Clique Analysis and Bypassing in Continuous-Time Conflict-Based Search

Thayne T. Walker[1 2], Nathan R. Sturtevant[3] and Ariel Felner[4]
[1]University of Denver, USA
[2]Lockheed Martin Corp., USA
[3]Department of Computing Science, Alberta Machine Intelligence Institute (Amii), University of Alberta, Canada
[4]Ben-Gurion University of Negev, Israel
thayne.walker@du.edu, nathanst@ualberta.ca, felner@bgu.ac.il

Abstract

While the study of unit-cost Multi-Agent Pathfinding (MAPF) problems has been popular, many real-world problems require continuous time and costs. In this context, this paper studies symmetry-breaking enhancements for Continuous-Time Conflict-Based Search (CCBS), a solver for continuous-time MAPF. Resolving conflict symmetries in MAPF can require an exponential amount of work. We adapt known symmetry-breaking enhancements from unit-cost domains for CCBS: *bypassing* and *biclique constraints*. We then improve upon these to produce a new state-of-the-art algorithm: CCBS with disjoint k-partite cliques (CCBS+DK). Finally, we show empirically that CCBS+DK solves for up to 20% more agents in the same amount of time when compared to previous state-of-the-art.

Introduction

The objective of multi-agent pathfinding (MAPF) is to find non-conflicting paths for multiple agents being routed on a graph. Agents paths are conflicting if at any time their shapes overlap. MAPF has applications in warehouses (Li et al. 2021b), package delivery (Choudhury et al. 2021), games (Botea et al. 2013) and firefighting (Roldán-Gómez, González-Gironda, and Barrientos 2021).

A significant amount of work has focused on "classic" MAPF where agents move on a grid or planar graph with unit-cost edges (Stern et al. 2019). Additionally, all actions take one time step and agents always occupy exactly one vertex. These limitations simplify the problem, but cannot be applied to domains which may exhibit variable size agents and continuous-time, variable-duration motion and wait actions. We seek optimal solutions to the continuous-time MAPF problem ($MAPF_R$) (Walker, Sturtevant, and Felner 2018; Andreychuk et al. 2019), denoted with subscript "R" for real-valued action durations and costs on general graphs (e.g., planar, non-panar, unit-cost and non-unit cost graphs).

Continuous-Time Conflict-Based Search (CCBS) (Andreychuk et al. 2022) is a solver for $MAPF_R$. CCBS re-formulates the Conflict-Based Search (CBS) algorithm (Sharon et al. 2015) to allow variable-duration wait actions and constraints which account for continuous-time execution. CCBS was shown to be effective on several settings that are inspired by real-world applications. Additional enhancements, such as heuristics (Li et al. 2019a), disjoint splitting (DS) (Li et al. 2019b) and conflict prioritization (Boyarski et al. 2015b) were added to CCBS (Andreychuk et al. 2021). These enhancements improve the runtime of CCBS. In contrast to other prior optimal continuous-time approaches (Walker, Sturtevant, and Felner 2020) which assume only fixed-duration wait actions, CCBS plans optimal, arbitrary duration wait times.

CCBS represents a significant advancement for $MAPF_R$. However, efficient conflict resolution continues to pose a problem for this algorithm. A *conflict symmetry* (Li et al. 2021a) occurs when two or more agents have many different ways to reach their goals efficiently, but all combinations of them result in a conflict. For optimal algorithms like CCBS, this means that all equal-cost path combinations must be explored before proving that a cost increase is necessary to avoid the conflict. Conflict symmetries can cause an exponential amount of work to resolve (Li et al. 2021a). In this paper, we address conflict symmetries by adapting and building new symmetry-breaking enhancements for CCBS:

- We adapt the bypass (BP) enhancement (Boyarski et al. 2015a), originally for CBS, to be used with CCBS.
- We re-formulate Biclique Constraints (BC) (Walker, Sturtevant, and Felner 2020), originally formulated for CBS, to be used with CCBS.
- We create a new algorithm: CCBS with *disjoint bicliques* (CCBS+DB) by combining disjoint splitting (DS) (Li et al. 2019b) as formulated for CCBS (Andreychuk et al. 2021) with BC.
- We extend CCBS+DB to use k-partite cliques to create CCBS with *disjoint k-partite cliques* (CCBS+DK).

