Meta-agent Conflict-Based Search For Optimal Multi-Agent Path Finding

Guni Sharon
ISE Department
Ben-Gurion University
Israel
gunisharon@gmail.com

Roni Stern
ISE Department
Ben-Gurion University
Israel
roni.stern@gmail.com

Ariel Felner
ISE Department
Ben-Gurion University
Israel
felner@bgu.ac.il

Nathan Sturtevant
CS Department
University of Denver
USA
Sturtevant@cs.du.edu

Abstract

The task in the *multi-agent path finding* problem (MAPF) is to find paths for multiple agents, each with a different start and goal position, such that agents do not collide. It is possible to solve this problem optimally with algorithms that are based on the A* algorithm. Recently, we proposed an alternative algorithm called Conflict-Based Search (CBS) (Sharon *et al.* 2012), which was shown to outperform the A*-based algorithms in some cases. CBS is a two-level algorithm. At the high level, a search is performed on a tree based on conflicts between agents. At the low level, a search is performed only for a single agent at a time. While in some cases CBS is very efficient, in other cases it is worse than A*-based algorithms. This paper focuses on the latter case by generalizing CBS to *Meta-Agent* CBS (MA-CBS). The main idea is to couple groups of agents into *meta-agents* if the number of internal conflicts between them exceeds a given bound. MA-CBS acts as a framework that can run on top of any complete MAPF solver. We analyze our new approach and provide experimental results demonstrating that it outperforms basic CBS and other A*-based optimal solvers in many cases.

Introduction and Background

In the *multi-agent path finding* (MAPF) problem, we are given a graph, $G(V, E)$, and a set of k agents labeled $a_1 \ldots a_k$. Each agent a_i has a start position $s_i \in V$ and goal position $g_i \in V$. At each time step an agent can either *move* to a neighboring location or can *wait* in its current location. The task is to return the least-cost set of actions for all agents that will move each of the agents to its goal without *conflicting* with other agents (i.e., without being in the same location at the same time or crossing the same edge simultaneously in opposite directions). MAPF has practical applications in robotics, video games, vehicle routing, and other domains (Silver 2005; Dresner & Stone 2008). In its general form, MAPF is NP-complete, because it is a generalization of the sliding tile puzzle, which is NP-complete (Ratner & Warrnuth 1986).

There are many variants to the MAPF problem. In this paper we consider the following common setting. The cumulative cost function to minimize is the sum over all agents of the number of time steps required to reach the goal location (Standley 2010; Sharon *et al.* 2011a). Both *move*

and *wait* actions cost one. A *centralized computing* setting with a single CPU that controls all the agents is assumed. Note that a *centralized computing* setting is logically equivalent to a *decentralized setting* where each agent has its own computing power but agents are fully cooperative with full knowledge sharing and free communication.

There are two main approaches for solving the MAPF in the *centralized computing* setting: the *coupled* and the *decoupled* approaches. In the *decoupled approach*, paths are planned for each agent separately. Algorithms from the *decoupled* approach run relatively fast, but optimality and even completeness are not always guaranteed (Silver 2005; Wang & Botea 2008; Jansen & Sturtevant 2008). New complete (but not optimal) *decoupled* algorithms were recently introduced for trees (Khorshid, Holte, & Sturtevant 2011) and for general graphs (Luna & Bekris 2011).

Our aim is to solve the MAPF problem optimally and therefore the focus of this paper is on the *coupled approach*. In this approach MAPF is formalized as a global, single-agent search problem. One can activate an A*-based algorithm that searches a state space that includes all the different ways to permute the k agents into $|V|$ locations. Consequently, the state space that is searched by the A*-based algorithms grow exponentially with the number of agents. Hence, finding the optimal solutions with A*-based algorithms requires significant computational expense.

Previous optimal solvers dealt with this large search space in several ways. Ryan (2008; 2010) abstracted the problem into pre-defined structures such as cliques, halls and rings. He then modeled and solved the problem as a CSP problem. Note that the algorithm Ryan proposed does not necessarily returns the optimal solutions. Standley (2010; 2011) partitioned the given problem into smaller independent problems, if possible. Sharon et. al. (2011a; 2011b) suggested the *increasing cost search tree* (ICTS) - a two-level framework where the high-level phase searches a tree with exact path costs for each of the agents and the low-level phase aims to verify whether there is a solution of this cost.

In this paper we focus on the new Conflict Based Search algorithm (CBS) (Sharon *et al.* 2012) which optimally solves MAPF. CBS is a two-level algorithm where the high-level search is performed on a *constraint tree* (CT) whose nodes include constraints on time and locations of a single agent. At each node in the constraint tree a low-level search

is performed to find individual paths for all agents under the constraints given by the high-level node.

Sharon et al. (2011a; 2011b; 2012) showed that the behavior of optimal MAPF algorithms can be very sensitive to characteristics of the given problem instance such as the topology and size of the graph, the number of agents, the branching factor etc. There is no universally dominant algorithm; different algorithms work well in different circumstances. In particular, experimental results have shown that CBS can significantly outperform all existing optimal MAPF algorithms on some domains (Sharon *et al.* 2012). However, Sharon et al. (2012) also identified cases where the CBS algorithm performs poorly. In such cases, CBS may even perform exponentially worse than A*.

