AIFA : Report on CBS

-Aryan Saxena
-Prabhu Chandra
-Vishal Katroth
-Ajay Kumar

I*ntroduction*: In the multi-agent pathfinding problem (MAPF) we are given a set of agents each with respective start and goal positions. The task is to find paths for all agents while avoiding collisions. We have used Conflict Based Search algorithm to solve the *Multi-agent-pathfinding* problem. A brief introduction about the search method is provided here in the below report.

Conflict Based Search algorithm : CBS is a two-level algorithm that does not convert the problem into the single 'joint agent' model. At the high level, a search is performed on a *Conflict Tree* (CT) which is a tree based on conflicts between individual agents. Each node in the CT represents a set of constraints on the motion of the agents. At the low level, fast single-agent searches are performed to satisfy the constraints imposed by the high level CT node. In many cases this two-level formulation enables CBS to examine fewer states than A* while still maintaining optimality. Single-agent pathfinding is the problem of finding a path between two vertices in a graph. Solving pathfinding problems optimally is commonly done with search algorithms based on the A* algorithm. Such algorithms perform a best-first search that is guided by $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the shortest known path from the start state to state n and $h(n)$ is a heuristic function estimating the cost from n to the nearest goal state. If the heuristic function h is admissible, meaning that it never overestimates the shortest path from n to the goal, then A* (and other algorithms that are guided by the same cost function) are guaranteed to find an optimal path from the start state to a goal state, if one exists.

The following definitions are used in the remainder of the paper.

- 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 is a tuple (ai,v,t) where agent ai is prohibited from occupying vertex v at time step t. During the course of the algorithm, agents will be associated with constraints. A consistent path for agent ai 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 any agent ai is consistent with the constraints of ai.
- A conflict is a tuple (ai,aj,v,t) where agent ai and agent aj 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 individual paths are consistent with the constraints associated with their agents, these paths still have conflicts.

The key idea of CBS is to grow a set of constraints 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 finds paths for individual agents that are consistent with the new constraints. Next, we describe each part of this process in more detail.

The constraint tree :

At the high level, CBS searches a tree called the *constraint tree* (CT). A CT is a binary tree. Each node *N* in the CT consists of:

1. A set of constraints (N.constraints). Each of these constraints belongs to a single agent. 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 ai must be consistent with the constraints of ai. Such paths are found by the low-level search.
3. The total cost (N.cost) of the current solution (summed over all the single-agent path costs). This cost is referred to as the *f*-value of node *N*.

Node *N* in the CT is a goal node when N.solution is valid, i.e., the set of paths for all agents has no conflicts. The high level performs a best-first search on the CT where nodes are ordered by their costs. In our implementation, ties are broken in favor of CT nodes whose associated solution contains fewer conflicts. Further ties were broken in a FIFO manner.

Processing a node in the CT

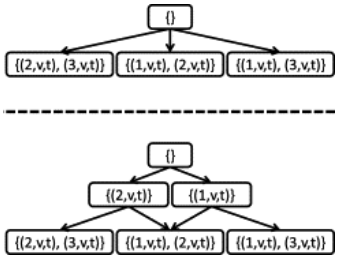
Given the list of constraints for a node *N* of the CT, the low-level search is invoked. The low-level search (described in detail below) returns one shortest path for each agent, ai, that is consistent with all the constraints associated with ai in node *N*. Once a consistent path has been found for each agent (with respect to its own constraints) these paths are then *validated* with respect to the other agents. The *validation* is performed by iterating through all time steps and matching the locations reserved by all agents. If no two agents plan to be at the same location at the same time, 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 = (ai,aj,v,t) is found for two or more agents ai and aj, the validation halts and the node is declared a non-goal node.

