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

In the *multi agent path finding* problem (MAPF) paths should be found for several agents, each with a different start and goal position such that agents do not collide. Previous optimal solvers applied global A*-based searches. We present a new search algorithm called Conflict Based Search (CBS). 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. In many cases this reformulation enables CBS to examine **fewer** states than A* while still maintaining optimality. We analyze CBS and show its benefits and drawbacks. Experimental results on various problems shows a speedup of up to a full order of magnitude over previous approaches.

Introduction

In the *multi-agent path finding* (MAPF) problem, we are given a graph, $G(V, E)$, and a set of k agents labeled $a_1 \dots a_k$. 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 a set of actions for each agent, 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) while minimizing a cumulative cost function. The common cost function is the sum over all agents of the number of time steps required to reach the goal location. Therefore, both *move* and *wait* actions cost 1.0.

MAPF has practical applications in robotics, video games, vehicle routing etc. (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 known to be NP-complete (Ratner & Warrnuth 1986).

We assume a *centralized computing* setting with a single CPU which needs to solve a MAPF problem. This 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 a *centralized* manner. In the *decoupled approach* paths are planned for each agent separately. A prominent example is HCA* (Silver 2005). Given an agent ordering, a path is found for agent a_i and then written (*reserved*) into a global *reservation table*. The search for successive agents must avoid locations and time points that were reserved by previous agents. A similar approach was used for guiding cars that need to cross traffic junctions (Dresner & Stone 2008). Other *decoupled* approaches establish flow restrictions similar to traffic laws, directing agents at a given location to move only in a designated direction (Wang & Botea 2008; Jansen & Sturtevant 2008). *Decoupled* approaches run relatively fast, but optimality and even completeness are not always guaranteed. New complete *decoupled* algorithms were recently introduced for trees (Khorshid, Holte, & Sturtevant 2011) and for general graphs (Luna & Bekris 2011).

As we aim at solving the MAPF problem optimally, the focus of this paper is on the *coupled approach*. In this approach MAPF is formalized as a global, single-agent search problem. This formulation could be solved by an A*-based algorithm that searches a state space that grows exponentially with the number of agents. Coupled (global) searches usually return the optimal solution at significant computational expense. Previous coupled approaches dealt with the large search space in different ways (Ryan 2008; 2010; Standley 2010; Standley & Korf 2011; Sharon *et al.* 2011a).

Sharon et al. (2011a; 2011b) 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.

We present a new algorithm called Conflict Based Search (CBS). CBS is a continuum of coupled and decoupled approaches. CBS guarantees optimal solutions, like most *coupled* approaches, but the pathfinding that CBS performs are strictly single-agent searches, similar to the decoupled approaches. CBS is a two-level algorithm where the high level search is performed in a *constraint tree* (CT) whose nodes include constraints on time and location for a single agent. At each node in the constraint tree a low-level search is performed to find new paths for all agents under the constraints given by the high-level node. Unlike A*-based searches where the search tree is exponential in the number of agents, the high-level search tree of CBS is exponential in the number of conflicts encountered during the solving process.

We analyze CBS and study circumstances where CBS is

weak and when it is strong compared to the A*-based approaches. In many cases, CBS outperforms other optimal solvers by up to a full order of magnitude.

Previous Optimal Solvers

The straightforward direction for optimal MAPF solvers is to formalize the problem as a global search space and solve it with A*. The *states* are the different ways to place the k agents into the V vertices without conflicts. At the start (goal) state agent a_i is located at vertex s_i (g_i). *Operators* between states are all the non-conflicting actions (including *wait*) that all agents have. Let b_{base} be the branching factor for a single agent. The global branching factor is $b = O((b_{base})^k)$. All $(b_{base})^k$ combinations of actions should be considered and only those with no conflicts are *legal* neighbors. Any A*-based algorithm can then be used to solve the problem.

A common heuristic function for MAPF solvers (Standley 2010; Sharon *et al.* 2011a) is the *sum of individual costs* heuristic (SIC). For each agent a_i we assume that no other agents exist and calculate its optimal individual path cost. We then return the sum of these costs. For grids with no obstacles this is exactly the Manhattan distance. The costs that make up the SIC heuristics can be easily precalculated and stored in k lookup tables, each of size $|V|$.

Standley's enhancements

Recently, three methods that substantially improve the basic A* setting were introduced by (Standley 2010). CBS uses them so we describe each of them in turn.

Independence detection (ID): ID is a general framework which 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.