This paper is organized as follows: We first provide a definition of the $MAPF_R$ problem. Next, we describe the CCBS algorithm and related work. This is followed by a description of the new enhancements. Finally, we present a comprehensive ablation study of the enhancements.

Problem Definition

MAPF was originally defined for a "classic" setting (Stern et al. 2019) where the movements of agents are coordinated on a unit-cost planar graph. Edges have a unit cost/unit

Copyright © 2024, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

time duration and agents occupy a point in space. Thus, two agents can only have conflicts when on the same vertex at the same time, or traversing the same edge in opposite directions. MAPF_R (Walker, Sturtevant, and Felner 2018; Andreychuk et al. 2019), an extension of MAPF for *real-valued* action durations, uses a weighted graph $G{=}(V,E)$ which may be non-planar. Every vertex $v{\in}V$ is associated with coordinates in a metric space and every edge $e{\in}E$ is associated with a positive real-valued edge weight $w(e){\in}\mathbb{R}_+$. For the purposes of this paper, weights represent the times it takes to traverse edges. However, time duration and cost can be treated separately. There are k agents, $A{=}\{1,..,k\}$. Each agent has a start and a goal vertex $V_s{=}\{start_1,..,start_k\}{\subseteq}V$ and $V_g{=}\{goal_1,..,goal_k\}{\subseteq}V$ such that $start_i{\neq}start_j$ and $goal_i{\neq}goal_j$ for all $i{\neq}j$.

A *solution* to a MAPF_R instance is $\Pi{=}\{\pi_1,..,\pi_k\}$, a set of single-agent *paths* which are sequences of *states*. A state $s{=}(v,t)$ is a pair composed of a vertex $v{\in}V$ and a time $t{\in}\mathbb{R}_+$. A path for agent i is a sequence of $d{+}1$ states $\pi_i{=}[s_i^0,..,s_i^d]$, where $s_i^0{=}(start_i,0)$ and $s_i^d{=}(goal_i,t_g)$ where t_g is the time the agent arrives at its goal and all vertices in the path follow edges in E.

Agents have a shape which is situated relative to an agent-specific *reference point* (Li et al. 2019c). Agents' shapes may vary, but this paper uses circular agents. Agents move along edges via a straight vector in the metric space. Traversing an edge is called an *action*, $a{=}(s,s')$, where s and s' are a pair of neighboring states. *Wait actions* of any positive, real-valued duration is allowed at any vertex. Variable velocity and acceleration are allowed. For simplicity, this paper assumes fixed velocity motion with no acceleration.

A *conflict* happens when two agents perform actions $\langle a_i,a_j\rangle$ such that their shapes overlap at the same time (Walker, Sturtevant, and Felner 2018). A *feasible solution* has no conflicts between any pairs of its constituent paths. The objective is to minimize the sum-of-costs $c(\Pi){=}\sum_{\pi\in\Pi}c(\pi)$, where $c(\pi)$ is the sum of edge weights of all edges traversed in π. We seek Π^*, a solution with minimal cost among all feasible solutions. Optimization of the classic MAPF problem is NP-hard (Yu and LaValle 2013), hence, optimization of the MAPF_R problem is also NP-hard.

Background

We now describe CCBS and other prior work.

Conflict-Based Search

Continuous-time Conflict-Based Search (CCBS) (Andreychuk et al. 2022) is based on the classic Conflict-Based Search (CBS) (Sharon et al. 2015) algorithm, so we describe it next. CBS performs search on two levels. The *high level* searches a constraint tree (CT). Each node N in the CT contains a solution $N.\Pi$, and a set of constraints $N.C$. Each path $\pi_i{\in}N.\Pi$ of agent i in N is constructed using a *low-level* search which respects constraints. A *constraint* blocks an agent from performing action(s) and is defined as a tuple $\langle i,v,t\rangle$, where i is the agent, v is a vertex (or edge) and t is the time step in which the agent must avoid the vertex (or edge). Next, CBS checks for conflicts between any pairs of paths π_i and π_j in $N.\Pi$. If $N.\Pi$ contains no conflict, then N is a goal node and CBS terminates. If $N.\Pi$ contains a conflict between any π_i and π_j, then CBS performs a *split*, meaning that it generates two child nodes N_i and N_j of N and adds constraints c_i and c_j to $N_i.C$ and $N_j.C$ respectively. CBS systematically checks for conflicts, generates child nodes with constraints to avoid the conflict and re-plans the conflicting paths with the new constraints and other constraints inherited from ancestor nodes. CCBS prioritizes the search by the total cost of $N.\Pi$. It terminates when a feasible solution is found.

