.
- **Variable costs variant** – In this variant the different actions have different costs for the different agents. Let ϵ be the minimum possible step cost. The difference between two successive levels in the ICT must be at least ϵ (in order to maintain optimality). The size of the ICT will now be exponential in $\frac{\Delta}{\epsilon}$.

Thus, the relative behavior of the algorithms for these variants of the problem might be different than the ones reported in this paper.

8. ICT pruning techniques

All ICT nodes visited by the high-level search are non-goal nodes, except for the last node (the goal node). For all these nodes the goal test will return false. This is done by the low-level search by scanning the entire k-agent-MDD search space.

We now turn to discuss a number of useful pruning methods that can be optionally activated before the low-level phase on an ICT node n. This is shown in lines 5–10 of Algorithm 2. If the pruning was *successful*, n can be immediately declared as non-goal and there is no need to activate the low-level search on n. In this case, the high-level search 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.

8.1. 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-MDD search space* that corresponds to MDD_{ij}. If no solution exists to this 2-agent problem (i.e., searching the 2-agent-MDD search space 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 is done in the *Simple Pairwise Pruning* (SPP) variant where we iterate over all $\binom{k}{2}$ pairs of agents to find such cases.

Algorithm 3: 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     if solution not found then
6         return SUCCESS    // Next ICT node
7 return FAILURE    // Activate low-level search on n
```

SPP, presented in Algorithm 3 is optional and can be performed just before the low-level search (lines 5–10 in Algorithm 2). SPP iterates over all pairs (MDD_i, MDD_j) and searches the 2-agent-MDD search space that corresponds to MDD_{ij}. If a pair of MDDs with no pairwise solution is found (line 5), SUCCESS is returned. 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, FAILURE is returned (line 7) and the low-level search must be performed over the k-agent-MDD search space of the given ICT node n.

SPP is performed with a DFS on MDD_{ij} (line 2). The reason is again that a solution 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. Algorithm 3 will then move to the next pair of agents and try to perform pruning for the new pair on node n. In the worst-case, all pairwise searches found a 2-agent solution and FAILURE is returned. This will incur $\binom{k}{2} = O(k^2)$ different searches of a 2-agent-MDD search, one for every pair of agents.

8.2. Enhanced pairwise pruning

It is still possible to gain knowledge from searching all the pairs of 2-agent-MDD, even if all such searches resolved in a 2-agents solution (and thus the ICT node was not pruned). This is done by changing the search strategy of the pairwise pruning from depth-first search 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. A node at level t of MDD_{ij} represents a valid location of agent i and agent j at time step t. Conflicting locations, e.g., where agent i and agent j are at the same location, are of course discarded from MDD_{ij}. We can now unfold MDD_{ij} back into two single-agent MDDs, $MDD{*}_i$ and $MDD{*}_j$. $MDD{*}_i$ and $MDD{*}_j$ can be sparser than the original MDDs, since $MDD{*}_i$ 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 are deleted.

Fig. 8(ii) shows $MDD{*}^3_1$ after it was unfolded from MDD^3_{12}. Dashed nodes and edges correspond to parts of the original MDD that were pruned. For example, node C in the right path of MDD^3_1 is invalid as it was not unified with any node of MDD^3_2. Thus, this node and its incident edges, as well as its only descendant (F) can be removed, and are not included in $MDD{*}^3_1$.

The unfolding process described above can require searching the entire 2-agent-MDD search space of MDD_{ij}. The outcome of this process is $MDD{*}_i$ and $MDD{*}_j$, which are potentially sparser than MDD_i and MDD_j. Having sparser MDDs is useful for the following two tasks:

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 (high-level) node pruning as they have a smaller number of possible options for unifying nodes and lower chances of avoiding conflicts. 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-MDD search space for the low-level search than the original MDDs.

We call this improved pruning process Enhanced Pairwise Pruning (EPP).

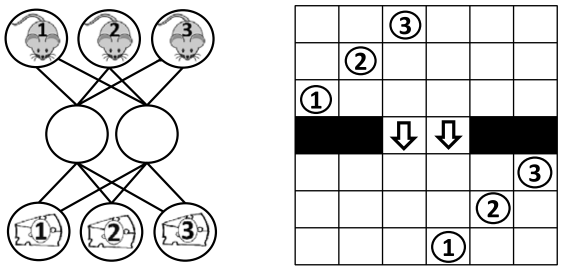

Fig. 13. Example of a bottleneck for three agents.