Conflict voidance table (CAT): Within the ID framework, a MAPF solver for a given group of agents is invoked. Since many optimal solutions may exist for this group, Standley suggested using a dynamic lookup table called the *conflict avoidance table* (CAT). The CAT stores the location and time of every agent in every group. Then, when an MAPF solver is applied for a given group, ties between nodes with the same f-value are broken in favor of the node that has fewer entries in the CAT.

Operator decomposition (OD): While the former two ideas are general and can be used by any MAPF solver, OD is specific for an A*-based solver. OD reduces the branching factor by introducing *intermediate states* between the regular states. *Intermediate states* are generated by applying an operator to a single agent only. This helps in pruning misleading directions at the intermediate stage without considering moves of all of the agents (in a regular state).

The Increasing Cost Tree Search (ICTS)

(Sharon *et al.* 2011a) introduced the *increasing cost tree search* (ICTS). ICTS is 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 levels. At the **high-level** it searches a tree called the *increasing cost tree* (ICT). Each node in this tree is associated with a vector of costs, one per individual agent. At the **low-level** it performs a goal test on the high-level ICT nodes. Given the cost-per-agent vector it searches over combinations of single-agents paths, each with its cost from the vector, to either find a non-conflicting solution with these costs or to verify that such a solution does not exist.

(Sharon *et al.* 2011a) defined Δ as the difference between the optimal solution and the SIC heuristic of the initial state. ICTS was shown to outperform the A* approaches in cases where $\Delta < k$ (the number of agents) and vise versa. (Sharon *et al.* 2011b) suggested a number of pruning techniques that allow the high-level to recognize that a given node is not a goal without invoking the low-level. The best of these techniques is called ICTS+3E. This enhancement uses information about small groups of up to 3 agents and their internal conflicts. This is equivalent to an A* search with a heuristic that is based on similar groups of up to 3 agents, e.g., in the form of *additive PDBs* (Felner, Korf, & Hanan 2004). However, efficiently building such heuristics for general MAPF algorithms is an open question.

The Conflict Based Search Algorithm (CBS)

The state space of MAPF is exponential in k the number of agents. By contrast, in a single-agent pathfinding problem, $(k = 1)$, and the state space is only linear in the graph size. CBS solves the MAPF problem by decomposing it into a large number of single-agent pathfinding problems. Each problem is relatively simple to solve, but there may be an exponential number of such single-agent problems.

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. 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 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 conflicts are found and constraints are added. The low-level updates the agents paths to be consistent with the new constraints. We describe each part of this process in more detail below.

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 **(3) The total cost (**$N.cost$**)** 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 performs a best-first search on the CT where nodes are ordered by their costs. Ties are broken by using a *conflict avoidance table* (CAT) as described above.

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 and 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).

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.

Algorithm 1: high-level of CBS

```
   Input: MAPF instance
1  R.constraints = ∅
2  R.solution = find individual paths using 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     foreach agent a_i in C do
12         A ← new node
13         A.constriants ← P.constriants + (a_i, s, t)
14         A.solution ← P.solution.
15         Update A.solution by invoking low-level(a_i)
16         A.cost = SIC(A.solution)
17         Insert A to OPEN
```

CBS Example

The high-level of CBS is shown in Algorithm 1. It 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. The low-level now returns an optimal solution for each agent (line 2 of Algorithm 1) , $< S_1, A_1, C, G_1 >$ 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 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 11). 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 15) to find an optimal path that also satisfies the new constraint. For the left child, 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 for 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 17). 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.

Conflicts of $k > 2$ agents. It may be the case that while performing the validation (Line 7) between the dif-