Various improvements for CBS have been proposed such as adding high-level heuristics (Li et al. 2019a), conflict prioritization (Boyarski et al. 2015b), disjoint splitting (Li et al. 2019b) and conflict symmetry resolution (Zhang et al. 2020; Li et al. 2021a). Some enhancements were also proposed for MAPF_R, such as kinodynamic constraints (Kottinger, Almagor, and Lahijanian 2022; Wen, Liu, and Li 2022), biclique constraints (BC) (Walker, Sturtevant, and Felner 2020) (which will be described later) and CCBS itself (Andreychuk et al. 2022).

Safe Interval Path Planning

Safe Interval Path Planning (SIPP) (Phillips and Likhachev 2011) is an algorithm for planning a single agent on the same graph as moving obstacles. In the case of CCBS, the moving obstacles are other agents. SIPP uses an A*-based algorithm with a specialized successor generation routine. SIPP omits the generation of actions which would result in conflicts and adds wait actions with a specific duration to avoid conflicts. SIPP does this by computing a set of *safe intervals* for each vertex, that is, time intervals in which an agent may occupy a vertex without conflicting with moving obstacles. In this way, SIPP guarantees conflict-free shortest paths.

Continuous-Time Conflict-Based Search

CCBS (Andreychuk et al. 2022) modifies CBS by allowing continuous-time actions. This is accomplished by adding additional functionality to CBS:

- CCBS uses continuous-time-and-space collision detection. It uses arbitrary wait times instead of fixed duration wait actions.
- CCBS handles *durative* conflicts, (where agents' shapes overlap for a period of time), by utilizing time-range constraints (Atzmon et al. 2018).
- CCBS uses SIPP (Phillips and Likhachev 2011) at the low level. CCBS interprets time-range constraints (i.e., unsafe intervals) as safe intervals for SIPP.

We now describe CCBS in detail. An example for CCBS is illustrated in Figure 1. Figure 1(b) shows a problem instance on the simple planning graph in Figure 1(a) in which three agents exist, one at each vertex. Each of the agents needs to rotate one edge in the clockwise direction (or two edges in the counter-clockwise direction) in order to reach their goal. Assuming an agent radius of $\sqrt{2}/4$, actions $A{\rightarrow}B$ and $C{\rightarrow}A$ conflict if taken simultaneously.

After CCBS detects the conflict, *time-range* constraints are constructed for the agents. Time-range constraints block

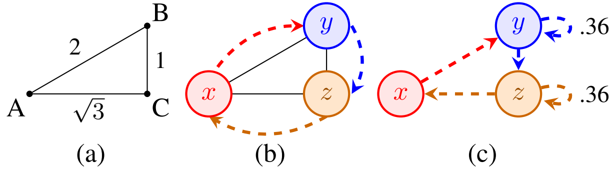

Figure 1: (a) A planning graph, (b) a MAPF_R instance, and (c) a solution that uses fractional wait times.

agents from performing actions inside of a given time range. This is done by computing the delay time necessary for the action $C \rightarrow A$ to avoid conflict with the action $A \rightarrow B$ (and vice-versa). This delay time is used to create an *unsafe interval*, the interval in which, if the action is taken, it will conflict. In this case, the unsafe interval for action $C \rightarrow A$ is $[0, 0.36)$. Hence, if action $C \rightarrow A$ is delayed by 0.36, it can be executed without conflicting with action $A \rightarrow B$.

After CCBS constructs the time-range constraint for agent z, the low-level SIPP solver is called. CCBS re-interprets the unsafe interval $[0, 0.36)$ to the safe interval, $[0.36, \infty)$ for SIPP. Agent z will be forced to wait 0.36 time steps before executing action $C \rightarrow A$, and the conflict with the action $A \rightarrow B$ is avoided. The wait action is shown by the self-loop on agent z in the solution, Figure 1(c).