In this paper we aim at mitigating the worst-case performance of CBS by generalizing CBS into a new algorithm called *Meta-agent CBS* (MA-CBS). In MA-CBS the number of conflicts allowed at the high-level phase between any pair of agents is bounded by a predefined parameter B. When the number of conflicts exceed B, the conflicting agents are merged into a *meta-agent* and then treated as a joint composite agent by the low-level solver. By bounding the number of conflicts between any pair of agents, we prevent the exponential worst-case of basic CBS. This results in an new MAPF solver that significantly outperforms existing algorithms in a variety of domains. We present both theoretical and empirical support for this claim. In the low-level search, MA-CBS can use any complete MAPF solver. Thus, MA-CBS can be viewed as a solving framework and future MAPF algorithms could also be used by MA-CBS to improve its performance.

Furthermore, we show that the original CBS algorithm corresponds to the extreme cases where $B = \infty$ (never merge agents), and the Independence Dependence (ID) framework (Standley 2010) is the other extreme case where $B = 0$ (always merge agents when conflicts occur). Thus, MA-CBS allows a continuum between CBS and ID, by setting different values of B between these two extremes.

The Conflict Based Search Algorithm (CBS)

The MA-CBS algorithm presented in this paper is based on the CBS algorithm (Sharon *et al.* 2012). We thus first describe the CBS algorithm in detail.

Definitions for CBS We use the term *path* only in the context of a single agent and use the term *solution* to denote a set of k paths for the given set of k agents. A *constraint* for a given agent a_i is a tuple (a_i, v, t) where agent a_i is prohibited from occupying vertex v at time step t.[1] During the course of the algorithm, agents are associated with constraints. A *consistent path* for agent a_i is a path that satisfies all its constraints. Likewise, a *consistent solution* is a solution that is made up from paths, such that the path for agent a_i is consistent with the constraints of a_i. A *conflict* is a tuple (a_i, a_j, v, t) where agent a_i and agent a_j occupy vertex v at time point t. A solution (of k paths) is *valid* if all its

---

[1] A conflict (as well as a constraint) may apply also to an edge when two agents traverse the same edge in opposite directions.

paths have no conflicts. A consistent solution can be *invalid* if, despite the fact that the paths are consistent with their individual agent constraints, these paths still have conflicts.

The key idea of CBS is to grow a set of constraints for each of the agents and find paths that are consistent with these constraints. If these paths have conflicts, and are thus invalid, the conflicts are resolved by adding new constraints. CBS works in two levels. At the high-level phase conflicts are found and constraints are added. At the low-level phase, the paths of the agents are updated to be consistent with the new constraints. We now describe each part of this process.

High-level: Search the Constraint Tree (CT)

At the high-level, CBS searches a *constraint tree* (CT). A CT is a binary tree. Each node N in the CT contains the following fields of data:

1. **A set of constraints** ($N.constraints$). The root of the CT contains an empty set of constraints. The child of a node in the CT inherits the constraints of the parent and adds one new constraint for one agent.

2. **A solution** ($N.solution$). A set of k paths, one path for each agent. The path for agent a_i must be consistent with the constraints of a_i. Such paths are found by the low-level search algorithm.

3. **The total cost** ($N.cost$). The cost of the current solution (summation over all the single-agent path costs). We denote this cost the f-value of the node.

Node N in the CT is a goal node when $N.solution$ is valid, i.e., the set of paths for all agents have no conflicts. The high-level phase performs a best-first search on the CT where nodes are ordered by their costs.

Processing a node in the CT Given the list of constraints for a node N of the CT, the low-level search is invoked. This search returns one shortest path for each agent, a_i, that is consistent with all the constraints associated with a_i in node N. Once a consistent path has been found for each agent with respect to its constraints, these paths are then *validated* with respect to the other agents. The *validation* is performed by simulating the set of k paths. If all agents reach their goal without any conflict, this CT node N is declared as the goal node, and the current solution ($N.solution$) that contains this set of paths is returned. If, however, while performing the *validation* a conflict $C = (a_i, a_j, v, t)$ is found for two (or more) agents a_i and a_j, the validation halts and the node is declared as a non-goal node.

Resolving a conflict Given a non-goal CT node N whose solution $N.solution$ includes a *conflict* $C_n = (a_i, a_j, v, t)$ we know that in any valid solution at most one of the conflicting agents (a_i or a_j) may occupy vertex v at time t. Therefore, at least one of the constraints (a_i, v, t) or (a_j, v, t) must be added to the set of constraints in $N.constraints$. To guarantee optimality, both possibilities are examined and N, is split into two children. Both children inherit the set of constraints from N. The left child resolves the conflict by adding the constraint (a_i, v, t) and the right child adds the constraint (a_j, v, t).
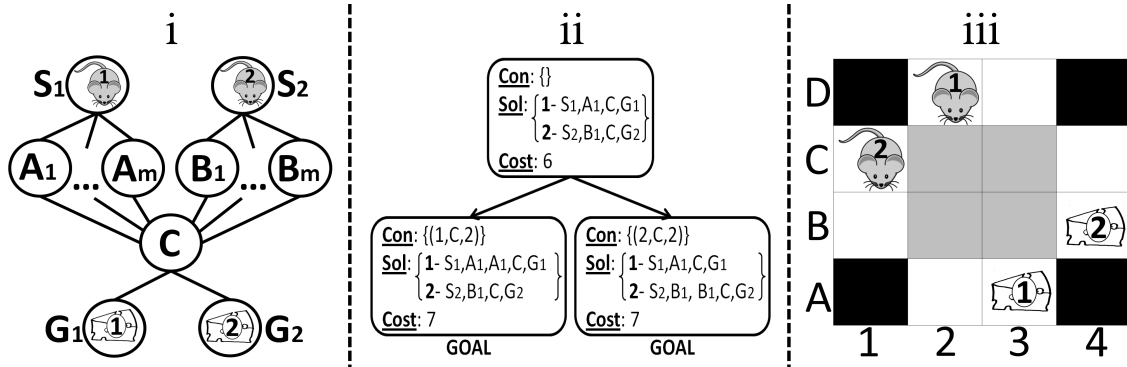
Figure 1: (i) MAPF example (ii) CT (iii) A case where A* outperforms CBS.

