Don't Split, Try To Work It Out: Bypassing Conflicts in Multi-Agent Pathfinding

Eli Boyarski
CS Department
Bar-Ilan University
Israel
eli.boyarski@gmail.com

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

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

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

Abstract

Conflict-Based Search (CBS) is a recently introduced algorithm for Multi-Agent Path Finding (MAPF) whose runtime is exponential in the number of conflicts found between the agents' paths. We present an improved version of CBS that bypasses conflicts thereby reducing the CBS search tree. Experimental results show that this improvement reduces the runtime by an order of magnitude in many cases.

Introduction and Overview

A *Multi-Agent Path Finding* (MAPF) problem is defined by a graph, $G = (V, E)$ and a set of k agents labeled $a_1 \dots a_k$, where 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 an adjacent location or *wait* in its current location. The task is to plan a sequence of move/wait actions for each agent a_i, moving it from s_i to g_i such that agents do not *conflict*, i.e., occupy the same location at the same time. MAPF has practical applications in video games, traffic control, robotics etc. (See (Sharon et al. 2013) for a survey).

In this paper we focus on solving MAPF problems *optimally*, i.e., where the cost of the resulting plan is minimal. There is a range of algorithms that optimally solve different variants of MAPF using various search techniques (Standley 2010; Wagner and Choset 2011; Sharon et al. 2013) or by compiling it to other known NP-complete problems (Surynek 2012; Yu and LaValle 2013; Erdem et al. 2013). Each of these solvers has pros and cons. There is no universal winner. Which algorithm performs best under what circumstances is an open research question.

Conflict-Based Search (CBS) (Sharon et al. 2012a), is an optimal MAPF solver shown to be very effective in many domains. It is a two-level algorithm. The low-level finds optimal paths for the individual agents. If the paths include conflicts, the high level, via a *split* action (described below), imposes constraints on the conflicting agents to avoid these conflicts. CBS is exponential in the number of conflicts seen.

This paper introduces an improved version of CBS. When a conflict is found, we first attempt to bypass the conflict and avoid the need to perform a split and add new constraints. If no bypass is found we resort to the drastic split action of adding explicit constraints to avoid the conflict. We provide a number of variants that search for bypasses differing in their search effort. Experimental results show speedups over basic CBS by more than an order of magnitude.

The Conflict Based Search Algorithm (CBS)

A sequence of individual agent move/wait actions leading an agent from s_i to g_i is referred to as a *path*, and the term *solution* refers to a set of k paths, one for each agent. A *conflict* between two paths is a tuple $\langle a_i, a_j, v, t \rangle$ where agent a_i and agent a_j are planned to occupy vertex v at time point t. A solution is *valid* if it is conflict-free. The *cost* of a path is the number of actions in it (including wait), and the cost of a solution is the sum of the costs of its constituent paths.

In CBS, agents are associated with constraints. A *constraint* for agent a_i is a tuple $\langle a_i, v, t \rangle$ where agent a_i is prohibited from occupying vertex v at time step t. A *consistent path* for agent a_i is a path that satisfies *all* of a_i's constraints, and a *consistent solution* is a solution composed of only consistent paths. Note that a consistent solution can be *invalid* if despite the fact that the paths are consistent with the individual agent constraints, they still have inter-agent conflicts.

The high-level of CBS searches the *constraint tree* (CT). The CT is a binary tree, in which each node N contains: **(1:)** A set of constraints imposed on the agents ($N.constraints$), **(2:)** A single solution ($N.solution$) consistent with these constraints. **(3:)** The cost of $N.solution$ ($N.cost$).

The root of the CT contains an empty set of constraints. A successor of a node in the CT inherits the constraints of the parent and adds a single new constraint for a single agent. $N.solution$ is found by the low-level search described below. A CT node N is a goal node when $N.solution$ is valid, i.e., the set of paths for all agents have no conflicts. The high-level of CBS performs a best-first search on the CT where nodes are ordered by their costs ($N.cost$).