This problem instance is unsolvable without a non-unit cost wait action. If, for example, agent z were to wait one full time step, it would cause agent y to wait one full time step as well, causing it to conflict with agent x. This is one reason why arbitrary-duration wait actions are important for MAPF_R. In addition, lower-cost solutions are possible with arbitrary-duration wait actions, since agents are allowed to wait for fractional times instead of whole time steps.

There are several enhancements for CCBS as described so far. The CCBS authors introduced a high-level heuristic based on the max-weight independent set problem. It was formerly formulated as as an integer linear program (ILP) for classic MAPF (Li et al. 2019a), but reformulated for continuous time as a linear program (LP) (Andreychuk et al. 2021). Finally, a special formulation of disjoint splitting (Andreychuk et al. 2021) was added to CCBS. Disjoint splitting is now explained in further detail.

Disjoint Splitting

Disjoint splitting (DS) (Li et al. 2019b) is a technique for CBS which helps avoid resolving the same conflict multiple times in different sub-trees of the CT. The split procedure for DS is as follows: one child node uses a *negative constraint* defined as a tuple $\langle i, v, t \rangle$ which causes the agent (i) to avoid a vertex (v) at a specific time (t). The other child node uses a *positive constraint*, $\langle i, v, t \rangle$, which forces the agent (i) to visit a vertex (v) at a specific time (t). A positive constraint for agent i also acts as a negative constraint for all other agents so that they avoid conflicting with agent i. Positive constraints are enforced at the low level by adding time-specific sub-goals or *landmarks* to the search. It was shown that DS helps CBS to do less work in general (Li et al. 2019b).

The implementation of DS for CCBS differs from CBS in two ways (Andreychuk et al. 2021): (1) Constraints for agent i in CCBS include a time range $\langle i, v, (t_1, t_2] \rangle$ (Atzmon et al. 2018). Since the arrival at a landmark is allowed at multiple times, special logic is required to determine which exact time is optimal and feasible with respect to all other constraints. (2) Unlike DS for Classic MAPF, positive constraints do not act as a negative constraint for all other agents. Instead, a single negative constraint is added for agent j to help it avoid the landmark for agent i.

This second difference is a limitation that must be solved in CCBS using bipartite analysis, covered later in this paper. Despite this weakness, DS yields consistent improvements in runtime performance versus the original splitting method (Andreychuk et al. 2021).

New Enhancements

Bypass

The bypass enhancement (BP) (Boyarski et al. 2015a) is a technique used to avoid some splits in the CT. BP was never implemented for CCBS, and to our knowledge, no study has been performed to determine its effectiveness for CCBS. Our intent in including BP in this paper is to experiment with its effectiveness in continuous-time domains. Our findings show that it is very effective in problem instances with similarities to "classic" MAPF, but much less effective in certain continuous-time settings. These details will be discussed in the empirical results section.

The implementation of BP for CCBS is straightforward, and follows the original formulation, with some adaptations for continuous-time. BP (for both CBS and CCBS) inspects the paths for two agents involved in a conflict. If a new path of the same cost is available for one of the conflicting agents such that: (1) the new path does not have an increased cost, (2) the new path respects new constraints which are required to avoid the conflict that caused the split and (3) the new path has fewer conflicts with all agents than the respective path in the parent node, this new path is called a *bypass*. If a bypass is found, child nodes are not generated, instead, the current node is updated with the bypass path for one of the agents and re-inserted into the OPEN list. This enhancement improves performance by avoiding splits in the tree which would otherwise result in two new sub-trees.

Biclique Constraints

Biclique constraints (BC) were empirically studied for MAPF_R (Walker, Sturtevant, and Felner 2020) with unit-cost wait actions. We now study BC with variable-duration wait actions. Recall from the discussion of CBS that during a split, agents i and j each receive a new constraint c_i and c_j in their respective child nodes. For BC, CCBS must be combined with Multi-Constraint CBS (Li et al. 2019c) which allows each agent to receive *sets* of one or more new constraints C_i and C_j, respectively. Constraint sets are *valid* iff no solution exists when both agents i and j violate any constraints $c_i \in C_i$ and $c_j \in C_j$, respectively at the same time. Thus, completeness is ensured only when the

actions blocked by C_i are mutually conflicting with all actions blocked by C_j. BC builds valid sets of constraints for MAPF$_R$ using *bipartite conflict graphs*.

