Iterative-Deepening Conflict-Based Search

Eli Boyarski\textsuperscript{1}, Ariel Felner\textsuperscript{1}, Daniel Harabor\textsuperscript{3}, Peter J. Stuckey\textsuperscript{3}
Liron Cohen\textsuperscript{2}, Jiaoyang Li\textsuperscript{2} and Sven Koenig\textsuperscript{2}
\textsuperscript{1}Ben-Gurion University of the Negev
\textsuperscript{2}University of Southern California
\textsuperscript{3}Monash University

boyarske@post.bgu.ac.il, felner@bgu.ac.il, \{daniel.harabor, peter.stuckey\}@monash.edu,
\{lironcoh, jiaoyanl, skoenig\}@usc.edu

Abstract

Conflict-Based Search (CBS) is a leading algorithm for optimal Multi-Agent Path Finding (MAPF) which features strong performance. CBS variants typically compute MAPF solutions using some variation on A* search, however they often do so under strict time limits so as to avoid exhausting available memory. In this paper we present IDCBS, an iterative-deepening variant which can be executed without exhausting memory and without tight time limits. IDCBS can be substantially faster than CBS due to incremental methods that can be applied when processing CBS nodes. It can optimally solve a large number of problems presently thought to be out of reach for optimal and bounded-suboptimal CBS.

1 Introduction

Multi-agent Path Finding (MAPF) is a coordination problem which asks us to find a set of collision-free paths for a team of mobile agents, each from its initial start location to its designated goal location. MAPF is a well known and well studied topic with numerous real-world applications. For example, MAPF appears as a core challenge in automated warehouse logistics [Wurman et al., 2008], in automated parcel sortation [Kou et al., 2020], in automated valet parking [Okoso et al., 2019], in computer games [Silver, 2006] and in a variety of other contexts [Ma et al., 2014], [Barer et al., 2015]. Many optimal approaches exist [Yu and LaValle, 2016; Surynek, 2018]. See also the survey by Felner et al. [2017].

Conflict-based Search (CBS) [Sharon et al., 2015] is a two-level optimal MAPF solver which is popular and successful. The low level finds optimal paths for the individual agents. If the paths include collisions, the high level, via a split action, imposes constraints on the agents to avoid these collisions. The search space of CBS is therefore a binary Conflict Tree (CT) which the algorithm explores in best-first order. CBS is complete, optimal and often highly performant; e.g., recent variants [Li et al., 2019a; 2019b; 2019c] can solve MAPF problems with > 100 agents. Unfortunately all these algorithms suffer from the same significant drawback — as the search continues they all need to store the entire search frontier, i.e., the entire open-list (OPEN), in memory. In CBS variants, the number of frontier nodes is exponential in the height of the CT, which means that available memory may be exhausted long before they have an opportunity to expand the goal node. For this reason experiments with CBS are always executed with short time limits of 1 to 5 minutes. From a practical perspective this means CBS will either find a solution quickly or it will fail and leave the practitioner without any recourse. There is no possibility to e.g., allow the algorithm to run longer.\textsuperscript{1}

In this work we introduce Iterative-Deepening CBS (ID-CBS), a new optimal MAPF algorithm which replaces the high-level A*-like search of CBS with IDA* [Korf, 1985] which is a search algorithm for exponential domains that is not memory limited. IDCBS explores the CT tree using repeated depth-first iterations. Unlike A*, DFS only moves from a parent node to its child and back and such nodes have many similarities in their content. This is a substantial change that requires some fundamental re-thinking of the CBS algorithm. As a second contribution, we identify 6 main components required to process a high-level CBS node and we show how each one can be improved using incremental data structures that exploit similarities between parent nodes and their children. Third, we undertake an extensive empirical comparisons which demonstrate practical benefits. IDCBS is able to optimally solve many more problems than CBS and we report substantial improvements in search times even for problems which can be solved by a currently leading CBS variant. As a fourth contribution, we show that incremental approaches can also be used to improve ECBS [Barer et al., 2014], a bounded suboptimal variant which we scale to problems thought to be substantially beyond the reach of current bounded suboptimal algorithms. In one headline result, we report the first known near-optimal solutions (within 1%) for MAPF problems with up to six hundred agents.