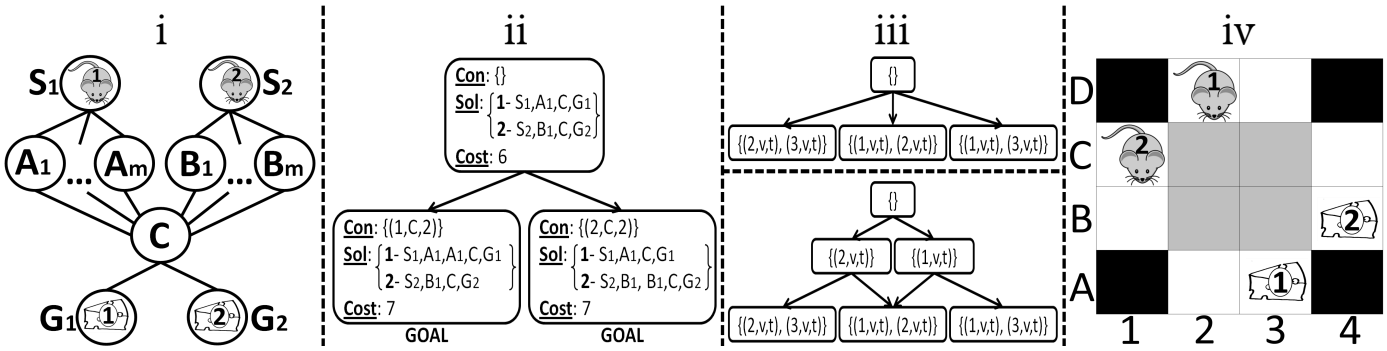

Figure 1: (i) MAPF example (ii) CT (iii) A k-way (top) and binary (bottom) CT (iv) A case where A* outperforms CBS.

ferent 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.

This is illustrated in Figure 1(iii). The top tree represent a k-way branching CT for a problem that contains a 3-agent conflict at vertex v at time t. The bottom tree presents a binary CT for the same problem. As can be seen the size of the deepest layer in both trees is similar. The complexity of the two approaches is the same as they both will end up with k nodes each with $k-1$ new constraints. For simplicity of description we chose the second option.

Low-level: Find Paths 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 satisfy 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. 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.

A low-level CAT was used for each CT node N. It is initialized by the current paths of node N. 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 results in a higher quality solution (less conflicting agents) for each high-level node.

Theoretical Analysis: Optimality of CBS

We now prove that CBS will returns an optimal solution if one exists. First, we provide several supporting claims.

Definition 1 *For given node N in a constraint tree, let $CV(N)$ be the set of* all *solutions that are: (1)* consistent *with the set of constraints of N and (2) are also* valid *(i.e, without conflicts).*

If N is not a goal node, then the solution at N will not be part of $CV(N)$ because it is not valid.

Definition 2 *For any solution $p \in CV(N)$ we say that node N* permits *the solution p.*

The root of the CT, for example, has an empty set of constraints. Any valid solution satisfies the empty set of constraints. Thus the root node *permits* all valid solutions.

The cost of a solution in $CV(N)$ is the sum of the costs of the individual agents. Let $minCost(CV(N))$ be the minimum cost over all solutions in $CV(N)$.

Lemma 1 *The cost of a node N in the CT is a lower bound on $minCost(CV(N))$.*

Proof: $N.cost$ is the optimal cost of a set of paths that satisfy the constraints of N. This set of paths is not necessarily a valid solution. Thus, $N.cost$ is a lower bound on the cost of any set of paths that make a valid solution for N as no single agent in any solution can achieve its goal faster. □

Lemma 2 *Let p be a valid solution. At all time steps there exists a CT node N in* OPEN *that* permits *p.*

Proof: By induction on the expansion cycle: For the base case OPEN only contains the root node, which has no constraints. Consequently, the root node *permits* all valid solutions and also p. Now, assume this is true for the first i expansion cycles. In cycle $i+1$, assume that node N, which *permits* p, is expanded and its children N'_1, N'_2 are generated. Any valid solution in $VS(N)$ must be either in $VS(N'_1)$ or in $VS(N'_2)$, as any valid solution must satisfy at least one of the new constraints. □

Thus, at all times at least one CT node in OPEN *permits* the optimal solution (as a special case of Lemma 2).

Theorem 1 *CBS returns the optimal solution.*

Proof: Consider the expansion cycle when a goal node G is chosen for expansion by the high-level. At that point all valid solutions are *permitted* by at least one node from OPEN (lemma 2). Let p be a valid solution (with cost $c(p)$) and let $N(p)$ be the node that *permits* p in OPEN, Let $c(N)$ be the cost of node N. $c(N(p)) \leq c(p)$ (Lemma 1). Since