Bipartite Conflict Graphs Figure 2(a) shows an example problem instance where two agents must cross paths. Figure 2(b) shows an enumeration of all actions available to two agents at overlapping timeframes, (wait actions omitted). Figure 2(d) shows the bipartite conflict graph for the enumerated actions. A bipartite conflict graph (BCG) (Walker, Sturtevant, and Felner 2020) is constructed by creating two sets of nodes for the actions available to the two agents during overlapping timeframes. The nodes for one agent are arranged on the left, and nodes for the other agent are arranged on the right. Then edges are added between nodes for pairs of actions that conflict. The graph is bipartite because no node on the left is connected to any other node on the left, similarly for nodes on the right, but nodes on the left may be connected to nodes on the right.

In order to choose a valid set of constraints, the nodes chosen must form a bipartite clique or *biclique* in the BCG. That is, every node chosen for the set on the left must be connected to every node chosen for the set on the right of the BCG. A biclique is shown with thick lines in Figure 2(d), where the set of nodes in the biclique is $\{2, 3\}$ and $\{4, 5\}$ respectively. In practice, one can find a *max-vertex biclique* (a biclique with a maximal number of vertices) in polynomial time (Walker, Sturtevant, and Felner 2020).

Time-Annotated Bipartite Conflict Graphs Biclique nodes in a BCG can be annotated with unsafe intervals for SIPP. This is done by computing the unsafe times (i.e., the amount of time one agent should wait to avoid conflict) for each neighboring node in the biclique as shown for action 2 in Figure 2(c). Unsafe interval computation can be done using a binary search approach (Yakovlev and Andreychuk 2017) or, for circular agents, using an algebraic approach (Walker and Sturtevant 2019). In this example, action 2 cannot be performed in the time range $[0, 0.58)$ in order to avoid conflict with action 4, and $[0, 0.71)$ to avoid conflict with action 5. The intersection of those intervals is used to annotate the action in the biclique as shown for action 2 in Figure 2(d). Avoiding execution of the action in the intersected unsafe interval ensures the mutually conflicting property (Walker, Sturtevant, and Felner 2020).

In the example, blocking action 2 in the intersected interval ($[0, 0.58) \cap [0, 0.71) = [0, 0.58)$) ensures that the interval for which action 2 is blocked conflicts with all other actions in the biclique, namely actions 4 and 5. If the interval $[0, 0.71)$ were erroneously chosen instead, action 2 would be blocked for part of the timeframe ($[.58, .71)$) in which it does not conflict with action 4, potentially blocking a feasible and/or optimal solution that uses action 4 in that timeframe. This technique is necessary for creating valid sets of time-range constraints required by the SIPP routine of CCBS.

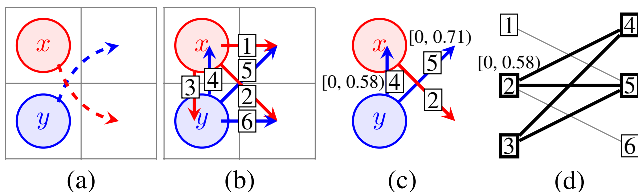

Figure 2: An example of bipartite conflict analysis. (a) A MAPF instance, (b) an enumeration of actions (wait actions omitted), (c) an illustration of unsafe intervals for action 2 and (d) a bipartite conflict graph and biclique.

New Biclique Constraints for CCBS

In this paper, we tested BC with CCBS for the first time. Although the use of max-vertex bicliques for BC was shown to be very effective in continuous-time domains with *fixed* wait actions (Walker, Sturtevant, and Felner 2020), when applied to CCBS, which computes *arbitrary* wait actions, we found that using max-vertex bicliques to generate biclique constraints was usually detrimental to performance. As noted earlier, taking the intersection of unsafe intervals of adjacent edges usually causes the interval to be shortened. Because of this shortened interval, the resulting safe intervals used with SIPP will not cause wait actions to be generated with a long enough duration to avoid conflict. This can result in causing a conflict between the two actions which caused the split (which we will call the *core action pair*) to recur at a slightly later time, resulting in another split in the sub-tree for the same two actions.