2 Definitions and Background

MAPF is defined by a graph \( G = (V, E) \) and a set of \( k \) agents \( \{a_1, \ldots, a_k\} \), where each agent \( a_i \) has a start location \( \text{start}_i \in V \) and a goal location \( \text{goal}_i \in V \). We assume time is discretised into timesteps of unit size. At each timestep every agent can either move to an adjacent vertex or wait at its

\textsuperscript{1}Bounded-suboptimal variants of CBS (see below) can mitigate this issue to a limited extent. They are still exponential, only with a smaller base.
Algorithm 1: High-level of CBS

1 Main(MAPF problem instance)
2 Root ← new CT node with no constraints
3 Root.solution ← \{a path for each agent\}
4 insert Root into OPEN
5 while OPEN not empty do
6 \( N \) ← node with best-\( f \) from OPEN // lowest \( f \)-cost cost
7 Delete \( N \) from OPEN
8 if \( N \) has no conflict then
9 \[ return N.solution // \( N \) is a goal node \]
10 Classify \( N \).conflicts into types
11 Compute \( N.h \) if it hasn’t been computed before
12 if \( N.f > OPEN.top().f \) then
13 \[ insert N back into OPEN \]
14 \[ continue \]
15 \( C \) ← Choose Conflict(\( N \)) // (Prioritize Conflicts)
16 Children ← \[
17 \text{foreach agent } a \text{ in } C = (a_i, a_j, v or e, t) \{\}
18 \[ N’ ← \text{Generate Child}(N, (a, v or e, t)) \]
19 if \( (N’.cost = N.cost) \) and
20 \( (N’.#conflicts < N.#conflicts) \) then // (BP)
21 \[ N.solution ← N’.solution \]
22 \[ Children ← [N] \]
23 \[ break \]
24 \[ insert N’ into Children \]
25 \[ insert Children into OPEN \]
26 \[ generate new CT node \]
27 \[ N’.constraints ← N.constraints \text{ + (a, v or e, t)} \]
28 \[ N’.solution ← N.solution \]
29 \[ CAT ← Build CAT(N.solution, i) \]
30 Update \( N’.solution \) by invoking low level(\( a_i, CAT \))
31 \[ N’.cost ← SIC(N’.solution) \]
32 \[ N’.conflicts ← \text{Find Conflicts}(N’.solution) \]
33 \[ return N’ \]

A path for agent \( a_i \) is a sequence of move/wait actions that takes the agent from \( \text{start}_i \) to \( \text{goal}_i \). We say that two agents are in collision (equiv. in conflict) if they attempt to occupy the same vertex at the same time or if they attempt to traverse the same edge at the same time. To represent vertex conflicts we use the notation \( \langle a_i, a_j, v, t \rangle \) which means agents \( a_i \) and \( a_j \) both attempt to occupy vertex \( v \) at timestep \( t \). When discussing edge conflicts we use the notation \( \langle a_i, a_j, (u, v), t \rangle \) which means agents \( a_i \) and \( a_j \) attempt to swap positions, by traversing edge \( (u, v) \) at the same time. A complete assignment of paths to agents is called a plan and any collision-free plan is a valid solution to the MAPF problem. Our objective is to find a valid plan whose sum-of-costs, across all constituent paths, is minimum.

2.1 Background: Conflict-Based Search (CBS)

CBS [Sharon et al., 2015] is a two-level MAPF algorithm. The CBS low level computes optimal paths for each agent individually, usually with some variant of A* running on a time-expanded graph, or TEG. Each vertex \( v_i \) in the TEG is a time-indexed location; i.e. it represents the vertex \( v \in V \) at timestep \( i \). The edges of \( v_i \) are similarly time indexed; i.e. each \( (v_i, v_{i+1}) \) represents a move; from vertex \( v \) at timestep \( i \) to vertex \( w \) at timestep \( i + 1 \). (see Figure 1(a)). During a low-level search each agent may be subject to a set of collision-avoiding constraints. A vertex constraint, written \( (a_i, v, t) \), prohibits agent \( a_i \) from occupying vertex \( v \) at timestep \( t \). An edge constraint, written \( (a_i, (u, v), t) \), prohibits agents \( a_i \) from traversing edge \( (u, v) \) at timestep \( t \).