	#Generated nodes				Run-Time (ms)				
k	A*	A*+OD	CBS(hl)	CBS(ll)	A*	A*+OD	ICTS	ICTS3	CBS
3	414	**82**	7	229	14	3	1	1	2
4	2,843	**243**	31	1,135	380	10	2	1	13
5	19,061	**556**	42	1,142	9,522	50	5	5	13
6	64,734	**677**	42	1,465	47,783	90	**9**	10	16
7	NA	**4,451**	287	13,804	NA	1,122	92	**44**	186
8	NA	**8,035**	308	13,820	NA	1,756	271	**128**	211
9	NA	30,707	740	**25,722**	NA	7,058	2,926	921	**550**
10	NA	54,502	1,095	**34,191**	NA	21,334	10,943	2,335	**1,049**
11	NA	NA	2,150	**69,363**	NA	NA	38,776	5,243	**2,403**
12	NA	NA	11,694	**395,777**	NA	NA	NA	25,537	**15,272**
13	NA	NA	22,995	**838,149**	NA	NA	NA	45,994	**36,210**

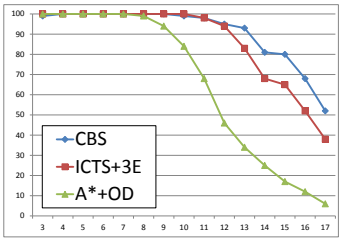

Figure 2: Nodes generated and running time on 8x8 grid (left). Success rate (right)

G is a goal node $c(G)$ is a cost of a valid solution. Since the high-level search explores solution costs in a best-first manner we get that $c(g) \leq c(N(p)) \leq c(p)$ $\square$

Comparison with other algorithms

Let C^* be the cost of the optimal solution, let χ be the set of nodes with $f < C^*$ and let $X = |\chi|$. It is well known that A* must expand all nodes in χ in order to guarantee optimality. Furthermore, A* is known to be "optimally effective", which means that A* expands the minimal number of nodes necessary to ensure an optimal solution (Dechter & Pearl 1985), for a given h function.

(Sharon *et al.* 2011a) showed that A* may generate up to $X \times (b_{base})^k$ nodes while ICTS might generate up to $X \times k^{\Delta}$ nodes. As ICTS also expand all nodes in χ, we currently limit the discussion to A*. But, what is the size of set χ? Given a heuristic h and the solution depth d some prediction algorithms were developed for IDA* but they are based on sampling (Korf, Reid, & Edelkamp 2001; Zahavi *et al.* 2010; Lelis, Zilles, & Holte 2011). The simplest upper bound for X is the entire state space, although this can be a gross overestimate of X. In the case of MAPF this is $\binom{|V|}{k} = O(|V|^k)$.

Similar reasoning apply for CBS. Let Υ be the set of nodes with $f < C^*$ in the CT tree and let $Y = |\Upsilon|$. As a best-first search guided by f, CBS must expand all the nodes in Υ. Again, prediction formulas might be developed but we suffice by giving an upper bound. As the branching factor of the CT is 2, 2^d nodes must be expanded in the worst case where d is depth of the solution.

At each node of the CT exactly one constraint is added. In the worst case an agent will be constrained to avoid every vertex except one at every time step in the solution. The total number of time steps summed over all agents is C^*. Thus, an upper bound on the number of CT nodes that CBS must expand is $2^{|V| \times C^*}$. For each of these nodes the low-level is invoked and expands at most $|V|$ (single-agent) states for each time step. The total number of time steps summed over all agents is C^*. Let $\overline{Y}$ be the number of states expanded in the underlying graph (low-level nodes) .$\overline{Y}$ is bounded by $2^{|V| \times C^*} \times |V| \times C^*$. Again, in practice Y and $\overline{Y}$ can be significantly smaller.

A* searches a single search tree and expands X nodes. CBS performs a high-level search in the CT and then a low-level search in the underlying graph and expands a total of $\overline{Y}$ low-level states. Thus, if $\overline{Y} << X$ then CBS will outperform A* and vise versa. We now show special cases where $\overline{Y} << X$ (bottleneck) and where $X << \overline{Y}$ (open space). This seems to be typical for MAPF, where the topology of the domain greatly influences the behavior of algorithms. The main difference between the behavior of the algorithms in the different cases is the amount of work that needs to be done before the f-cost of the searches increases. The examples below demonstrate this for simple cases but they can be easily generalized.

A case where CBS outperforms A* (bottlenecks) Figure 1(i) demonstrates a case where $\overline{Y} << X$, i.e., a case where CBS expands fewer nodes than A*. As detailed above, CBS generates three CT nodes. At the root, the low-level is invoked for the two agents. The low-level search finds an optimal path for each of the two agents (each of length 3), and expands a total of 8 low-level nodes for the CT root. Now, a conflict is found at C. Two new CT children nodes are generated. 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 $f = 4$ and 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 were expanded. Similar $m + 3$ states are expanded for the right child. Adding all these to the 8 states expanded at the root we get that a total of $\overline{Y} = 2m + 14$ low-level nodes were expanded.