Resolving a conflict : Given a non-goal CT node *N* whose solution N.solution includes a *conflict* Cn = (ai,aj,v,t) we know that in any valid solution, at most one of the conflicting agents (ai and aj) may occupy vertex *v* at time *t*. Therefore, at least one of the constraints (ai,v,t) or (aj,v,t) must be added to the set of

constraints in N.constraints. To guarantee optimality, both possibilities are examined and node *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 (ai,v,t) and the right child adds the constraint (aj,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 ai which is associated with the newly added constraint. The paths of other agents remain the same as no new constraints are added for them.

Conflicts of k>2 agents It may be the case that while performing the validation 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. This is illustrated in Figure below. The top tree represents a variant of CT where *k*-way branching is allowed for a single conflict that includes *k*-agents for the case where k=3. Each new successor adds k−1(=2) new constraints (on all agents but one). The bottom tree presents a binary CT for the same problem. Note that the bottom middle state is a duplicate state, and if duplicate detection is not applied there will be two occurrences of this node instead of one. As can be seen the size of the deepest layer in both trees is identical. The complexity of the two approaches is similar, as they both will end up with *k* nodes, each with k−1 new constraints. For simplicity we implemented and describe only the second option.



(ai,aj,v1,v2,t) where two agents “swap” locations (ai moves from v1 to v2 while aj moves from v2 to v1) between time step *t* to time step t+1. An edge constraint is defined as (ai,v1,v2,t), where agent ai is prohibited of starting to move along the edge from v1 to v2 at time step *t* (and reaching v2 at time step t+1). When applicable, edge conflicts are treated by the high level in the same manner as vertex conflicts.

When validating the two-agent solution given by the two individual paths (line 7), a conflict is found when both agents arrive at vertex *D* at time step 2. This creates a conflict (a1,a2,D,2) (line 10). As a result, the root is declared as a non-goal and two children are generated in order to resolve the conflict (line 19). The left child, adds the constraint (a1,D,2) while the right child adds the constraint (a2,D,2). The low-level search is now invoked (line 23) for the left child to find an optimal path that also satisfies the new constraint. For this, a1 must wait one time step either at A1 or at S1 and the path ⟨S1,A1,A1,D,G1⟩ is returned for a1. The path for a2, ⟨S2,B1,D,G2⟩ 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 inserted to OPEN (line 26). In the next iteration of the while loop (line 5) the left child is chosen for expansion, and the underlying paths are validated. Since no conflicts exist, the left child is declared a goal node (line 9) and its solution is returned as an optimal solution.

Low level: find paths for CT nodes :

The low-level search is given an agent, ai, and the set of constraints associated with ai. It performs a search in the underlying graph to find an optimal path for agent ai that satisfies all its constraints while completely ignoring the other agents. The

search space for the low-level search has two dimensions: the spatial dimension and the time dimension.2 Any single-agent pathfinding algorithm can be used to find the path for agent ai, while verifying that the constraints are satisfied. We implemented the low-level search of CBS with A* which handled the constraints as follows. Whenever a state (v,t) is generated where *v* is a location and *t* a time step and there exists a constraint (ai,v,t) in the current CT (high-level) node, this state is discarded. The heuristic we used is the shortest path in the spatial dimension, ignoring other agents and constraints.

For cases where two low-level A* states have the same *f*-value, we used a tie-breaking policy based on Standley's tie-breaking *conflict avoidance table* (CAT) (described in Section 3.3.4). States that contain a conflict with a smaller number of other agents are preferred. For example, if states s1=(v1,t1) and s2=(v2,t2) have the same *f* value, but v1 is used by two other agents at time t1 while v2 is not used by any other agent at time t2, then s2 will be expanded first. This tie-breaking policy improves the total running time by a factor of 2 compared to arbitrary tie breaking. Duplicate states detection and pruning (DD) speeds up the low-level procedure. Unlike single-agent pathfinding, the low-level state-space also includes the time dimension and dynamic 'obstacles' caused by constraints. Therefore, two states are considered duplicates if both the position of ai and the time step are identical in both states.