8.3. *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 search 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 $MDD*_j$ could potentially be 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 the $\binom{k}{2}$ pairs and repeatedly makes the single-agent MDDs sparser. This process is continued until either the ICT node is pruned (because there exist a pair a_i and a_j such that there is no solution to $MDD*_{ij}$) or until no single-agent MDD can be made sparser by further pairwise pruning. Note that REPP can be viewed as a form of arc-consistency, a well-known technique in CSP solvers [17].

8.4. *Tradeoffs*

Natural tradeoffs exist between the different pairwise pruning techniques. Per pair of agents, SPP is the fastest because as soon as the first two-agent solution is found we stop and move to the next pair of agents. EPP is slower than SPP 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 described above.

Let d be the depth of a single *MDD*. Recall that the size of every layer in a given *MDD* is bounded by $|V|$. While searching a 2-agents *MDD* all combinations of 2-agents locations might be examined in every layer. In the worst case $|V|^2 \times d$ or $O(|V|^2)$ states will be visited. This will be done for all $\binom{k}{2}$ pairs $(= O(k^2))$. The total work done by SPP (worst case) and EPP is $O(|V|^2 \times k^2)$. Recall that searching the entire search space is $O(|V|^k)$. Since $|V| > k$, we have that in general the pruning is much faster than the actual search in the k-agent MDD search space. REPP is even slower than EPP per ICT node but can cause further pruning of ICT nodes and of single-agent MDDs.

8.5. *m-Agent pruning*

Not all conflicts can be captured by pairwise pruning. To illustrate this, consider Fig. 13 (left), where there is a bottleneck of two locations through which three agents need to pass at the same time. Each pair of agents can pass without conflicts (at a cost of three moves per agent) but the three agents cannot pass it at the same time with a total of nine moves. Fig. 13 (right) shows a slightly more complex example of a three-way bottleneck in a 4-connected grid.

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 search moves to the next ICT node.

Next, we demonstrate experimentally that the pruning techniques described above can yield substantial speedup for ICTS.

9. Experiments: ICTS pruning techniques

In this section we experiment with the pruning techniques described above and study their behavior. We compared 7 different variants of pruning techniques for ICTS. As explained above, each of these techniques was activated before the low-level search. If a pruning was *successful* the low-level search was not activated. The 7 variants are labeled as follows:

1. No pruning (NP). This is the basic ICTS.
2. Simple pairwise pruning (2S).
3. Enhanced pairwise pruning (2E).
4. Repeated enhanced pairwise pruning (2RE).

Table 4
Number of (non-goal) ICT nodes where the low-level search was activated for 3×3 grid, 4×4 grid, and 8×8 grid and the Den520d map. ID was activated for the den520d map only.

k	k'	Ins	Δ	NP	2S	2E	2RE	3S	3E	3RE
3×3 grid										
4	–	100	0.8	5	3	1	1	0	0	0
5	–	100	1.9	30	15	4	4	3	1	1
6	–	100	3.3	203	112	26	25	24	5	5
7	–	100	5.8	2861	1730	641	627	420	90	84
8	–	80	9.0	36,588	23,317	7609	7454	5444	775	686
4×4 grid										
5	–	100	0.8	7	5	2	2	1	0	0
6	–	100	1.5	28	12	4	4	4	0	0
7	–	100	2.0	87	65	18	18	15	1	1
8	–	99	3.3	528	300	53	51	46	4	4
9	–	98	4.6	3441	1528	349	347	189	12	12
10	–	77	5.6	8658	3618	584	582	382	9	7
8×8 grid										
5	–	100	1.5	12	9	1	1	4	0	0
6	–	99	1.9	43	21	2	2	7	0	0
7	–	98	2.2	67	25	7	7	5	1	1
8	–	96	2.4	135	53	9	9	17	1	1
9	–	88	2.5	258	79	18	17	43	1	1
10	–	65	2.8	402	93	24	23	56	2	2
den520d										
15	1.17	100	0.27	0.48	0.00	0.00	0.00	0.00	0.00	0.00
30	1.52	85	0.80	1.60	0.00	0.00	0.00	0.00	0.00	0.00
45	1.82	77	1.12	2.83	0.06	0.00	0.00	0.00	0.00	0.00
60	1.99	68	1.19	3.81	0.09	0.00	0.00	0.00	0.00	0.00
75	2.13	53	1.40	6.96	0.08	0.00	0.00	0.00	0.00	0.00
90	2.35	32	1.35	10.19	0.29	0.00	0.00	0.00	0.00	0.00

5. Simple triple pruning (3S).
6. Enhanced triple pruning (3E).
7. Repeated enhanced triple pruning (3RE).

Again, 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. We experimented on a 3×3, 4×4 and 8×8 4-connected grids with no obstacles. We also experimented on a large 257×257 map (den520d) from the game *Dragon Age: Origins* (DAO) taken from Sturtevant's repository [28]. This map is shown below in Fig. 16 (top). For the grids, we used the coupling mechanism (described in Section 7.1) to generate hard instances and the ID framework was not activated (experiment of type 2). For the game map, ID was activated (experiment of type 1).