The CBS high level performs a best-first search on a binary Conflict Tree (CT) where each node \( N \in CT \) is a complete (but possibly invalid) solution to the MAPF problem. Algorithm 1 shows the main steps. At Lines 2–3, CBS generates a CT root node where the tentative solution assigns to every agent its individually optimal shortest path (no constraints). At Lines 6–7, CBS removes the most promising candidate solution from OPEN and checks it for conflicts. If there are none, the search can terminate with an optimal solution (Line 9). Otherwise (Line 15), CBS chooses one of the remaining pairwise conflicts and works to resolve it. Resolving a conflict (Lines 16–24) means adding new conflict-avoiding constraints to each of the two agents and generating two new candidate solutions (Lines 26–33) corresponding to each of the conflicting agents being assigned a new and individually optimal paths, subject to constraints. Each new solution is added to OPEN (Line 24) and the process continues until a conflict-free solution is found or until the list is exhausted. The CT is explored in an A*-like search guided by \( f = g + h \) where \( g \) is the cost of the current plan and \( h \) is an admissible heuristic function (Lines 11-14), as suggested in [Felner et al., 2018]. In some cases (Lines 19-22) conflicts can be resolved without any branching by using an enhancement called bypassing, first suggested in [Boyarski et al., 2015a].

CBS decides which conflict to resolve (Line 15) by prioritizing the conflicts [Boyarski et al., 2015b]. The highest priority conflicts are called cardinal; both child nodes that are generated have a higher cost than the parent as the individual path cost for each of the agents increases. Next highest are semi-cardinal conflicts, which increase the path cost for just one of the conflicting agents. Finally are non-cardinal conflicts, which can be resolved without increasing any path cost. To classify each conflict CBS uses multi-value decision diagrams (MDDs) [Sharon et al., 2013]. An \( MDD^c \) is a DAG that compactly stores all paths of cost \( c \) for agent \( a_j \). Figure 1(e) shows an example for paths of cost 4 from \( S \) to \( G \) under some constraints. We call a set of MDD nodes that have the same time-step an MDD level. To determine if agent \( a_i \) must increase its cost CBS checks if the conflict location appears in \( MDD^c \) as the only node in the corresponding level.

3 Iterative Deepening CBS (IDCBS)

Traditionally (see papers listed above), the time limit for MAPF experiments was set at 1–5 minutes per problem instance. This was perhaps partly inspired by Ruml’s encouragement to use small benchmarks, and partly motivated by the need to average over many problem instances. However, another reason to set a short time
To overcome the memory issue, and let CBS run longer, we propose an iterative-deepening version of CBS, called IDCBS. Similarly to CBS, IDCBS may also use an admissible heuristic to guide its search. For simplicity, we hereafter use the terms CBS and IDCBS but they always include a heuristic function. Similarly to IDA*, in each iteration of IDCBS, the CT is traversed in f-limited DFS iterations, and the limit of the next iteration is set to the lowest f-value of a node that was generated in the previous iteration but not expanded. Optimality is guaranteed by the same proof as IDA*’s.

To demonstrate the advantages of IDCBS, we set a time limit of 1 hour for both algorithms and iteratively solved instances from the first scenario of each map in the standard MAPF benchmark [Stern et al., 2019], adding agents until we failed. Table 1 shows the results. IDCBS times out less often than CBS and also succeeds more often, producing optimal solutions for 190 instances that CBS did not solve.

Figure 2(a) gives further insight. Here we compare the average memory usage (out of 8GB available) for the two algorithms (y-axis) and as a function of CPU time per instance (in buckets of 2 minutes). Notice that as CBS runs longer, it needs more memory, while the memory needs of IDCBS remain relatively constant. In Section 4 we show how IDCBS can substantially improve the processing time per CT node vs CBS (see Figure 2(c)), which allows the algorithm to expand more CT nodes and find solutions faster. In Section 6 we will show that IDCBS is in general much stronger.