In order to remedy this, we computed the biclique so that it only includes the pairs of actions whose unsafe interval is a superset of that for the *core action pair*. For example, if the core action pair were actions 2 and 4, the biclique would include both actions 4 and 5, because its unsafe interval is a superset: $[0, 0.71) \supseteq [0, 0.58)$. On the other hand, if the core action pair were actions 2 and 5, action 4 could not be included in the set. In this way, the correct wait time is generated by SIPP and the same conflict is always avoided in the sub-tree. This approach of computing an *interval-superset biclique* often results in a smaller biclique and thus a smaller set of biclique constraints than the max-vertex biclique approach, nevertheless, the result yields performance gains versus CCBS with the original splitting method in many settings.

Biclique constraints in CBS are not usually applicable in "classic" planar graphs (assuming agents' size is sufficiently small) since agents would only conflict with one other action at a time (e.g., agents crossing the same edge in opposite directions), resulting in a 1x1 biclique which is equivalent to classic constraints. However, with CCBS, biclique constraints are useful in planar graphs since multiple wait actions are possible for the same agent at a vertex at a specific point in time, which makes multiple conflicts possible.

Disjoint Splitting with Biclique Constraints

Recall that with disjoint splitting, in one CT child node there is a positive constraint for agent i, and in the other there is a negative constraint for agent i. Additionally, (specifically in the formulation for CCBS), a single negative constraint for agent j is added to the node with the positive constraint. Doing this alone is quite effective (Andreychuk et al. 2021).

Algorithm 1 Disjoint Biclique Constraint Generation

1: Input:
2: A pair of conflicting actions: a_i, a_j.
3: Graph on which motion is planned: $G{=}(V, E)$.
4: Output:
5: Child node constraints: $N_i.C$, $N_j.C$.
6: ▷ Construct a Bipartite Conflict Graph:
7: $V_i \leftarrow \{(a_i.s.v\} \times N(a_i.s.v)\}$ ▷ Possible actions
8: $V_j \leftarrow \{(a_j.s.v\} \times N(a_j.s.v)\}$
9: $E_i \leftarrow \{\forall(v_i, v_j) \in \{v_i\} \times V_j |$ CONFLICT$(v_i, v_j)\}$
10: $E_j \leftarrow \{\forall(v_j, v_i) \in \{v_j\} \times V_i |$ CONFLICT$(v_j, v_i)\}$
11: **if** $|E_i| > |E_j|$ **then**
12: $Pos \leftarrow i$ ▷ Reference for positive constraint.
13: $Neg \leftarrow j$ ▷ Reference for negative constraint.
14: **else**
15: $Pos \leftarrow j$
16: $Neg \leftarrow i$
17: **end if**
18: ▷ Assign positive constraint.
19: $N_{Pos}.C \leftarrow \langle a_{Pos}$,UNSAFEINTERVAL$(a_{Pos}, a_{Neg})\rangle$
20: ▷ Assign neg. constraints as part of pos. constraint.
21: **for** $(v_a, v_b) \in E_{Pos}$ **do**
22: $N_{Pos}.C \leftarrow \langle v_a$,UNSAFEINTERVAL$(v_a, v_b)\rangle$
23: **end for**
24: ▷ Assign negative constraint.
25: $N_{Neg}.C \leftarrow \langle a_{Neg}$,UNSAFEINTERVAL$(a_{Neg}, a_{Pos})\rangle$

However, it is still possible for a conflict with the positively constrained action to recur in the sub-tree of the CT. We propose that when performing a disjoint split between agent i and agent j that along with a positive constraint for agent i, a *set* of negative constraints be added for agent j to the same CT node so that while agent i is forced to take an action, agent j is forced to avoid *all actions* which conflict with it.

To comprehensively enforce that agent j avoids the positively-constrained action for agent i (or vice-versa), we perform a modified form of bipartite analysis outlined in Algorithm 1. This procedure for computing time-annotated biclique constraints is simpler than outlined in the previous section because it is not necessary to test for the addition of edges for all pairs of vertices in the BCG and therefore the interval intersection and shortening steps are not necessary for the negative constraints. This is because the negative constraints must avoid the positive constraint's action for the entire duration. The input is the pair of conflicting actions, a_i and a_j and the graph G. The procedure starts by enumerating possible actions (via adjacent edges) that start at the start vertex of action a_i (resp. a_j) (see lines 7 and 8). Edges are then added for actions which conflict with the core action (lines 9 and 10). Our implementation chooses the positively constrained agent to be the one with the most nodes (line 11). However, alternative approaches could be taken such as choosing the set with the largest cumulative unsafe interval, the largest mean unsafe interval, etc. Finally, one positive constraint is added with the set of negative constraints for conflicting actions for the positively-constrained agent (lines 18-23), and one negative constraint for the negatively-constrained agent (line 25). We make sure to compute the proper unsafe interval for each constraint, using the UNSAFEINTERVAL routine.