Algorithm 1: high-level of CBS (and MA-CBS)

Input: MAPF instance

1   $R.constraints = \emptyset$
2   $R.solution$ = find individual paths by the low-level()
3   $R.cost = SIC(R.solution)$
4   insert R to OPEN
5   **while** OPEN *not empty* **do**
6   P ← best node from OPEN // *lowest solution cost*
7   Validate the paths in P until a conflict occurs.
8   **if** *P has no conflict* **then**
9   **return** P.solution // *P is goal*
10   C ← first conflict (a_i, a_j, v, t) in P
11   **if** *should-merge(a_i, a_j) // Optional, MA-CBS only* **then**
12      $a_{i,j}$ = merge(a_i, a_j)
13   Update P.constraints().
14   Update P.solution by invoking low-level$(a_{i,j})$
15   Insert P to OPEN
16   continue // go back to the while statement
17   **foreach** *agent a_i in C* **do**
18   A ← new node
19   A.constraints ← P.constraints + (a_i, v, t)
20   A.solution ← P.solution.
21   Update A.solution by invoking low-level(a_i)
22      $A.cost = SIC(A.solution)$
23   Insert A to OPEN

Note that for a given CT node N one does not have to save all its cumulative constraints. Instead, it can save only its latest constraint and extract the other constraints by traversing the path from N to the root via its ancestors. Similarly, with the exception of the root node, the low-level search should only be performed for agent a_i which is associated with the newly added constraint. The paths of other agents remain the same as no new constraint was added for them.

CBS Example

Pseudo-code for CBS is shown in Algorithm 1. We note that lines (11-16) are to be ignored for basic CBS (that is, the should-merge() function (Line 11) always returns *false* for basic CBS). These lines will be added later for MA-CBS.

CBS has the structure of a best-first search. We cover it using the example in Figure 1(i), where the mice need to get to their respective pieces of cheese. The corresponding CT is shown in Figure 1(ii). The root contains an empty set of constraints. In line 2 the low-level now returns an optimal solution for each agent, $< S_1, A_1, C, G_1 >$ for a_1 and $< S_2, B_1, C, G_2 >$ for a_2. Thus, the total cost of this node is 6. All this information is kept inside this node. The root is then inserted into the sorted OPEN list and will be expanded next.

When validating the two-agent solution given by the two individual paths (line 7), a conflict is found when both agents arrive to vertex C at time step 2. This creates the conflict $(a_1, a_2, C, 2)$. As a result, the root is declared as non-goal and two children are generated in order to resolve the conflict (Line 17). The left child, adds the constraint $(a_1, C, 2)$ while the right child adds the constraint $(a_2, C, 2)$. The low-level search is now invoked (Line 21) for the left child to find an optimal path that also satisfies the new constraint. For this, a_1 must wait one time step either at S_1 (or at A_1) and the path $< S_1, A_1, A_1, C, G_1 >$ is returned for a_1. The path for a_2, $< S_2, B_1, C, G_2 >$ remains unchanged in the left child. The total cost for the left child is now 7. In a similar way, the right child is generated, also with cost 7. Both children are added to OPEN (Line 23). In the final step the left child is chosen for expansion, and the underlying paths are validated. Since no conflicts exist, the left child is declared as a goal node (Line 9) and its solution is returned as an optimal solution.

It may be the case that while performing the validation (Line 7) between the different paths a *k-agent conflict* is found for $k > 2$. There are two ways to handle such k-agent conflicts. We can generate k children, each of which adds a constraint to $k-1$ agents (i.e., each child allows only one agent to occupy the conflicting vertex v at time t). Or, an equivalent formalization is to only focus on the first two agents that are found to conflict, and only branch according to their conflict. This leaves further conflicts for deeper levels of the tree. For simplicity of description we chose the second option.

Low-level: Find Solutions for CT Nodes

The low-level is given an agent, a_i, and a set of associated constraints. It performs a search in the underlying

graph to find an optimal path for agent a_i that satisfies all its constraints. Agent a_i is solved in a *decoupled manner*, i.e., while ignoring the other agents. This search is three-dimensional, as it includes two spatial dimensions, and one dimension of time. Any single-agent path-finding algorithm can be used to find the path for agent a_i, while verifying that the constraints are satisfied. We used A* with a perfect heuristic in the two spatial dimensions. To get this heuristic, we pre-calculated and stored the all-pairs shortest path information. Whenever a state x is generated with $g(x) = t$ and there exists a constraint (a_i, x, t) in the current CT node this state is discarded.

Additionally, in this A* search ties were broken by using a *conflict avoidance table* (CAT) (Standley 2010)). The CAT is initialized by the current solution of node N, storing the number of agents passing via a given vertex at a given time, according to this solution. When two low-level states have the same f-values, the state with the smallest number of conflicts in the low-level CAT is preferred. This tie-breaking mechanism guides the low-level search towards solutions with less inter-agent conflicts. As a result, the optimal solution (which has no conflicts) is found faster.