### 4 Reducing the Time-Per-Node via DFS

A modern CBS solver performs 6 main activities: (1) The low level path finding (Line 30). (2) Building MDDs which represent all possible paths of the same cost for an agent when necessary (Line 10). (3) Building a conflict avoidance table (CAT) [Standley, 2010] which tries to steer agents away from each other (Line 29) (4) A high-level heuristic computation (Line 11). (5) Finding all new conflicts that were caused by the newly-planned paths in a CT node (Line 32). And, (6) other high-level work, i.e. queuing and node construction operations to maintain the open list (remaining lines).

Figure 2(b) shows the breakdown of the average time per node of CBS to these 6 components as the numbers of agents increase over all problem instances that were solved by both algorithms. Clearly, each of the components takes a measurable part of the time, with only the high-level heuristic (component 5) and finding conflicts (components 4) being substantially smaller than the rest, for reasons described below.

Generating a node both under a best-first-search (denoted hereafter as BFS) or under a DFS is done in the context of the parent node. But, there is a fundamental difference between BFS and DFS. BFS jumps around the search tree. The internal information within a node is not passed between successive node expansions because these nodes can be very different and in different parts of the search space. Therefore, in BFS each node must have all the necessary information in its own data structures. By contrast, DFS only moves between parents and children (and back). So, a global current-node data-structure is maintained. When a node is generated, we calculate and store the ∆ between a parent and a child and undo this ∆ when backtracking. In the next subsections, we exploit this idea and for each of the components we introduce mechanisms that compute the ∆ from the parent efficiently, mostly using incremental algorithms. Incremental algorithms use the work of previous executions to speed up solving a different but relatively similar problem instance.

Figure 2(c) shows the same breakdown for IDCBS with the incremental methods presented next and shows almost an order of magnitude reduction in CPU time per node. The incremental approaches have reduced all the components independently of each other as seen in the figure. We next cover each component in turn.

#### 4.1 Component 1: Low-Level Path Finding

In every CT node a single constraint is added for one agent. Consequently, the low-level path finding task is very similar to the last path-planning task for this agent that was performed (in one of its ancestors). Therefore, executing the low-level search from scratch might be wasteful. Instead, we can use incremental search algorithms; they use information from previous similar searches to speed up the new search. We use Lifelong Planning A* (LPA*) [Koenig et al., 2004] for the low-level search and provide a brief overview of LPA* next (see the original paper on LPA* for a full description).

LPA* is an incremental version of A*. LPA* is invoked again when edges or vertices are added to or deleted from
the environment graph, or when the costs of some of the edges change. LPA* replans a new path by using information from the previous executions, thereby significantly reducing the effort needed to plan from scratch. Like A*, LPA* maintains a priority queue \( \mathcal{PEN} \) which contains only the locally inconsistent vertices. LPA* also maintains the \( \text{rhs-value} \) of each vertex \( v \), which represents the number of conflicts of the best known path of the given agent from \( \text{start} \) to \( v \) with the paths of other agents, as stored in the CAT. \( k_3 \) is used in case there was a tie in both \( k_1 \) and \( k_2 \).

In a first execution, LPA* behaves like A*. A new execution is invoked after some changes to the underlying graph were made. In new executions LPA* starts by first carrying over \( \mathcal{OPEN} \) and \( \mathcal{CLOSED} \) from the previous execution including the \( g \)-values and \( \text{rhs-values} \). LPA* then updates the \( \text{rhs-values} \) of the vertices which are direct neighbors of the \( \text{changed components} \), e.g., edge with a new updated cost. It also updates \( \mathcal{OPEN} \) to reflect new consistent and inconsistent vertices from this set (this is a relatively cheap operation). Then, LPA* repeatedly expands vertices until the goal is locally consistent and its key is smaller than the smallest key in \( \mathcal{OPEN} \). In this case, its \( g \)-value is the cost of the shortest path from the start vertex. This allows LPA* to focus only on inconsistent vertices and it was shown that this requires less work than running A* from scratch on the new graph.

**Using LPA* for the Low-Level of CBS**

Adding a constraint \( \langle a_i, v, t \rangle \) to a CT node \( N \) can be viewed as deleting vertex \( v \) from the TEG. Thus, the TEG of \( N \) is very similar to the previous TEG which was used the last time a path for \( a_i \) was needed (in an ancestor of \( N \)). This calls for using LPA* to search the new TEG instead of using A*.