For each of theses grids and for every given number of agents, we randomized 100 problem instances. The numbers in the following tables are averages over the instances that were solved by *all* variants out of the 100 instances. The number of such instances is given in the *Ins* columns of the tables discussed below.

9.1. Pruning effectiveness

Table 4 compares the *effectiveness* of the different pruning variants for a given number of agents (indicated by the k column). The *effectiveness* of a pruning technique was measured by the number of non-goal ICT nodes that were not pruned, i.e., the nodes for which the low-level search was activated. Obviously, lower numbers of non-goal ICT nodes indicate better efficiency of pruning, where zero means prefect pruning. The total number of all non-goal ICT nodes is given in the NP column. For example, consider the line that corresponds to $k = 7$ (the last number where all 100 instances could be solved by all variants) for the 3×3 grid in the top of the table. There were 2861 non-goal ICT nodes. For all of these nodes basic ICTS (with no pruning) activated the low-level search. When 2S was activated almost half of them were pruned and the low-level search was only activated for 1730 non-goal ICT nodes. This number decreases with the more sophisticated technique and for 2RE most of the nodes were pruned and only 641 nodes activated the low-level search. Triple pruning show the same tendency and it is not surprising that triple pruning was always able to prune more ICT nodes than the similar pairwise pruning.

The middle part of the table corresponds to 4×4 and 8×8 grids. Similar tendencies can be observed. Note that, solving problems with the same number of agents but on larger grids result in less conflicts, since the grids are less dense. This leads to small values of Δ and therefore a small numbers of ICT nodes. This is counter intuitive because one might expect

Table 5
Runtime (in ms) for 3×3 grid, 4×4 grid, 8×8 grid and den520d map.

k	k'	Ins	Δ	NP	2S	2E	2RE	3S	3E	3RE
3×3 grid										
4	–	100	0.8	1	3	1	1	4	**0**	1
5	–	100	1.9	9	30	**5**	7	37	6	7
6	–	100	3.3	92	316	61	77	447	**56**	68
7	–	100	5.8	3800	9408	2174	2936	14,343	**1821**	2450
8	–	80	9.0	119,374	220,871	54,045	68,401	306,261	**43,332**	55,991
4×4 grid										
5	–	100	0.8	5	14	**2**	3	13	**2**	3
6	–	100	1.5	25	48	10	14	69	**9**	11
7	–	100	2.0	217	406	54	71	463	**41**	58
8	–	99	3.3	2387	3604	**456**	586	3657	364	515
9	–	98	4.6	23,254	28,097	4731	5872	26,933	**3148**	4359
10	–	77	5.6	76,052	82,248	11,130	14,017	74,115	**9348**	12,817
8×8 grid										
5	–	100	1.5	781	797	50	55	643	**13**	19
6	–	99	1.9	2454	2326	54	66	1531	**44**	62
7	–	98	2.2	5183	3745	507	536	1615	**92**	124
8	–	96	2.4	9487	5320	517	566	2918	**189**	257
9	–	88	2.5	47,778	31,733	2042	2183	18,428	**451**	628
10	–	65	2.8	61,666	38,835	4830	5160	28,677	**1218**	11,755
den520d										
15	1.17	100	0.27	3458	3865	**701**	725	3857	705	763
30	1.52	85	0.80	19,038	18,482	**1671**	1787	18,723	1688	1858
45	1.82	77	1.12	23,672	24,478	**6317**	6513	25,003	6413	6627
60	1.99	68	1.19	38,810	41,140	**20,749**	21,079	41,901	20,826	21,155
75	2.13	53	1.40	69,665	67,107	**25,316**	25,771	67,897	25,508	25,910
90	2.35	32	1.35	55,734	54,474	**26,696**	27,296	54,598	26,871	27,435

that problems on larger graphs will be more difficult. While this is generally true for the case of a single agent, for $k > 1$ agents the amount of conflicts between agents plays a significant role.

The bottom part of the table shows results for the DAO map (den520d). As mentioned above, for this domain we have applied ID, and therefore present in the table both k and k' (the number of agents in the largest independent subgroup). Notice that there is no need to use the advanced pruning methods for this map. The graph is very sparse with agents and therefore k' and Δ are very small. This makes the ICT tree very small and easy to prune even by the simple pruning techniques.