Meta-agent Conflict Based Search (MA-CBS)

In this section we present our new generalized framework, MA-CBS. First, we motivate it by focusing in the limitations of the basic version of CBS that was just described.

Motivation for Meta-Agent CBS

In (Sharon *et al.* 2012) we showed that CBS is very efficient (compared to other approaches) for some MAPF problems and very inefficient for others. In general, it was shown that in domains with many bottlenecks, doorways and narrow passages, A* based algorithms (as well as ICTS) might do exponential work while CBS can solve the problem much faster by resolving a small number of conflicts.

Such a case is presented in Figure 1(i) in which both agents have m different routes to reach a bottleneck vertex (C). In this example, CBS generates three CT nodes. First, the root node is generated, invoking the low-level for the two agents, and expanding a total of 8 low-level nodes for the CT root. Now, a conflict is found at C and the CT root node is split into two children. In the left child the low-level searches for an alternative path for agent a_1 that does not pass through C at time step 2. S_1 plus all m states $A_1, \ldots, A_m$ are expanded with $f = 3$. Then, C and G_1 are expanded with $f = 4$ then the search halts and returns the path $< S_1, A_1, A_1, C, G_1 >$. Thus, at the left child a total of $m+3$ nodes are expanded. Similar $m+3$ states are expanded for the right child. Adding all these to the 8 states expanded at the root we get a total of $2m + 14$ low-level node expansions. For the same problem A*, which runs in a 2-agent search space, will expand $m^2 + 3$ nodes. For $m \geq 5$ this is larger than $2m + 14$ and consequently, CBS will expand fewer nodes. A* must expand the cross product of all single-agent paths with $f = 3$. By contrast, CBS must only try two such paths to realize that no solution of cost 6 is valid. Furthermore, the nodes counted for A* are multi-agent nodes

while for CBS, they are single-agent states. This is another advantage of CBS – smaller constant time per node.

By contrast, there are cases where many conflicts exist and CBS is very inefficient compared to the A* variants. Figure 1(iii) presents such a case where A* outperforms CBS. In this problem there are 4 optimal paths for each agent but each of the 16 paths combinations has a conflict in one of the gray cells. Consequently, one agent must wait at least one step to avoid collision. For this problem A* will expand 5 nodes with $f = 8$ and 3 nodes with $f = 9$ until the goal is found and a total of 8 nodes are expanded. Now, consider CBS. Each agent has 4 different optimal paths. All 16 combinations have conflicts in one of the 4 gray cells $\{C2, C3, B2, B3\}$. Therefore, for $f = 8$ a total of 16 CT nodes will be expanded, each will expand 4 low-level single-agent states to a total of $16 \times 4 = 64$ low-level nodes. Then, at the goal CT node with $f = 9$, CBS will expand 7 new states. Thus, a total of 71 states are expanded for CBS.

This general tendency that different MAPF algorithms behave differently for different environments or topologies was already seen in previous work (Sharon *et al.* 2011a; 2012). Furthermore, a given domain might have different areas with different topologies. This calls for an algorithm that will change its strategy based on the exact task and on the area it searches in. There is room for a significant amount of research in understanding topologies. We provide a first step in the context of CBS by dynamically grouping agents into a *meta-agent* and solving them in the low-level phase by a coupled algorithm (e.g., A*).

The two examples in Figure 1 suggest that in some cases it is more efficient to jointly plan for all agents, while in other cases it is more efficiently to plan for each agent independently. In general, CBS behaves poorly when a group of agents is *strongly coupled*, i.e., when there is a high rate of internal conflicts between agents in the group. In such cases, basic CBS may encounter many conflicts before finding the optimal solution.

In some cases, the number of conflicts encountered by CBS, for a given group of agents, is so large that it would have been more efficient to solve that group optimally with a coupled solver. MA-CBS exploits this by identifying groups of strongly coupled agents and merging them into a *meta-agent*. Then, the high-level CBS continues, but this meta-agent is treated as a single agent. The low-level search therefore solves it in a coupled manner with any MAPF solver. To this end, the low-level solver of CBS can be any MAPF solver, e.g., A*+OD (Standley 2010) or Enhanced Partial Expansion A* (Felner *et al.* 2012). Thus, MA-CBS is in fact a framework that can be used on top of any other MAPF solver. Next, we describe MA-CBS in detail.

Merging Agents Into a Meta-Agent

The main difference between basic CBS and MA-CBS is the new operation of merging agents into a *meta-agent*. A meta-agent consists of M agents, each agent is associated with its own position. Thus, a single agent is just a meta-agent of size 1. Returning to Algorithm 1, we introduce the merging action which occurs just after a new conflict was found by

the validation process (C in line 10) for a given CT node. At this point MA-CBS has two options:

- **Branch:** Branch into two CT nodes based on a new conflict (lines 17-23). This is the basic option which is performed by the basic CBS.

- **Merge:** Merge the two conflicting (meta) agents into a single meta-agent (Lines 12-16). This is a new option.