However, a few modifications are in order in this scenario.

(1) LPA* assumes a single goal node. For example, a typical path-finding execution terminates when the presumably-known goal node is consistent and smaller than all other nodes in \( \mathcal{OPEN} \). But, in a TEG, the goal location appears in many time-steps. To enable LPA* to search a TEG, we add a single vertex \( g_{\text{final}} \) to the TEG that serves as LPA*’s single goal node. Each goal node \( g \) in the TEG is connected with a zero-cost (dummy) edge to this new goal node. Luckily, the low-level heuristic only takes the location into account (the time step is ignored) making it robust across different time steps.

(2) LPA* performs better when the changes in the graph occur closer to the goal. LPA* will re-expand a smaller subtree in this case [Koenig et al., 2004]. Therefore, as a tie-breaker (e.g., among cardinal conflicts), choose the closest conflict to the goals of the conflicting agents.

(3) To make use of a CAT, LPA* keys need a third component \( k_3(v) \) that represents the number of conflicts of the best known path of the given agent from \( \text{start} \) to \( v \) with the paths of other agents, as stored in the CAT. \( k_3 \) is used in case there was a tie in both \( k_1 \) and \( k_2 \).

\( k_3(v) \) may be out of date if the paths of other agents changed from the time that \( k_3(v) \) was computed in a previous execution of LPA*. In this case \( v \) remained consistent since then and \( k_3(v) \) was not recomputed. This can cause LPA* to find paths with new non-cardinal or semi-cardinal conflicts that could have been avoided if \( k_3(v) \) were fully up-to-date.\(^2\)

This problem is mitigated by continuing the LPA* execution until all paths of the same cost are found, and taking the path with the smallest number of conflicts by consulting the CAT. However, this is only applied when the CT node has no cardinal conflicts, because when a cardinal conflict is resolved, other conflicts are likely to be indirectly resolved and this process will be redundant. An additional mitigation of this problem is to use the Bypassing Conflicts (BP) enhancement [Boyarski et al., 2015a] (Line 19 of Algorithm 1). BP resolves some non-cardinal or semi-cardinal conflicts without increasing the size of the high-level search tree. When a child CT node is found to have the same sum of costs as its parent, and fewer conflicts, the replanned path is copied into the parent node, replacing the parent’s path for the agent. The updated parent node can then be re-inserted into \( \mathcal{OPEN} \), and the child node is discarded. BP is crucial in IDCBS because it further resolves the issue of increased numbers of non-cardinal and semi-cardinal conflicts.

### 4.2 Component 2: Computing MDD Levels

Recall that CBS builds \( MDD^0_i \) in the context of the CT node \( N \) taking the constraints on \( a_i \) into account. Building \( MDD^0_i \) non-incrementally is implemented as a two-stage process with forward and backward searches. First, a forward A* over the TEG is executed from \( \text{start}^0_i \) under the constraints that are imposed on agent \( a_i \). Backward MDD

\(^2\)This only happens for LPA* because it reuses information. It would not happen for A* as it runs from scratch considering the current up-to-date paths of other agents.
edges from each node back to all of its possible parents are constructed. This search does not stop when \( \text{goal}_i \) is expanded, but continues to expand all low-level nodes \( v \) such that \( f(v) \leq c \) and does not generate low-level nodes \( v \) such that \( f(v) > c \). At the end of this stage, we have generated all nodes with \( f \leq c \) even if they are not part of a path of cost \( c \) to the goal. Such surplus nodes are pruned next. Second, a backward breadth-first search is performed over the nodes generated in the forward search. The search starts from \( \text{goal}_i \), and generates forward MDD edges to \( \text{goal}_i \) from all nodes \( v \) for which (1) \( (v, \text{goal}_i) \in E \) and (2) \( t = c - 1 \). The search continues backward in this way until \( \text{start}_i \) is reached. All generated nodes that do not have forward edges (from the backward search) connected to them at the end of the backward search are repeatedly deleted, and the resulting doubly-connected structure is \( MDD^r_i \).