It is important to note the correlation between k (or k' when applicable), Δ and the NP columns (the number of ICT nodes). When more agents exist, Δ increases too but the number of ICT nodes increases exponentially with Δ. This phenomenon was studied in Section 6.

9.2. Runtime

Table 5 shows the runtime results in ms for the same set of experiments.[16] 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 managed to always prune more ICT nodes (as shown in Table 4) this is not necessarily reflected in the running time. However, clearly, one can observe the following trend. As the problem becomes denser with more agents it pays off to use the enhanced pruning variants. Note that the best variant (given in **bold**) outperformed the basic NP variant by up to a factor of 50 in many cases. It is interesting to note from both tables that, for the cases we tested, 2E, 2RE, 3E and 3RE performed similarly. They all managed to prune almost all non-goal ICT nodes and their time performance was very similar.

10. Experiments: ICTS vs. A* on different domains

In Section 7, experimental results were provided for comparing A* with basic ICTS. In Section 9, experimental results were provided to analyze and compare the different pruning techniques. In this section, we conclude the experimental results by comparing the best versions of A*, ICTS and ICTS with pruning, over a range of problems domains. The strongest variant for the A*-based approach is A* + OD, as presented by Standley [25]. While in general all pruning techniques (except for SPP) showed similar trends, we observed empirically that the ICTS + 3E pruning techniques was the best pruning technique. Therefore, in our last set of experiments we only compared A*, A* + OD, basic ICTS and ICTS + 3E. Note that the aim

[16] Numbers are different from Table 3 because different instances were considered. Here problems that were not solved by A* + OD were also considered.

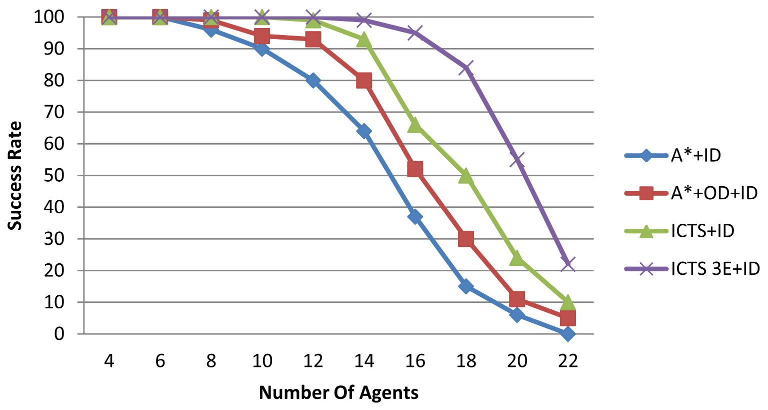

Fig. 14. Success rate on an 8×8 grid. ID activated.

Table 6
Runtime (in ms) on 8×8 grid. Type 1 experiment; ID activated.

k	k'	Ins	Nodes generated			Runtime		
			$A^* + OD$	ICTS	ICTS + 3E	$A^* + OD$	ICTS	ICTS + 3E
4	1.09	100	54	34	32	5	6	4
6	1.22	100	226	408	52	1046	59	10
8	1.71	100	>3793	3078	111	>5102	593	31
10	2.44	100	>5264	1262	165	>19,227	470	27
12	3.41	99	>12,895	10,542	251	>22,856	4310	73
14	4.15	93	>16,982	5358	475	>44,473	2134	265
16	5.02	66	>41,253	15,275	1215	>77,216	9453	1167

in the results presented in this section was to solve problems with as many agents as possible. Therefore, ID was always activated (experiments of type 1).

10.1. 8×8 grids

The first set of experiments presented in this section is on the same 8×8 open grid described in Section 7.

Fig. 14 presents the number of instances that were solved under 5 minutes. Again it is easy to see that the ICTS variants could solve more instances than the A* variants. Table 6 presents the number of nodes and runtime for the same experiment, averaged over the instances that could be solved by three algorithms: $A^* + OD$, ICTS and ICTS + 3E (indicated in the *Ins* column). It is clear that ICTS + 3E is faster than ICTS and outperforms $A^* + OD$ by almost three orders of magnitude.

Note that previous results shown for the 8×8 grid (displayed in Fig. 11 and Tables 3 and 5) were experiments of type 2 (no ID). Therefore, less instances were solved by all algorithms on the type 2 experiments, since when ID is applied, both A* and ICTS are activated on independent subgroups that often have significantly fewer agents than k.[17]

10.2. Grid with scattered obstacles

In this experiment we generated five grids of size 32×32 which differ in the percentage of random cells that were declared as obstacles. This number was varied from 0% to 25% in increases of 5%. We then randomized start and goal locations for 40 agents. The experiment is of type 1 and ID was always activated.