Processing a node in the CT: Given a CT node N, the low-level search is invoked for individual agents to return an optimal path that is consistent with their individual constraints in N. Any optimal single-agent path-finding algorithm can be used by the low level of CBS. We used A* with the true shortest distance heuristic (ignoring constraints). Once a consistent path has been found (by the low level) for each agent, these paths are *validated* with respect to the other agents by simulating the movement of the agents along

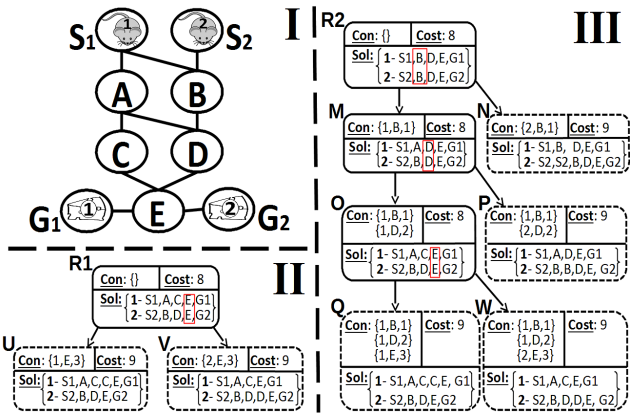

Figure 1: (I) MAPF example (II) CT (III) Alternative CT

their planned paths ($N.solution$). If all agents reach their goal without any conflict N is declared as the goal node, and $N.solution$ is returned. If, however, while performing the validation, a conflict is found for two (or more) agents the validation halts and the node is declared as non-goal.

Resolving a conflict: the *split* action Given a non-goal CT node, N, whose solution, $N.solution$, includes a *conflict*, $\langle a_i, a_j, v, t\rangle$, 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 satisfied. Consequently, CBS *splits* N and generates two new CT nodes as children of N, each adding one of these constraints to the previous set of constraints, $N.constraints$. Note that for each (non-root) CT node the low-level search is activated for one agent only – the agent for which the new constraint was added.

Algorithm 1: High-level of CBS

```
1  Main(MAPF problem instance)
2      R.constraints ← ∅
3      R.solution ← find individual paths using low level
4      R.cost ← SIC(R.solution)
5      insert R to OPEN
6      while OPEN not empty do
7          N ← best node from OPEN // lowest solution cost
8          Validate the paths in N until a conflict occurs.
9          if N has no conflict then
10             return N.solution // N is goal
11         C ← first conflict (a_i, a_j, v, t) in N
12         if Find-bypass(N,C) then
13             Continue //Optional
14         foreach agent a_i in C do
15             A ← generate child(N, (a_i, s, t))
16             Insert A to OPEN
17 Generate Child(Node N, Constraint C = (a_i, s, t))
18     A.constraints ← N.constraints + (a_i, s, t)
19     A.solution ← N.solution
20     Update A.solution by invoking low level(a_i)
21     A.cost ← SIC(A.solution)
22     return A
```

CBS Example: Pseudo-code for CBS is shown in Algorithm 1. 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 ($R1$) contains an empty set of constraints and the low-level search returns the following individual optimal paths: $P_1 = \langle S_1, A, C, E, G_1\rangle$ for agent a_1 and $P_2 = \langle S_2, B, D, E, G_2\rangle$ for agent a_2 (line 3). Thus, the total cost of $R1$ is 8. $R1$ is then inserted into the sorted OPEN-list and will be expanded next. When validating the two-agents solution (line 8), a conflict $\langle a_1, a_2, E, 3\rangle$ is found. As a result, $R1$ is declared as non-goal. $R1$ is split and two children are generated (via the *generate-child()* function, also shown in Algorithm 1) to resolve the conflict (line 15). The left child U adds the constraint $\langle a_1, E, 3\rangle$ while the right child V adds the constraint $\langle a_2, E, 3\rangle$. The low-level search is now invoked (line 20) for U to find an optimal path for agent a_1 that also satisfies the new constraint. For this, a_1 must wait one time step at C (or at S_1 or A) and the path $\langle S_1, A, C, C, E, G_1\rangle$ is returned for a_1. The path for a_2, $\langle S_2, B, D, E, G_2\rangle$ remains unchanged in U. Since the cost of a_1 increased from 4 to 5 time steps the cost of U is now 9. as the sum of right child V is generated, also with cost 9. Both children are added to OPEN (line 16). In the final step U is chosen for expansion, and the underlying paths are validated. Since no conflicts exist, U is declared as a goal node (lines 8-10) and its solution is returned. Lines 12-13 are optional but speed up the search. They are the main contribution of this paper.