The merging process is performed as follows. Assume a CT node N with k agents. Suppose that agents a_1, a_2 were chosen to be merged. We now have $k - 1$ agents with a new *meta-agent* of size 2, labeled $a_{1,2}$. This meta-agent will never be split again in the subtree of the CT below this given node; it might, however, be merged with other (meta) agents to new meta-agents. Since nothing changed for the other agents that were not merged, we now only call the low-level search again for this new meta-agent (Line 14). The low-level search for a meta-agent of size M is in fact an optimal MAPF problem for M agents, and is solved with a coupled MAPF solver (e.g., A*).

Note that the cost of this CT node may increase due to this action, as the optimal path for a meta-agent is at least as large as the sum of optimal paths of each of these agents separately. Thus, we recalculate the f-value of this node and add it again into OPEN to its new location. (Line 15).

MA-CBS has two important components. First, MA-CBS requires a *merging policy* to decide which option to choose (branch or merge) (Line 11). Second, MA-CBS requires a mechanism to define the constraints imposed on the new meta-agent (Line 13). This mechanism that merges the constraints must be designed such that MA-CBS still returns an optimal solution. Next, we discuss how to implement these two components.

Merge Policy

Many merging policies are possible. We present a simple merging policy which we have found to be experimentally efficient. In our merging policy we identify when agents should be merged using a *bound parameter*, B. Two agents a_i, a_j are merged into a meta-agent $a_{i,j}$ if the number of conflicts between a_i and a_j seen so far during the search exceeds B. We use the notation MA-CBS(B) to denote MA-CBS with a bound of B. Clearly, basic CBS is the special extreme case of MA-CBS(∞). That is, we never choose to merge and always branch according to a conflict. The other extreme case is MA-CBS(0), where we only allow 0 conflicts but merge as soon as a conflict occurs. Below, we show that this case is identical to the ID framework.

To implement this merge policy, a conflict matrix CM is maintained. $CM[i, j]$ stores the number of conflicts between agents a_i and a_j seen so far by MA-CBS. After a new conflict between a_i and a_j is found (Line 10) $CM[i, j]$ is incremented by 1. Now, if $CM[i, j] > B$ the *should-merge()* function (Line 11) returns true and a_i and a_j are merged into $a_{i,j}$. Again, other merging policies are possible.

Merging Constraints

Next, we describe how to merge the constraint imposed by each of the merged agents. Denote a meta-agent by $\overline{x}$. A *meta-constraint* for a given meta-agent $\overline{x}$ is a tuple $(\overline{x}, v, t)$ where any individual agent $x_i \in \overline{x}$ is prohibited from occupying vertex v at time step t. Similarly, a *meta-conflict* is a tuple $(\overline{x}, \overline{y}, v, t)$ where an individual agent $x' \in \overline{x}$ and an individual agent $y' \in \overline{y}$ occupy vertex v at time point t. It is important to note that the exact identity x' and y' is irrelevant because we will later split the CT node according to meta-agents, either allowing $\overline{x}$ in v at time t and constraining $\overline{y}$ from being in v at time t, or vice versa.

We want to merge the constraints of a_i and a_j when they are merged into a meta-agent $a_{i,j}$. Consider the set of constraints associated with agents a_i and a_j before the merge. These were generated due to conflicts between agents. These conflicts (and therefore the resulting constraints) can be divided to three groups.
(1) internal: conflicts between a_i and a_j.
(2) external(i): conflicts between a_i and any other agent a_k (where $k \neq j$).
(3) external(j): conflicts between a_j and any other agent a_k (where $k \neq i$).

Since a_i and a_j are now going to be merged, internal conflicts are no longer relevant as a_i and a_j will be solved in a coupled manner by the low-level solver. Thus, we only consider external constraints.[2] For each external constraint (a_i, v, t) we add a meta constraint $(a_{i,j}, v, t)$. Similarly, for each external constraint (a_j, v, t) we add a meta constraint $(a_{i,j}, v, t)$.

To illustrate MA-CBS, consider again the example shown in Figure 1(i). Assume that we are using MA-CBS(0). In this case, at the root of the CT, once the conflict $(a_1, a_2, C, 2)$ is found, $should - merge()$ returns *true* and agents a_1 and a_2 are merged into a new meta-agent $a_{1,2}$. Next the low-level solver is invoked to solve the newly created meta-agent and a (conflict-free) optimal path for the two agents is found. If A* is used, a 2-agent A* search will be executed for this. The high-level node is now re-inserted into OPEN as its f-value increased from 8 to 9. Since it is the only node in OPEN, it will be expanded next. On the second expansion the search halts as no conflicts exists - there is only one meta-agent, which has no internal conflicts by definitions. Thus, solution from the root node is returned. By contrast, for MA-CBS(B) for $B > 0$, the root node will be split according the conflict as describe above.

In (Sharon *et al.* 2012) we provided a formal proof for the correctness of basic CBS. With a few adaptations, this proof generally holds for MA-CBS too, given that an agent can also be a *meta-agent*. We thus omit this proof here.

MA-CBS Generalizes ID

While MA-CBS(∞) is basic CBS, MA-CBS(0) is exactly the recently proposed technique for MAPF called Independence Detection (ID) (Standley 2010). Thus, MA-CBS, is a general framework which has these two previous algorithms as a special extreme cases.