Fig. 15 presents the number of instances (out of 100 random instances) solved by each algorithm within the 5 minutes limit. As can clearly be seen, for every obstacle percentage except 25% we see a similar trend. ICTS + 3E outperforms basic ICTS which in turn outperforms $A^* + OD$. Note that for the fixed number of agents (40) when there are more obstacles the problem becomes harder, since more conflicts between agents occur. As a result, all algorithms managed to solve a very small number of instances for 25% obstacles.

10.3. Dragon age maps

We also experimented with maps from the game *Dragon Age: Origins*, which are part of Sturtevant's repository of benchmarks [28]. Fig. 16 shows three such maps (`den520d` (top), `ost003d` (middle) and `brc202d` (bottom)) and the success

[17] For completeness, Fig. 11 also shows success rate for ICTS + 3E and clearly demonstrates that ICTS + 3E is far superior to $A^* + OD$ and basic ICTS in type 2 experiments.

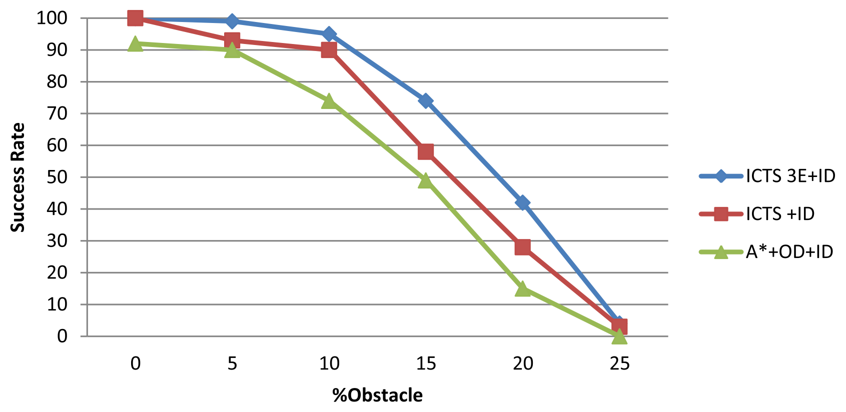

Fig. 15. Success rate for 40 agents on a 32×32 grid with scattered obstacles. Experiment of type 1 – ID activated.

rate of solving 100 random instances on these maps within the 5 minutes time limit. These specific maps were chosen as they have the characteristics of different domains. Map `den520d` (top) has many large open spaces and no bottlenecks, map `ost003d` (middle) has a few open spaces and a few bottlenecks and map `brc202d` (bottom) has almost no open spaces and many bottlenecks.

The different curves shown on the right side of Fig. 16 have the same meaning as the curves in Fig. 15. The experiments were of type 1 – ID always activated. Clearly, in all these maps ICTS + 3E significantly outperformed basic ICTS and $A^* + OD$. In `den520d` (top) and `ost003d` (middle), even the basic ICTS outperformed $A^* + OD$. By contrast, in `brc202d` (bottom), $A^* + OD$ outperformed basic ICTS. This supports by our theoretical analysis as follows. The `brc202d` map is similar to a maze. Mazes often have long and narrow corridors. Therefore, if a number of agents conflict, resolving their conflict might need a large number of extra steps. Each of these extra steps increases Δ and consequently, the performance of ICTS will degrade. Thus, in this map ICTS was outperformed by $A^* + OD$. Only the ICTS + 3E enhancement managed to outperform $A^* + OD$ in this map. Note that an example of such a maze-like case was given in Fig. 12, when discussing the limitation of ICTS.

Table 7 presents the average running times of $A^* + OD$, ICTS and ICTS + 3E, over the instances (out of same 100 instances used above) that could be solved by all three algorithms. When the number of agents increases, only relatively easy problems (out of these 100 instances) were solved, hence the numbers do not necessarily increase. In all these maps, ICTS + 3E significantly outperforms ICTS and $A^* + OD$ by up to two orders of magnitude. Again, due to the maze-like topology of `brc202d`, $A^* + OD$ outperforms basic ICTS and a relatively smaller advantage of ICTS + 3E over $A^* + OD$ is observed for this specific map.