Now, consider A* which is running in a 2-agent state space. The root (S_1, S_2) has $f = 6$. It generates m^2 nodes, all in the form of (A_i, B_j) for $(1 \leq i, j \leq m)$. All these nodes are expanded with $f = 6$. Now, node (A_1, C) with $f = 7$ is expanded (agent a_1 waits at A_1), Then nodes (C, G_2) and (G_1, G_2) are expanded and the solution is returned. So, in total A* expanded $X = m^2 + 3$ nodes. For $m \geq 5$ this is larger than $2m + 14$ and consequently, CBS will expand fewer nodes. A* must expand the cartesian product of single agent paths with $f = 3$. By contrast, CBS only tried two such paths to realize that no solution of cost 6 is valid. Furthermore, the nodes counted for A*

	den520d		ost003d		brc202d	
k	A*	CBS	A*	CBS	A*	CBS
5	2,210	**1,160**	1,057	**1,026**	7,674	**7,065**
10	4,992	**3,092**	6,360	**4,372**	22,149	**19,447**
15	5,599	**4,204**	14,822	**9,841**	41,789	**35,630**
20	8,630	**7,247**	20,431	**13,196**	61,587	**56,638**
25	**10,266**	14,373	NA	NA	NA	NA

Table 1: Nodes expanded on DAO problems.

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

A case where A* outperforms CBS (open space) Figure 1(iv) presents a case where $\overline{Y} >> X$ and A* will outperform CBS. For this problem the initial SIC heuristic is 8. 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, $C^* = 9$ as one agent must wait at least one step to avoid collision. For this problem A* will expand 5 nodes with $f = 8$: $\{(D2, C1), (D3, C2), (D3, B1), (C2, B1), (C3, B2)\}$ and 3 nodes with $f = 9$ $\{(B3, B2), (A3, B3), (A3, B4)\}$ 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 single-agent states to a total of $16 \times 4 = 64$ low-level nodes. Next, we consider the goal CT node with $f = 9$. It will expand 7 new states. Thus, a total of $\overline{Y} = 71$ states are expanded for CBS.

While it is hard to predict the performance of these algorithms in actual domains, the above observations can give some guidance. If there are more bottlenecks, CBS will have advantage over the A*-based approaches. If there are more open spaces, A* will have advantage over CBS. Next we show experimental results supporting both cases.

Experimental results

We experimented with A*, A* enhanced by OD (denoted A*+OD), ICTS, ICTS+3E and CBS. All algorithms, except ICTS+3E are based on the SIC heuristic. ICTS+3E uses more advanced pruning that could potentially apply to CBS and A* as advanced heuristics in the future. Despite this, CBS without this advanced heuristic still outperforms ICTS+3E in many scenarios. In all our experiments we also tried CBS without the *CAT-based tie braking*. It was two times slower than CBS with the *CAT-based tie breaking*. Only the stronger version is shown.

8x8 4-connected grid We begin with an 8x8 4-connected grid where the number of agents increases from 3 to 13. We followed (Sharon *et al.* 2011a) and used the ID framework only in a preprocessing level. We report results for agents which were all in the same independent group found by ID. Figure 2(left) presents the number of nodes generated and the run time averaged over 100 instances. For the case of CBS both the high-level nodes and low-level states