Figure 1(c–e) show the stages of building an MDD of depth 4 for agent \( a_1 \) from vertex \( S \) to vertex \( G \) on a graph in Figure 1(a). In this example, assume that another agent is planned to pass through \( a_1 \)'s goal taking a unidirectional edge into vertex \( X \) from some other vertex in the graph (not in the figure), moving from \( X \) to \( G \), and taking a unidirectional edge out of \( G \) and exiting the example. CBS has eventually added vertex constraints for \( a_1 \) on \( X \) and time step 2 and on \( G \) and time step 3, and an edge constraint on moving in the opposite direction from \( G \) to \( X \) at time step 2 (red marked crosses). Figure 1(c) shows all the nodes that were generated at the end of the first stage. Figure 1(d) shows the result of the second stage. Figure 1(e) shows the MDD that resulted from pruning nodes that don’t lead to \( G \) (e.g., \( G' \)).

This two-phase breadth-first search process is relatively costly. The cost is especially problematic when CBS uses a high-level heuristic, because MDDs are needed for every agent that has a vertex in the conflict graph (see definition below). Moreover, when a new MDD is needed due to new constraints, CBS builds it from scratch. This can be mitigated as follows. When using LPA* for the low-level search the MDD can be cheaply constructed. Every time LPA* halts with the new path, for every node \( v \) with \( k(v) < k(\text{goal}) \), \( g(v) \) holds the true distance from the start vertex to \( v \). So, when a new MDD is needed, IDCBS resumes the LPA* execution for \( a_i \), now expanding all nodes \( v \) with \( k(v) = k(\text{goal}) \) and finishing the forward search part of building the MDD with very little effort. IDCBS then performs a similar backward search to complete the MDD. Thus, not only the forward search effort is saved, but also the pruning effort in the backward search.

4.3 Component 3: Building the CAT

Consider a CT node where a path for agent \( a_i \) is needed by CBS. A Conflict-Avoidance Table (CAT) [Standley, 2010] holds the location-time pairs that make up the paths of all other agents in the CT node. While replanning a path for agent \( a_i \), the low level of CBS may use the CAT to break ties among paths of the same cost in favor of paths that cause fewer conflicts with the current paths of the other agents as given by the CAT. Because the size of the CAT is relatively large it is usually built from scratch for every new node mainly to save memory (Line 29).

In IDCBS, instead of building the CAT from scratch each time the low-level is invoked, a single global CAT is maintained throughout the search. This table holds location-time pairs of the paths of all the agents in the current CT node. The CAT is initialized with the paths in the root node of the CT. Every time a CAT is needed, the path that is going to be replanned is removed from the CAT. Then, when the low-level path planning finishes, the newly planned path is incorporated into the CAT and DFS continues. When the DFS backtracks, the same is done in reverse. Thus, for a MAPF instance with \( k \) agents, one path is removed from the CAT once and inserted into it once, instead of \( k \) paths being inserted into a newly generated CAT for each new node.

4.4 Component 4: Finding Conflicts in a CT Node.

Since in both BFS and DFS traversals the child node is constructed from its parent, some parts of CBS are already usually implemented in an incremental way. When generating a new CT node, only the newly-planned paths are checked for conflicts with all the other paths. The conflicts between the paths of the agents whose paths remain unchanged are carried over directly from the new node’s parent.

4.5 Component 5: Computing the Heuristic

For each CT node \( N \) the size of the minimum vertex cover (MVC) of a corresponding cardinal-conflict graph is used as an admissible heuristic for the remaining cost to reach a goal node in the subtree rooted at \( N \) [Felner et al., 2018]. The \( h \)-value of the root node is computed from scratch. Child nodes of a node with MVC of \( k \) only need to check whether the MVC is of size \( k - 1 \) and possibly also \( k \). Otherwise, it is \( k + 1 \) (see [Felner et al., 2018]). We tried a different approach of using a Mixed Integer Programming (MIP) model to directly compute the MVC. Modern MIP solvers reuse information from previous computations. We exploited that by maintaining the same MIP model, adding or removing edges from the cardinal conflict graph as needed. It turned out that this is far faster than the previous approach, so we use it for both CBS and IDCBS throughout our experimentation.