Note that a subset of the experiments shown in this paper have already been presented by the authors in previous publications [21]. A pedant reader might notice faster running-time in this paper for both A^* and $A^* + OD$. This is due to an improved implementation of A^* performed for this journal paper. This improved A^* implementation includes the following enhancement to both versions of A^*. If a state that has exactly the same f-value as its predecessor is generated, it is immediately expanded and does not enter the open list. This enhancement, called *immediate expand*, was previously proposed for A^* [31,27]. *Immediate expand* can result in significant speedups especially in high branching factor problems as the MAPF problem. This explains the faster A^* runtime shown above.

11. Discussion and future work

In this paper we presented the ICTS algorithm for optimally solving MAPF. We compared ICTS to A^* theoretically and experimentally on a range of domains. In particular, we observed that the performance of A^* tends to degrade mostly when k increases while the performance of ICTS with respect to the performance of A^* tends to degrade when Δ increases. Therefore, there is no universal winner. ICTS will be inefficient in very dense environments such as a 3×3 grid with 8 agents, where many conflicts occur, or in cases where resolving conflicts is very costly (e.g., the example shown in Fig. 12). However, we have demonstrated that in many natural domains this is not the case, and ICTS obtains a significant speedup of up to 2 orders of magnitude over $A^* + OD$ (the state-of-the-art A^* variant).

In this paper we also introduced a number of techniques for pruning ICT nodes without the need to activate the low-level search of ICTS. All of these techniques significantly outperform the basic variant of ICTS, where no pruning is performed and the low-level search is activated for every ICT node. There is a tradeoff between the different pruning techniques. More sophisticated pruning methods incur larger overhead but are able to prune a larger portion of the ICT nodes. While no pruning technique dominated all other techniques in all the problem instances, the following guideline was observed. When the problem becomes more dense and more conflicts exist (causing larger Δ) it is beneficial to apply the more advanced pruning techniques.

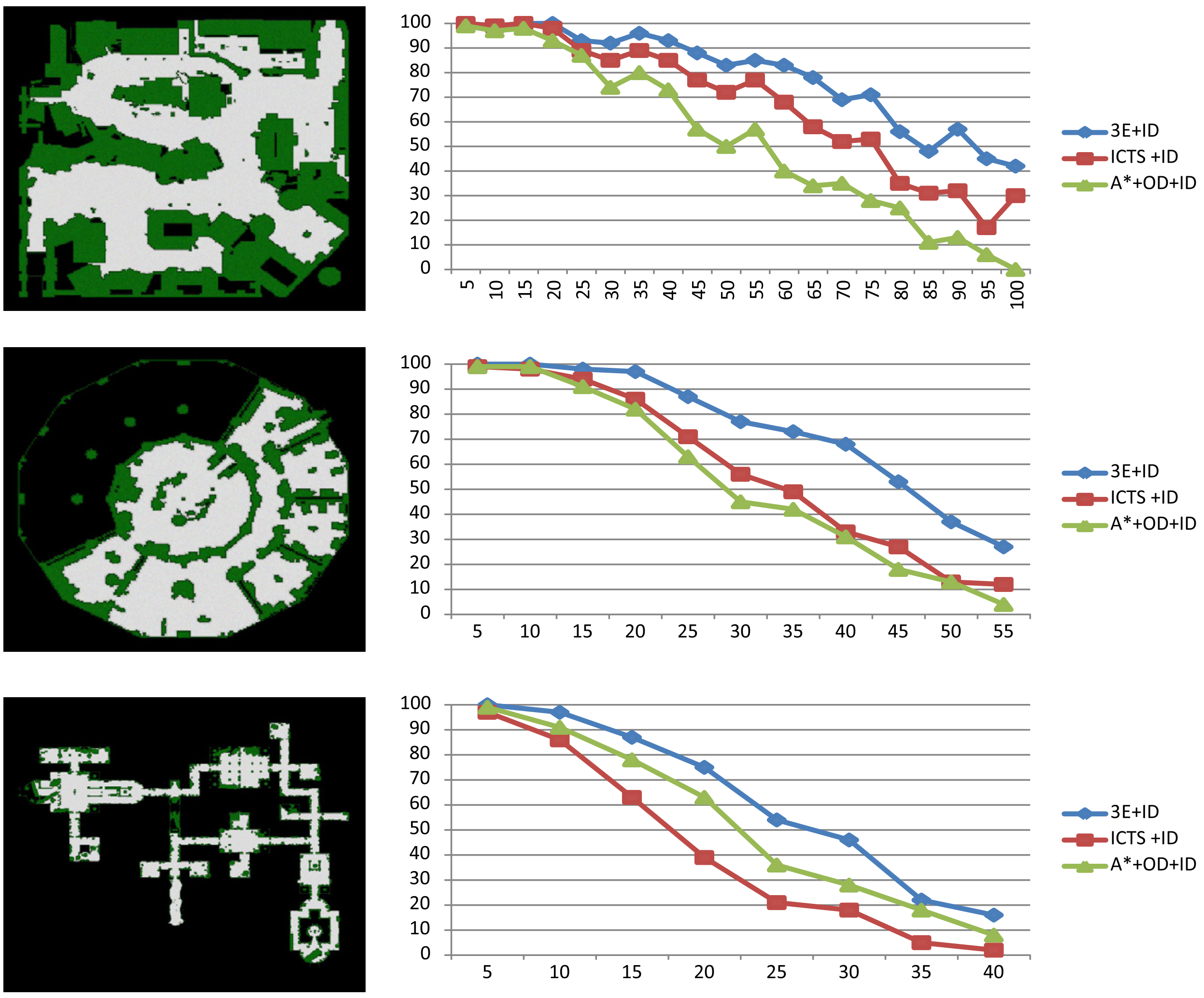

Fig. 16. DAO maps (left). Their performance (right). The x-axis = number of agents. The y-axis = success rate.