ID is a technique used to identify groups of agents that can be solved independently. ID is a general framework which

---

[2]To identify the type of a given constraint implementers might choose to store a list of conflict affiliated with each high-level node.

runs as a base level and can use any possible MAPF solver on top of it. Two groups of agents are *independent* if there is an optimal solution for each group such that the two solutions do not conflict. The basic idea of ID is to divide the agents into *independent* groups. Initially each agent is placed in its own group. Optimal solutions are found for each group separately. Given a solution for each group, paths are checked to see if a conflict occurs between two (or more) groups. If so, all agents in the conflicting groups are unified into a new group. Whenever a new group of $k \geq 1$ agents is formed, this new k-agent problem is solved optimally by any MAPF optimal solver. This process is repeated until no conflicts between groups occur. Since the problem is exponential in k, Standley observed that solving the largest group dominates the running time of solving the entire problem, as all others involve smaller groups.

It is easy to see that the ID framework is identical to MA-CBS(0). In the root node, MA-CBS(0) solves each agent separately. Then, MA-CBS(0) expands this CT root node, finding a conflict between the solutions of the single agents (if one exists). The conflicting agents that have at least one conflict will be merged as the threshold parameter $B = 0$. The combined group will be solved using the low-level MAPF solver. Next the root will be re-inserted into OPEN and a validation occurs. Since $B = 0$ the branching option will never be chosen and conflicts are always solved by merging the conflicted (meta) agents. Thus, this variant will only have one CT node which is being re-inserted into OPEN, until no conflicts occur.

However, MA-CBS(B) can be significantly better than ID, when it solves agents that are only loosely coupled, by adding constraints to these agents separately. For example, in the case of a bottleneck (such as Figure 1(i)) where the individual solutions of two agents conflict, ID (=MA-CBS(0)) will merge these agents to a single group and solve it in a coupled manner. By contrast, MA-CBS(B) (with $B > 0$) can avoid this bottleneck by adding a single constraint to one of the agents. Clearly, using MA-CBS(B) for any value of $B \geq 0$ adds much more flexibility and may significantly outperform ID. This is clearly seen in the experimental results section described next.[3]

Experimental Results

In this section we study the behavior of MA-CBS empirically on three standard benchmark maps from the game *Dragon Age: Origins* (Sturtevant 2012). Each of the three maps, shown in Figure 3, represent a different topology. 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. Our main objective was to study the effect of the *conflict bound* parameter B on the performance of MA-CBS. Recall again that

MA-CBS(0) is equivalent to A*+ID (or any other MAPF solver used instead of A* for the low-level search), while MA-CBS(∞) is the basic CBS algorithm (Sharon *et al.* 2012). We have run experiments with MA-CBS(B) where $B = 0, 1, 5, 10, 100, 500$ and ∞.

Since MA-CBS is a framework that can use any A*-based solver for the low-level search, we experimented with two such solvers: A* and Enhanced Partial Expansion A* (EPEA*) (Felner *et al.* 2012).[4] Both solvers used the SIC heuristic (defined above). A* was chosen as a baseline, while EPEA* was chosen since it is currently the state-of-the-art A*-based MAPF solver. The different variants of the ICTS algorithm (Sharon *et al.* 2011a) cannot detect unsolvable problems, so ICTS cannot be used inside the MA-CBS framework as a low level solver without modifications which are beyond the scope of this paper.

For each of the maps we varied the number of agents k. We ran our algorithms on 100 random instances for each value of k. If an algorithm did not solve a given problem instance within five minutes it was halted. The numbers reported are an average over all instances solved by all the algorithms. For variants where an algorithm did not solve at least 70% of the cases we report a lower bound on all cases. Thus, some numbers have $>$ before them.

The table in Figure 2 shows runtime in ms for the experiments described above. The k column denotes the number of agents in the experiment. MA-CBS(x) is denoted only by B(x). For a given number of agents, the result of the best-performing algorithm is given in bold. The top frame is for the den520d map where A* was used for the low-level search while the rest of the frames report results when EPEA* was used. Each frame presents a different map. Similar trends were observed in the data not shown.

The results clearly show that MA-CBS with non-extreme values, i.e., with $B \neq 0$ and $B \neq \infty$, is able to solve most instances faster than the two extreme cases, i.e., MA-CBS(0) (A*(EPEA*)+ID) and MA-CBS(∞) (basic CBS). The new variants achieved up to an order of magnitude speed-up over MA-CBS(∞) (e.g. in den520d for 35 and 40 agents with EPEA* as the low-level solver) and up to a factor of 4 over MA-CBS(0) (e.g., in ost003d with 25 agents).

Next, consider the effect of increasing the number of agents k for the den520d map where EPEA* was used (second frame). Problems with fewer agents ($k < 25$) were solved more quickly using MA-CBS with large values of B. As the problems become denser ($k > 30$), MA-CBS with smaller B values is faster. In addition, the relative performance of basic CBS ($\equiv$ MA-CBS(∞)) and MA-CBS(500) with respect to the best variant degrades. This is explained as follows. In dense problem instances, where there are many agents relative to the map size, many conflicts occur. Recall

---

[3]Standley 2010 also presented an enhanced variant of ID. When a conflict is found between the solutions of two groups, then a replanning phase tries to find alternative plans that avoid conflicting. This significantly increases the chances of finding independent plans. This variant is a reminiscent of MA-CBS(1).