4.6 Component 6: High-Level Work

Maintaining a stack is cheaper than maintaining a priority queue, so this runtime activity is also faster in IDCBS.

5 Incremental Methods for Best-First CBS

BFS methods can also make use of incremental data structures by an approach that we denote Lowest Common Ancestor Jumping (LCA-jumping). LCA-jumping is used in other BFS algorithms with large memory requirements per node such as Mixed Integer Programming and Constraint Programming [Région and Malapert, 2018]. In this approach a node on the frontier is stored with its \( f \)-value and the \( \Delta \) from its parent (like DFS). When we decide to move from current CT node \( N \) to the next best node \( N' \) somewhere else in the tree, we find their lowest common ancestor \( A \) and backtrack from \( N \) to \( A \) and then move forward from \( A \) to \( N' \) while updating the single current-node data structures incrementally during this traversal. Using LCA-jumping we can perform the same BFS as CBS but much more efficiently.
In order to reduce the path of LCA-jumping, it is preferable to break ties among CT nodes with the same $f$-value in favor of a node $N'$ that is closer in the CT to $N$. Normally, CBS breaks ties in favor of nodes with fewer conflicts. A limit can be placed on the number of consecutive times a child node is expanded immediately after its parent at least 90% of the time, and even more as $w$ increases. This results in a DFS-like traversal. We applied LCA-jumping to CBS to make its high level only move between parent and child nodes, enabling our ideas for reducing the time-per-node to be used.

6 Experimental Results

All our experiments were run on a Linux laptop with an Intel Core i7-8650U CPU running at 1.9GHz.

Main Long Experiment

We experimented on the MAPF benchmarks [Stern et al., 2019] which contain 32 grids with different attributes, each with a number of scenarios=start and goal locations for up to 7,000 agents). We increased the number of agents on these scenarios until we reached the runtime limit (1 hour) or memory limit (8GB). For higher numbers of agents the solver was considered an implicit fail.

Table 1 (page 3) shows the number of solved instances and number of different failures for CBS and IDCBS. Out of 3380 MAPF instances, 190 were solved by IDCBS and not by CBS, 45 were solved only by CBS and not by IDCBS, and 64 were not solved by either algorithm (the difference is statistically significant). To the best of our knowledge, those 190 problem instances have never been solved optimally before. Table 2 compares the average number of nodes generated by CBS and IDCBS. Similarly to IDA*, on exactly the same tree, IDCBS generates more CT nodes than CBS. In practice, however, the table shows that in some cases IDCBS generated fewer nodes than CBS. This is because IDCBS uses different low-level path finding (A* vs. LPA*) and chooses conflicts to resolve differently. So, their CT might be different. Figure 3 shows the average runtime for all the problem instances that were solved by both CBS and IDCBS. Note that the scale is logarithmic. IDCBS outperforms CBS in its CPU time by up to two orders of magnitude.

LCA-jumping

Finally, to demonstrate the effectiveness of LCA-jumping, we ran experiments with ECBS variants with 1 minute timeout on the same benchmarks. We used a variant of ECBS (called BCBS [Barer et al., 2014]) where only the high-level search is bounded-suboptimal and the low-level search is optimal. Figure 4 compares the success rate of three instances of ECBS ($w \in \{1, 1.01, 1.05\}$) with and without LCA-jumping and our incremental methods. Our enhanced solvers successfully solve up to twice as many agents (600! agents vs. 300), with a success rate up to 3 times higher.

7 Summary and Conclusions

CBS is a memory-intensive algorithm since its usual implementations store significant information for each CT node in its search frontier. This has gone largely unnoticed since the practice has been to run CBS for very short time limits. In this work we present a memory-efficient version of CBS using iterative deepening and incremental algorithms to update data.

There is huge scope for future work. In particular we need to compare different types of traversal of the CT, e.g. to discover when iterative deepening is preferable to LCA-jumping and experiment with other traversals. Since exploration of CT nodes is significantly cheaper we need to re-visit many of the design choices of CBS that were handled in this paper. We should investigate how incremental data structures affect other variants such as Lazy CBS [Gange et al., 2019], and CBSwP and PBS [Ma et al., 2019].
References