Meta-agent CBS (MA-CBS) MA-CBS(B) (Sharon et al. 2012b) is a generalization of CBS. When the number of conflicts between a given pair of agents exceeds a predefined parameter B, then conflicting agents are merged into a *meta-agent*. Meta agents are treated as a joint composite agent by the low-level solver. Basic CBS is, in fact, MA-CBS(∞), i.e., never merge agents.

Sensitivity of CBS Given a consistent h-function, A* must expand *all* nodes with $f < C^*$ (where C^* is the optimal solution cost) in order to guarantee optimality of the solution. No such *mandatory nodes* are known for CBS. In fact, CBS is very sensitive to the paths found by the low level and to the conflicts chosen to cause *split* as these can significantly influence the number of CT nodes. Consider the same example problem from Figure 1(I) but now assume that the paths found at the root of the CT by the low level are: $P_1' = \langle S_1, B, D, E, G_1\rangle$ for a_1 and $P_2 = \langle S_2, B, D, E, G_2\rangle$ for a_2 as shown in figure 1(III).[1] At the root ($R2$), the conflict $\langle a_1, a_2, B, 1\rangle$ is chosen to cause a split and the left child M is generated with path $P_1'' = \langle S_1, A, D, E, G_1\rangle$ (cost 8) but M includes another conflict $\langle a_1, a_2, D, 2\rangle$. It will also be split and its left child (O with path $P_1 = \langle S_1, A, C, E, G_1\rangle$, cost 8) is now identical to $R1$ (the root node of figure 1(II)). In O the conflict $\langle a_1, a_2, E, 3\rangle$ must be chosen and this is done in the same manner as described for figure 1(II). Fig-

[1] If a_2 were to be considered first, then, as first suggested by (Standley 2010) in his *Independence Detection* framework (ID), the low level of CBS for a_1 would try to avoid P_1' using a *conflict avoidance table* (See (Sharon et al. 2012a)). However, in our example a_1 was considered first and a_2 is forced to use P_2.

ure 1(II) (CT size 3) and figure 1(III) (CT size 7) correspond to running CBS on the same problem while choosing different paths and conflicts.

Improved CBS: Bypassing Conflicts

When a conflict is found between the paths of two agents CBS immediately splits the corresponding CT node into two children, each with its own new constraint. However, it is sometimes possible to prevent such a split and bypass the conflict by modifying the chosen path of one of the agents.

Consider again Figure 1(I) and assume that we start with the root of Figure 1(III) ($R2$ with paths P_1' and P_2) where conflict $\langle a_1, a_2, B, 1\rangle$ is found. CBS splits $R2$ as described above and generates two successors. At the left child M path $P_1'' = \langle S1, A, D, E, G1\rangle$ is assigned to agents a_1. However, $R2$ can *replace* path P_1' for agent a_1 with path P_1'', as it satisfies all of $R2.constraints$ and has the same cost. P_1'' is a better path for a_1 than path P_1' because the conflict at node B no longer occurs. Now, instead of a CT with 7 nodes we get a CT with only 5 nodes. Importantly, we note that since we aim to find the optimal solution, we only consider bypassing paths that have the same cost as the original path. Otherwise, to guarantee admissibility we must split the node.

Adding the option of bypassing requires only a small addition to the CBS pseudo code (lines 12-13 in Algorithm 1). After a conflict is found in node N, bypassing paths are searched for. If such a path is found, it is *adopted* by the corresponding CT node without the need to split N. Below, we provide a number of methods for finding such bypasses.