[4]EPEA* is a variant of A* that uses domain specific knowledge to generate only nodes with the same f-cost as the parent. n is re-inserted into the OPEN list, but with the f-cost of the next best child. This saves significant overhead caused by dealing with *surplus* nodes - nodes with f-value larger than that of the optimal cost. In EPEA* such nodes will not be generated and will not enter OPEN. In MAPF, the number of such nodes is exponential in the number of agents.

den520d with A* as a low-level solver						
k	A*	B(1)	B(5)	B(10)	B(100)	B(500)	CBS
5	0.223	273	218	220	**219**	222	**219**
10	1,099	1,458	553	552	549	552	**546**
15	1,182	1,620	1,838	1,810	1,829	1,703	**1,672**
20	4,792	4,375	1,996	2,011	2,020	1,857	**1,708**
25	7,633	14,749	**2,193**	2,255	2,320	2,888	3,046
30	> 62,717	> 60,214	8,082	8,055	8,107	8,013	**7,745**
35	> 65,947	> 51,815	13,670	**13,587**	15,981	28,274	> 45,954
40	> 81,487	>82,860	18,473	**18,399**	20,391	31,189	> 45,857

den520d with EPEA* as a low-level solver						
k	EPEA*	B(1)	B(5)	B(10)	B(100)	B(500)	CBS
5	899	190	**180**	181	**180**	**180**	256
10	1,633	1,782	470	**467**	469	469	632
15	1,621	2,241	1,708	**1,702**	1,713	1,738	1,807
20	3,393	3,725	1,527	**1,515**	1,553	1,555	1,867
25	7,675	8,327	1,701	**1,620**	1,731	2,071	3,264
30	12,574	13,308	3,955	**3,773**	5,276	16,191	>38,707
35	15,736	12,655	**4,974**	4,993	7,199	18,998	> 50,050
40	14,635	15,452	**4,860**	4,971	7,686	20,860	>50,891

ost003d with EPEA* as a low-level solver						
k	EPEA*	B(1)	B(5)	B(10)	B(100)	B(500)	CBS
5	187	231	**168**	**168**	169	169	222
10	1,718	1,983	764	**753**	757	757	935
15	4,888	4,593	1,597	1,592	**1,568**	1,570	1,909
20	10,463	13,426	3,701	3,654	3,623	**3,598**	4,119
25	> 60,140	>58,902	>28,881	**15,109**	18,159	35,536	> 73,860
30	>84,473	> 80,248	> 30,781	**25,860**	27,525	46,328	>92,209
35	>90,703	>81,633	>39,660	**21,466**	28,241	47,544	> 95,262

brc202d with EPEA* as a low-level solver						
k	EPEA*	B(1)	B(5)	B(10)	B(100)	B(500)	CBS
5	1,834	2,351	1,286	1,276	1,268	**1,267**	1,664
10	6,034	8,059	4,580	4,530	**4,498**	4,508	5,495
15	12,354	15,389	6,903	6,871	6,820	**6,793**	8,685
20	> 70,003	>73,511	35,095	21,729	**19,846**	31,229	>43,625

Figure 2: Runtime of the MA-CBS experiments.

den520d

ost003d

brc202d

Figure 3: Success rate of the MA-CBS experiments with EPEA* as the low-level solver

that basic CBS and MA-CBS are exponential in the number of conflicts encountered. Thus, increasing the number of agents makes the problem denser and, as a result, the relative performance of MA-CBS with large values of B (which behaves closer to basic CBS) degrades when compared to variants with small values of B. In separate experiments in the extreme scenario where $k = |V|-1$, like the sliding-tile-puzzle, we observed that MA-CBS(0) ($\equiv$ coupled solver) performs best.

Now, consider the results in the table where A* was used as a low-level solver (top frame). Here, we see the same general trend as observed in the results for EPEA*. However, we observe that the best-performing value of B was larger than that of MA-CBS with EPEA* (second frame). For example, in the den520d map with 30 agents, MA-CBS(5) with A* as the low-level solver did not obtained a significant speedup over CBS. For EPEA* as the low-level solver MA-CBS(5) obtained an order of magnitude speedup over CBS. The same tendency can also be observed in the other maps. The reason is that for a relatively weak MAPF solver,
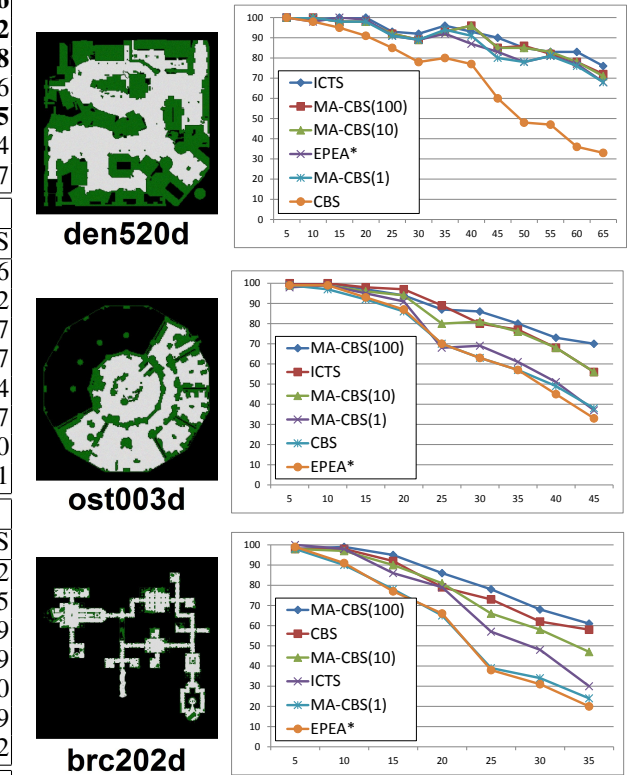
such as A*, solving a large group of agents is very inefficient. Thus, we would like to avoid merging agents and run in a more decoupled manner. For these cases a higher B is preferred. On the other hand fast MAPF solvers, such as EPEA*, would perform better with a lower value of B.