Table 7
A* + OD vs. basic ICTS and ICTS + 3E on the DAO maps (runtime in ms).

k	Ins	k'	Δ	A* + OD	ICTS	ICTS + 3E
den520d						
10	97	1.1	0.1	992	561	488
20	93	1.2	0.4	5454	2921	1266
30	71	1.4	0.6	16,316	6560	1390
40	70	1.5	0.7	32,376	9471	2617
50	45	1.8	1.1	49,511	8406	10,219
ost003d						
10	98	1.3	0.4	10,843	1555	359
20	79	1.6	1.1	31,777	4000	945
30	41	1.9	1.4	90,765	10,971	2449
brc202d						
5	96	1.2	0.3	6312	2238	552
10	86	1.4	0.9	23,218	6286	1576
15	63	1.7	1.4	36,590	18,354	2896
20	37	1.9	1.6	51,927	46,371	5010

We believe that in this line of work we have only touched the surface of tackling optimal MAPF. Future work will continue in a number of directions:

(1) The low-level search can be viewed as a Constraint Satisfaction Problem (CSP). The goal is to decide whether there is a solution where every agent is constrained to find a path of a specific cost. One way to encode the low-level search as a CSP would be to assign a variable for every agent-time pair (a, t). The values of this variable would be the possible locations of agent a at time t according to the MDD of a. Constraints will be added to avoid collisions and to ensure that a valid path for every agent is returned. This will allow using state-of-the-art CSP solvers, instead of the systematic search proposed in this paper. Note that the pruning techniques proposed in this paper can be viewed as forms of arc and path-consistency. Hence, a possible future work would be to use more advanced pruning techniques that are based on arc-consistency and path-consistency algorithms, such as AC3 [17].

(2) Deeper insights about the influence of the different parameters of the problems on the properties of a MAPF instance. Such deeper insight could, for example, better reveal when the ICTS framework is valuable and what pruning technique will perform best under which circumstances. Such understanding might give rise to new hybrid algorithms as well.

(3) Extending ICTS for weighted graphs and for cases where the agents have an abstract goal instead of a specific goal for every agent. For example, a group of agents can be given a goal to leave a specific area.

Acknowledgement

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

References

[1] Maren Bennewitz, Wolfram Burgard, Sebastian Thrun, Finding and optimizing solvable priority schemes for decoupled path planning techniques for teams of mobile robots, Robotics and Autonomous Systems 41 (2–3) (2002) 89–99.
[2] Blai Bonet, Hector Geffner, Planning as heuristic search, Artificial Intelligence 129 (1–2) (2001) 5–33.
[3] Rina Dechter, Judea Pearl, Generalized best-first search strategies and the optimality of A*, Journal of the ACM 32 (3) (1985) 505–536.
[4] Kurt Dresner, Peter Stone, A multiagent approach to autonomous intersection management, JAIR 31 (March 2008) 591–656.
[5] Ariel Felner, Meir Goldenberg, Guni Sharon, Roni Stern, Tal Beja, Nathan R. Sturtevant, Jonathan Schaeffer, Robert Holte, Partial-expansion A* with selective node generation, in: AAAI, 2012, pp. 471–477.
[6] Arnon Gilboa, Amnon Meisels, Ariel Felner, Distributed navigation in an unknown physical environment, in: AAMAS, 2006, pp. 553–560.
[7] Devin K. Grady, Kostas E. Bekris, Lydia E. Kavraki, Asynchronous distributed motion planning with safety guarantees under second-order dynamics, in: WAFR, 2010, pp. 53–70.