Definitions

(1) For each CT node N we use $N.NC$ to denote the total *number of conflicts* of the form (a_i, a_j, v, t) between the paths in $N.solution$. Calculating $N.NC$ is trivial.

(2) A path P_i' is a *valid bypass* to path P_i for agent a_i with respect to a conflict $C = \langle a_i, a_j, v, t\rangle$ and a CT node N, if the following conditions are satisfied: **(i)** Unlike P_i, P_i' does not include conflict C, **(ii)** $cost(P_i') = cost(P_i)$ and **(iii)** P_i and P_i' are both consistent with $N.constraints$.

(3) *Replacing* a path P_i at a CT node N means replacing P_i with a valid bypass P_i' for agent a_i. When this happens we say that P_i' was *adopted* by N.

Adopting a valid bypass may introduce more conflicts compared to the original path and potentially lead to worse overall runtime. Thus, we only allow to adopt bypasses that reduce $N.NC$. These are called *helpful bypasses*. That is:

(4) A valid bypass P_i' is a *helpful bypass* to P_i if $N'.NC < N.NC$ (where N' is the CT node that adopted P_i'). We use "$<$" (and not "$\leq$") in definition 4, to avoid an infinite loop that alternates between conflicts.

Next we cover methods for finding adoptable bypasses.

Bypass1: Peek at the Child

Our first method, denoted as *Bypass1* (BP1) peeks at either of the immediate children in the CT and tries to adopt their paths. This method was, in fact, demonstrated in our example of Figure 1(III) discussed above – where P_1'' was adopted by the root. Once the left child (M) is generated we notice that path P_1'' is a helpful bypass to path P_1' of $R2$. If P_1'' is adopted by $R2$ it would avoid the need to split $R2$ and branch according to the conflict $\langle a_1, a_2, B, 1\rangle$. In this case node M and its sibling node N are not added to the CT.

Algorithm 2: BP2. (BP1 changes line 1)

```
   Input: Node N
1  foreach l ∈ ST(N) in a best-first order do
2      C ← first conflict (a_i, a_j, v, t) in l
3      foreach agent a_i in C do
4          A ← generate child(l, (a_i, s, t))
5          if (A.cost = P.cost) and (A.NC < P.NC) then
6              N.solution ← A.solution
7              Insert N to OPEN
8              return true
9  return false
```

Formally, BP1 works as follows. Let N be a CT node where the paths for agents a_i and a_j in $N.solution$ are P_i and P_j, respectively. Assume that $N.solution$ includes a conflict $C = (a_i, a_j, v, t)$ that we want to bypass. We can now generate the left child and reveal the shortest path P_i' of a_i that satisfies the constraint (a_i, v, t) by using the *generate-child()* function. If P_i' is a helpful bypass to P_i wrt. conflict C, then P_i' is adopted by N without adding any new constraint to N. Importantly, while the left child of N was technically generated it will not be added to OPEN and to the CT. The right child will never be generated and the CT size is not increased. Similarly, if peeking at the left child failed because the resulting path was not a helpful bypass, the same peeking mechanism can be done for agent a_j at the right child. If both peek operations failed then we have these nodes at hand and insert them to OPEN normally.

While processing node N, BP1 doesn't incur any extra overhead over basic CBS due to peek operations. Basic CBS generates both children and adds both to OPEN. In the worst case when both peek operations fail, BP1 does exactly the same as CBS, i.e., generates and adds these children to OPEN. But if one of these nodes was found helpful then BP1 doesn't add new nodes to OPEN and might even avoid the need to generate and run a low-level search for the 2nd child.

The main reason for adopting a path from a child is that a split action is canceled and new nodes are not generated at this point. This can potentially save a significant amount of search due to a smaller size CT as shown in Figure 1.