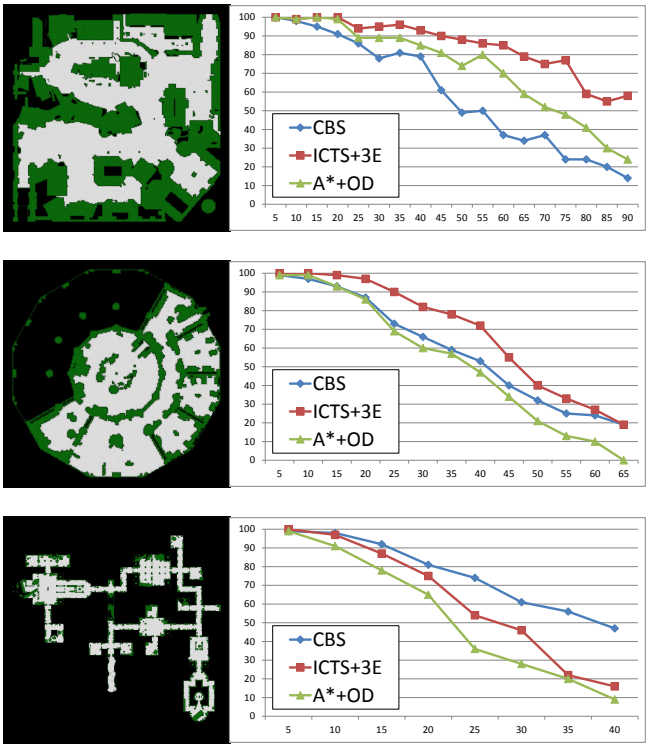

Figure 3: DAO maps den520d (top), ost003d (middle), brc202d (bottom) and their success rate

are reported. We set a time limit of 5 minutes. If an algorithm could not solve an instance within the time limit it was halted and *fail* was returned ("NA" in the table). Figure 2(right) shows the success rate, i.e., the percentage of instances that could be solved under 5 minutes by the different algorithms when the number of agents increase. Similar to (Sharon *et al.* 2011a) we do not report the number of nodes for the ICTS variants. This is because this algorithm is not based solely on search. Clearly, CBS significantly outperforms A* and A*+OD by up to a full order of magnitude. Note that although CBS sometimes generates more nodes than A*+OD, it is still faster in many cases due to the fact that the constant time per node of CBS (single-agent state) is smaller than that of A*+OD (multiple agents). CBS is faster than ICTS and ICTS+3E for ≥ 8 and ≥ 9 agents respectively.

DAO maps We also experimented on 3 benchmark maps from the game *Dragon Age: Origins* (Sturtevant 2012). Here we aimed to solve problems with as many agents and thus the ID framework was activated as an integral part of the problem solving. In this experiment we show results only for the three strongest algorithms: A*+OD, ICTS+3E and CBS. Table 1 shows the number of states expanded by CBS and by A* for the three DAO maps presented in Figure 3 (left). Clearly, CBS expands less states than A* in all these cases. Additionally, recall that the cost of a CBS node expansion is less than A*. The one exception is the den520d map for 25

agents.

Figure 3(right) shows the success rates given the number of agents for the three maps Here the results are mixed and there is no global winner. One can clearly see that ICTS+3E is always better than A*+OD. However, the performance of CBS on these maps supports our theoretical claims that CBS is very effective when dealing with corridors and bottlenecks but rather inefficient in open spaces. For den520d(top) there are no bottlenecks but there are large open spaces; CBS was third. For ost003d(middle) there are few bottlenecks and small open spaces; CBS was intermediate. Finally, for brc202b(bottom) there are many narrow corridors and bottlenecks but very few open spaces, thus CBS was best. Note that while both den520 and ost003 have open spaces they differ in the amount of bottlenecks.

Discussion and future work

This paper introduces the CBS algorithm for solving MAPF problems optimally. CBS is unique in that all low-level searches are performed as single-agent searches. The performance of CBS depends on the structure of the problem. We have demonstrated cases with bottlenecks (Figure 1(i)) where CBS performs well, and open spaces (Figure 1(iv)) where CBS performs poorly. Experimental results in testbed map problems support our theoretical claims. These domains have different rates of open spaces and bottlenecks. CBS outperforms other algorithms in cases where corridors and boutlnecks are more dominant.

Recently a number of optimal MAPF solvers have been introduced, each with pros and cons, but there is no universal winner. We have touched the surface here; further research is needed on the characteristics of MAPF and how they influence the behavior of different algorithms.

A number of ongoing directions for CBS include: (1) Adapting the meta-agent ideas from ICTS+3E. (2) Performing independence detection for CT nodes. (3) Finding admissible heuristics for the CT. (4) 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. (5) The relation to CSP solvers is still not deeply known.

Acknowledgements

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

References

Dechter, R., and Pearl, J. 1985. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM* 32(3):505–536.

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

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

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*.

Korf, R. E.; Reid, M.; and Edelkamp, S. 2001. Time complexity of iterative-deepening-A*. *Artificial Intelligence* 129(1–2):199–218.

Lelis, L.; Zilles, S.; and Holte, R. C. 2011. Improved prediction of IDA*s performance via ϵ-truncation. 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 $\times$ 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*.

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.

Zahavi, U.; Felner, A.; Burch, N.; and Holte, R. C. 2010. Predicting the performance of IDA* (with BPMX) with conditional distributions. *Journal of Artificial Intelligence Research* 37:41–83.