[8] P.E. Hart, N.J. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Transactions on Systems Science and Cybernetics SCC-4 (2) (1968) 100–107.
[9] Malte Helmert, Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition, Lecture Notes in Computer Science, vol. 4929, Springer, 2008.
[10] M. Renee Jansen, Nathan R. Sturtevant, A new approach to cooperative pathfinding, in: AAMAS, vol. 3, 2008, pp. 1401–1404.
[11] Mokhtar M. Khorshid, Robert C. Holte, Nathan R. Sturtevant, A polynomial-time algorithm for non-optimal multi-agent pathfinding, in: SOCS, 2011, pp. 76–83.
[12] Richard E. Korf, Finding optimal solutions to Rubik's cube using pattern databases, in: AAAI/IAAI, 1997, pp. 700–705.
[13] Richard E. Korf, Multi-way number partitioning, in: IJCAI, 2009, pp. 538–543.
[14] Richard E. Korf, Larry Taylor, Finding optimal solutions to the twenty-four puzzle, in: National Conference on Artificial Intelligence (AAAI-96), 1996, pp. 1202–1207.
[15] Steven M. Lavalle, Seth A. Hutchinson, Optimal motion planning for multiple robots having independent goals, IEEE Transactions on Robotics and Automation 14 (1998) 912–925.
[16] Ryan Luna, Kostas E. Bekris, Push and swap: Fast cooperative path-finding with completeness guarantees, in: IJCAI, 2011, pp. 294–300.
[17] Alan K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1) (1977) 99–118.
[18] Lucia Pallottino, Vincenzo Giovanni Scordio, Antonio Bicchi, Emilio Frazzoli, Decentralized cooperative policy for conflict resolution in multivehicle systems, IEEE Transactions on Robotics 23 (6) (2007) 1170–1183.
[19] Malcolm Ryan, Exploiting subgraph structure in multi-robot path planning, JAIR 31 (2008) 497–542.
[20] Malcolm Ryan, Constraint-based multi-robot path planning, in: ICRA, 2010, pp. 922–928.
[21] Guni Sharon, Roni Stern, Meir Goldenberg, Ariel Felner, The increasing cost tree search for optimal multi-agent pathfinding, in: IJCAI, 2011, pp. 662–667.
[22] Guni Sharon, Roni Stern, Meir Goldenberg, Ariel Felner, Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding, in: SOCS, 2011, pp. 150–157.
[23] David Silver, Cooperative pathfinding, in: AIIDE, 2005, pp. 117–122.
[24] A. Srinivasan, T. Kam, S. Malik, R.K. Brayton, Algorithms for discrete function manipulation, in: ICCAD, 1990, pp. 92–95.
[25] Trevor S. Standley, Finding optimal solutions to cooperative pathfinding problems, in: AAAI, 2010, pp. 173–178.
[26] Trevor S. Standley, Richard E. Korf, Complete algorithms for cooperative pathfinding problems, in: IJCAI, 2011, pp. 668–673.
[27] Roni Stern, Tamar Kulberis, Ariel Felner, Robert Holte, Using lookaheads with optimal best-first search, in: AAAI, 2010, pp. 185–190.
[28] Nathan R. Sturtevant, Benchmarks for grid-based pathfinding, IEEE Transactions on Computational Intelligence and AI in Games 4 (2) (2012) 144–148.
[29] Nathan R. Sturtevant, Michael Buro, Improving collaborative pathfinding using map abstraction, in: AIIDE, 2006, pp. 80–85.
[30] Nathan R. Sturtevant, Robert Geisberger, A comparison of high-level approaches for speeding up pathfinding, in: AIIDE, 2010, pp. 76–82.
[31] Xiaoxun Sun, William Yeoh, Po-An Chen, Sven Koenig, Simple optimization techniques for A*-based search, in: AAMAS, 2009, pp. 931–936.
[32] Pavel Surynek, An optimization variant of multi-robot path planning is intractable, in: AAAI, 2010, pp. 1271–1273.
[33] Jur van den Berg, Rajat Shah, Arthur Huang, Kenneth Y. Goldberg, Anytime nonparametric A*, in: AAAI, 2011, pp. 105–111.
[34] Ko-Hsin Cindy Wang, Adi Botea, Fast and memory-efficient multi-agent pathfinding, in: ICAPS, 2008, pp. 380–387.
[35] Ko-Hsin Cindy Wang, Adi Botea, Tractable multi-agent path planning on grid maps, in: IJCAI, 2009, pp. 1870–1875.
[36] W. Zhang, R.E. Korf, Performance of linear-space search algorithms, Artificial Intelligence 79 (1995) 241–292.