Bypass2: Deep Search for Bypasses

Bypass2 (BP2) generalizes BP1. Pseudo code for both is provided in Algorithm 2. Define $ST(N)$ as the subtree below N (including node N) containing only nodes with the same cost as $N.cost$. BP2 searches (in best-first manner according to $N.NC$) through the entire set of nodes in $ST(N)$ (line 1) in order to find a *helpful descendent*, i.e., a descendent $N' \in ST(N)$ such that $N'.NC < N.NC$. In line 4 it calls the function *generate-child* (shown in Algorithm1) which invokes the low-level. If the child is a helpful descendent (has the same cost but with fewer conflicts, line 5) then

k	Ins.	Runtime (ms)			Success rate %		
		CBS	BP1	BP2	CBS	BP1	BP2
5	100	452	**230**	257	100	**100**	100
6	100	153	144	**133**	100	**100**	100
7	95	367	227	**223**	95	**96**	95
8	85	10,692	**5,156**	7,615	85	**88**	88
9	59	20,374	**4,835**	9,900	60	65	**67**
10	36	43,488	**18,170**	21,970	40	**47**	42

Table 1: 5×5 grid, MA-CBS(5)

this low-level path is returned and adopted by N (lines 6-7).[2] BP1 uses the same pseudo code except that in line 1 it only allows the two immediate children.

It is important to note that in practical implementation when BP2 failed to find a helpful descendent (line 9), then we have all the frontier nodes of this search at hand and they can be passed to Algorithm 1 which will directly add them to OPEN without the need to generate them again (and invoke the low-level again for these nodes).

It is also important to note that node N in its "new suit" after adopting the solution of one of its descendants will now be the best node in OPEN and will be chosen for expansion next.[3] Therefore, if it again has a helpful descendent then the bypass process will be repeated here. In fact, the next *real* split will occur only when a bypass call fails.

Experimental results

Performance of BP1 and BP2 We experimented with CBS and with MA-CBS(B) with various values for B. In general, in all these settings BP1 outperformed CBS by up to an order of magnitude. Nevertheless, in very rare cases (a few out of 25,000) slowdown was observed due to bad tie breaking and due to recurring conflicts. In some domains BP2 further outperformed BP1 but in others it was slightly worse.

Table 1 shows representative results (of MA-CBS(5)) for 100 random instances of a 5×5 grid with 15% obstacles. As was done by Sharon et al. (2013; 2012a), we report the *success rate* - the number of instances solved by each of the algorithms within 5 minutes. We also report the average runtime (in ms) over instances solved by all three algorithms within 5 minutes (shown in the *Ins* column). BP1 clearly outperformed CBS in its success rate and provided speedup of up to a factor of 5 (for 9 agents). BP2 incurs more overhead than BP1 in its search but could not find more bypasses as this domain is dense with agents and most bypasses were found by BP1. Thus, it was generally worse than BP1.

Figure 2 shows the success rate over 300 instances from the three standard benchmark maps (brc202d, den520d, ost003d) of the game *Dragon Age: Origins* (DAO) (Sturtevant 2012) that were used by Sharon et al. (2013; 2012a), 100 instances per map. It also shows the average CPU time (in seconds) for all instances that were solved by all three algorithms up to 55 agents and for the BP variants only, for 60 agents or more; hence the jump in the curves at 60. Here too, BP1 significantly outperformed CBS by up to an order of magnitude (e.g., 50 agents). This domain is less dense with agents and many times BP1 could not find bypasses that BP2 did find. BP2 had better success rate than BP1 and provided a further factor of up to 1.5 speedup.

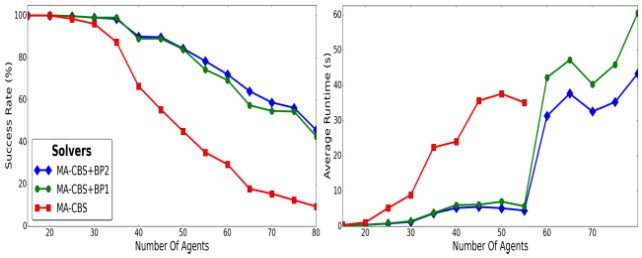

Figure 2: Success rate and runtime for the three DAO maps

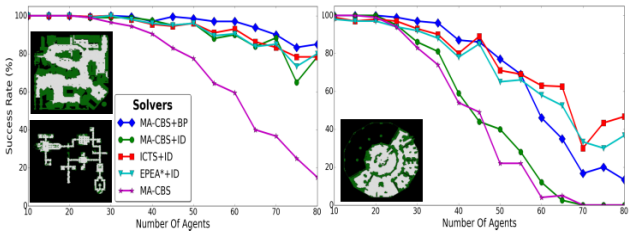

Figure 3: Success rate: brc202+den520 (left), ost003 (right).

