Conflict-Based Search for Multi-Agent Path Finding

Woonsang Kang

December 4, 2023

1 Introduction

Multi-agent path finding (MAPF) is a pivotal domain in the realm of robotics and artificial intelligence, focusing on the coordination of multiple autonomous agents as they navigate within a shared environment. Each agent must traverse from its starting position to a pre-defined goal, ensuring that paths do not result in any form of collision or congestion. The purpose of MAPF is not merely to avoid obstacles and conflicts. It centers on optimizing various parameters, such as minimizing total travel distance, reducing energy consumption, or balancing the workload among agents. With the rise of collaborative robotic systems in various sectors - from warehouses teeming with automated guided vehicles to urban skies dotted with delivery drones, and even hospitals with robot assistants - the necessity of MAPF becomes abundantly clear. Effective solutions in this domain not only amplify the efficiency and throughput of operations but are also paramount in ensuring safety, preventing costly damages, and fostering harmonious coexistence of multiple robots in intertwined trajectories. As our reliance on autonomous systems grows, so does the imperative for sophisticated MAPF strategies that can cater to the intricacies of diverse real-world scenarios.

2 Conflict-Based Search

Conflict-Based Search (CBS) is a two-level search algorithm used for MAPF. At the high-level, it builds a `constraint tree` where each node represents a set of constraints imposed on agents. When a conflict arises between two agents, CBS branches and creates a new (high-level) node in the CT with added constraints to resolve the conflict. At the low level, CBS uses a standard path finding algorithm, like A*, to find individual paths for agents that respect the constraints from the CT. CBS continues to iterate between the high and low levels until it either finds a solution with paths for all agents that don't conflict with each other, or determines that no such paths exist. Algorithm 1 presents the pseudo code for CBS.

Algorithm 1 Conflict-Based Search

```
1: function CBS(Agents, Map)
2:     Root ← CreateNode
3:     Root.Constraints ← ∅
4:     Root.Solutions ← FindIndividualPaths(Agents, Map)
5:     Open ← Priority queue containing Root
6:     while Open ≠ ∅ do
7:         P ← Open.pop()
8:         Conflict ← FindFirstConflict(P.Solutions)
9:         if Conflict = null then return P.Solutions        ▷ Solution found
10:        end if
11:        for agent in Conflict do
12:            Q ← CreateNode
13:            Q.Constraints ← P.Constraints ∪ {new_constraints}
14:            Q.Solutions ← P.Solutions
15:            UpdatePath(Q.Solutions[agent], Q.Constraints)
16:            if PathValid(Q.Solutions[agent], Q.Constraints) then
17:                Open.push(Q)
18:            end if
19:        end for
20:    end while
21:    Return Failure
22: end function
```

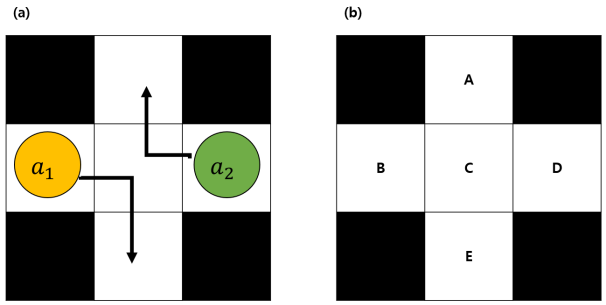

Figure 1: Simple scenario with two agents and five vertices.

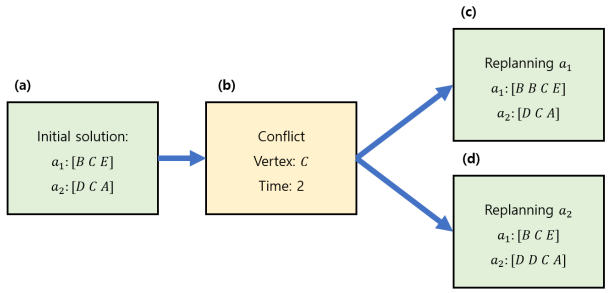

Figure 2: The example for `constraint tree` corresponding to Figure 1.

3 Simple Example

In this section, we provide a detailed explanation with a simple example. Let's consider the following scenario in Figure 1. An agent named a_1 needs to move from **B** to **E**, and another agent a_2, must travel from **D** to **A**. If we calculate the initial solution, it can be represented a_1 : [**B, C, E**] and a_2 : [**D, C, A**] as shown in Figure 2.a. However, at the second time step, we can identify a conflict at the vertex **C** as shown in Figure 2.b. To resolve this conflict, there are two possible methods. One is to replan for a_1, as illustrated in Figure 2.c. By stopping a_1 for 1 time step and proceeding with a_2 first, the conflict can be resolved. And the other is to replan for a_2, as shown in Figure 2.d. By stopping a_2 for 1 time step and proceeding with a_1 first, the conflict can be resolved.

4 Complicated Example

By expanding `constraint tree` in the way illustrated above, we can obtain a solution without any conflicts. The solution with the lowest cost obtained as a result of the high-level search becomes the final solution. As shown in Figure 3, let's apply CBS to a slightly more complex scenario. Then, we can obtain a solution as shown in Figure 4.

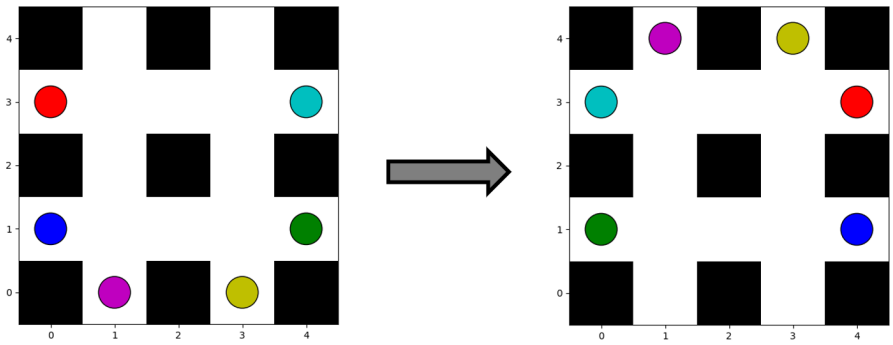

Figure 3: The complicated scenario for CBS example.

```
agent_0 t0->(0, 1) t1->(1, 1) t2->(2, 1) t3->(3, 1) t4->(4, 1)
agent_1 t0->(0, 3) t1->(1, 3) t2->(1, 3) t3->(1, 2) t4->(1, 3) t5->(2, 3) t6->(3, 3) t7->(4, 3)
agent_2 t0->(4, 1) t1->(4, 1) t2->(3, 1) t3->(3, 0) t4->(3, 1) t5->(2, 1) t6->(1, 1) t7->(0, 1)
agent_3 t0->(4, 3) t1->(3, 3) t2->(2, 3) t3->(1, 3) t4->(0, 3)
agent_4 t0->(1, 0) t1->(1, 0) t2->(1, 1) t3->(1, 1) t4->(1, 2) t5->(1, 3) t6->(1, 4)
agent_5 t0->(3, 0) t1->(3, 1) t2->(3, 2) t3->(3, 3) t4->(3, 4)
```

Figure 4: The solution of complicated example in Figure 3.