Figure 3 shows the *success rate*, i.e., the number of instances solved before the timeout, for MA-CBS with $B = 0, 1, 10, 100$ and ∞. Only results for EPEA* as the low-level is presented as A* results were similar. Additionally, for comparison we also report the success rate of the best ICTS variant (Sharon *et al.* 2011b). Note that curves are ordered according to their performance in the given map. Thus, similar algorithm might have a different legend in the different maps.

As can be seen, in all the experiments MA-CBS with intermediate values, i.e., $0 < B < \infty$, is able to solve more instance than both extreme cases, i.e., EPEA* ($\equiv MA-CBS(0)$) and basic CBS ($\equiv MA-CBS(\infty)$). Additionally, MA-CBS with intermediate values also outperforms the ICTS solver in the brc202d and ost003d maps, and

performed very similar to ICTS for the den520d map. This supports the understanding from (Sharon *et al.* 2011b) that ICTS is especially effective for maps with large open spaces such as den520d but not as good for domains with corridors and deadends such as brc202d.

Consider the performance of MA-CBS variants with $B < \infty$ in comparison with the basic CBS ($B = \infty$). Basic CBS performs very poor for den520d (top), somewhat poor for ost003d (middle) but rather well for brc202d (bottom). This is because in maps with no bottlenecks and large open spaces, such as den520d, CBS will be inefficient, since many conflicts will occur. This phenomenon is explained in the pathological example of CBS given in Figure 1(iii). Thus, in den520d the benefit of merging agents is high, as we avoid many conflicts. By contrast, for maps without large open spaces and many bottlenecks, such as brc202d, CBS encounters few conflicts, and thus merging agents results in only a small reduction in conflicts. Indeed, as the results show, for brc202d the basic CBS (MA-CBS(∞)) achieves almost the same performance as setting lower values of B.

In problems with higher conflict rates it is, in general, more helpful to merge agents, and hence lower values of B perform better. For example, for den520d (top) MA-CBS(10) obtained the highest success rates. By contrast, for ost003d and brc202d MA-CBS(100) obtained the highest success rates.

Summarizing the experimental results, we observed the following trends:

- MA-CBS with non-trivial B values ($0 < B < \infty$) outperforms previous algorithms: A*, EPEA* and CBS.

- **Density.** In dense maps with many agents, low values of B are more efficient.

- **Topology.** In maps with large open spaces and few bottlenecks, low values of B are more efficient.

- **Low-level solver.** If a slow MAPF solver is used for the low-level search, high values of B are preferred.

Discussion and Future Work

This paper introduces the MA-CBS algorithm, a generalization of the CBS algorithm (Sharon *et al.* 2012) for solving MAPF problems optimally. MA-CBS serves as a bridge between CBS and completely coupled solvers, such as A*, A*+OD (Standley 2010) and EPEA* (Felner *et al.* 2012). It starts as a regular CBS solver, where all the low-level search is performed by single-agent searches. If MA-CBS identifies that a pair of agents often conflict, it groups them together. The low-level solver treats this group as one single composite agent, and finds solutions for that group using a given MAPF solver (e.g., A*). As a result, MA-CBS is flexible and can enjoy the complementary benefits of both CBS and traditional coupled solvers by setting the correct value for B in the range between the two extremes. In cases where only a few conflicts occur, MA-CBS can act like CBS, while if conflicts are common, MA-CBS can converge to a completely coupled solver. Experimental results showed that MA-CBS with non-extreme values of B (i.e., neither

$B = 0$ nor $B = \infty$) outperforms both CBS and other state-of-the-art MAPF algorithms.

MA-CBS is in fact a general framework that can use any MPAF solver as a low-level solver. Furthermore, MA-CBS can be viewed as a generalization of the Independence Detection (ID) framework introduced by Standley (2010).

A number of ongoing directions for MA-CBS include: (1) devising more intelligent merge policies, (2) dynamically changing B based on the exact topology currently seen. Finally, (3) the approach of mixing constraints and search is related to recent work on the theoretical properties of A* and SAT algorithms (Rintanen 2011) and is an important connection that needs more study.

Acknowledgments

References

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

Felner, A.; Goldenberg, M.; Sturtevant, N.; Stern, R.; Sharon, G.; Beja, T.; Holte, R.; and Schaeffer, J. 2012. Partial-expansion A* with selective node generation. In *AAAI*.

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

Khorshid, M. M.; Holte, R. C.; and Sturtevant, N. R. 2011. A polynomial-time algorithm for non-optimal multi-agent pathfinding. In *SOCS*.

Luna, R., and Bekris, K. E. 2011. Push and swap: Fast cooperative path-finding with completeness guarantees. In *IJCAI*, 294–300.

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

Rintanen, J. 2011. Planning with SAT, admissible heuristics and A*. In *IJCAI*, 2015–2020.

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

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

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011a. The increasing cost tree search for optimal multi-agent pathfinding. In *IJCAI*, 662–667.

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2011b. Pruning techniques for the increasing cost tree search for optimal multi-agent pathfinding. In *SOCS*.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Conflict-based search for optimal multi-agent path finding. In *AAAI*.

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

Standley, T., and Korf, R. 2011. Complete algorithms for cooperative pathnding problems. In *IJCAI*, 668–673.

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

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games*.

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