Comparison with other algorithms Finally, we compared CBS+BP to the following other MAPF solvers: **(1)** our best CBS/MA-CBS variant.[4] **(2)** EPEA* (Felner et al. 2012) which is an enhanced version of A* designed for cases with large branching factor. **(3)** ICTS+p (Sharon et al. 2013). Following Sharon et al. (2012b; 2013) all these were enhanced by the *Independent Detection* framework (ID) (Standley 2010) which identifies independent groups of agents and runs separate solvers for each group. In addition, we executed **(4)** MA-CBS enhanced with our best BP.[5] Figure 3(left) presents the success rates of each of the solvers on both den520 and brc202 (total of 200 instances). MA-CBS+BP clearly outperforms all the other solvers on these maps. Figure 3(right) presents the results on ost003 (100 instances). MA-CBS+BP outperformed the other MA-CBS-based solvers on this map too. However, here the trends shift and ICTS+ID was the best and EPEA*+ID was second for more than 55 agents. In this map for this range of agents, there were too many conflicts and the CBS variants are in-

[2]This is called *first-fit adoption*. By contrast, *best-fit adoption* continues to search all nodes in $ST(N)$ and the path of the helpful descendant with the smallest NC is returned. We can also parameterize the amount of search in $ST(N)$ and halt once we reach a predefined threshold for the number of allowed nodes Q. BP1 is a special case of this where $Q = 2$, while BP2 is the case where $B = \infty$. We experimented with all these combinations but only report the best results for BP2 (of first-fit adoption and $Q = \infty$).

[3]A mechanism like *immediate expand* (Stern et al. 2010) which bypasses OPEN can be efficiently activated here.

[4]MA-CBS(10) for den520 and and MA-CBS(100) for ost003 and brc202. EPEA* was used as the low-level solver.

[5]Combining MA-CBS with both ID and BP (not shown) degraded the performance due to duplicating some of the work.

ferior. This supports the claim that no MAPF solver is best across *all* circumstances and topologies.

Conclusions and Future Work

When CBS is used, adding BP on top is a great enhancement. This was demonstrated empirically on standard benchmarks. Future work will (1) further investigate search for bypasses (2) try to find better low-level paths in the first place (3) compare and better understand the pros and cons of the various MAPF algorithms under different circumstances.

Acknowledgments: The research was supported by the Israeli Science Foundation (ISF). Grant #417/13 to A. Felner.

References

Erdem, E.; Kisa, D. G.; Oztok, U.; and Schueller, P. 2013. A general formal framework for pathfinding problems with multiple agents. In *AAAI*.

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

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

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

Sharon, G.; Stern, R.; Goldenberg, M.; and Felner, A. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artif. Intell.* 195:470–495.

Standley, T. S. 2010. Finding optimal solutions to cooperative pathfinding problems. In *AAAI*.

Stern, R.; Kulberis, T.; Felner, A.; and Holte, R. 2010. Using lookaheads with optimal best-first search. In *AAAI*.

Sturtevant, N. 2012. Benchmarks for grid-based pathfinding. *Transactions on Computational Intelligence and AI in Games* 4(2):144 – 148.

Surynek, P. 2012. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *The Pacific Rim International Conference on Artificial Intelligence (PRICAI)*. 564–576.

Wagner, G., and Choset, H. 2011. M*: A complete multirobot path planning algorithm with performance bounds. In *IROS*, 3260–3267.

Yu, J., and LaValle, S. M. 2013. Planning optimal paths for multiple robots on graphs. In *International Conference on Robotics and Automation (ICRA)*, 3612–3617.