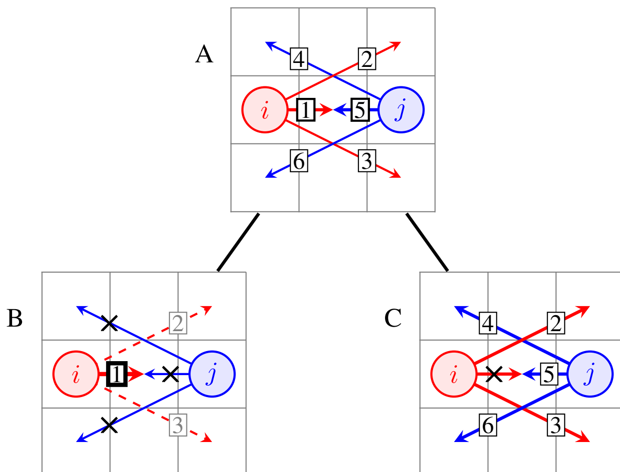

Figure 3: A disjoint split with biclique constraints.

The placement of constraints is illustrated in Figure 3. Node A is a node in the CT with a conflict between actions 1 and 5 shown in bold; this is the core action pair. Nodes B and C are child nodes. Node B shows the positive constraint for the red agent in bold for action 1, with negative constraints for the blue agent's conflicting actions shown with 'x's. The other actions for the red agent in node B are dashed, meaning that they are no longer reachable because of the positive constraint. Finally, node C shows a single negative constraint that mirrors the positive constraint in node B.

In summary, while regular disjoint splitting would only create a positive constraint for agent i with a single negative constraint for agent j to child node 1 and a negative constraint for agent i to child node 2, disjoint splitting with bicliques, or *disjoint bicliques* (DB) adds multiple negative constraints for agent j to child node 1 as well. The effect of adding the extra constraints for agent j is that agent j avoids the positively-constrained action for agent i from multiple paths, preemptively avoiding conflicts further down in the CT. Ultimately, a significant number of nodes are pruned from the CT.

We now show that this approach is complete (if a solution to the problem instance exists) and that it also ensures optimality.

Lemma 1. *Using biclique constraints with disjoint splitting (disjoint bicliques) never blocks a feasible solution from being found by CCBS.*

Proof. Let N be a CT node. Let a_i and a_j be a core action pair – a conflicting pair of actions from $\pi_i \in N.\Pi$ and $\pi_j \in N.\Pi$ respectively.

Let $\bar{N}_i$ be a child of N which contains a single negative constraint blocking action a_i. Let $\hat{N}_i$ be a second child of N with a single positive constraint forcing agent i to perform action a_i, and multiple negative constraints $\bar{C}_j$, blocking agent j from conflicting with a_i.

Let Π^* be the only feasible solution to the MAPF problem instance. There are three possible cases:

1. Π^* contains a_i
2. Π^* contains a_j
3. Π^* contains neither a_i nor a_j

If case 1 is true, Π^* is guaranteed to be found in the sub-tree of $\hat{N}_i$ because a_i is enforced by the positive constraint. If case 2 is true, Π^* is guaranteed to be found in the sub-tree of $\bar{N}_i$ because a_j is not blocked. If case 3 is true, Π^* is guaranteed to be found in the sub-tree of $\bar{N}_i$ because a_i is blocked and a_j is not enforced.

Because the BCG only includes actions which conflict with a_i, it is not possible to block any action that does not conflict with a_i. Since in case 1, Π^* cannot contain any actions that conflict with a_i, blocking all actions for agent j which conflict with a_i cannot preclude Π^*. Thus case 1 still holds with disjoint bicliques. □

Theorem 2. *CCBS with disjoint bicliques is optimal.*

Proof. Per Lemma 1, using disjoint bicliques can never preclude CCBS from finding a solution (if any exist). Since the OPEN list is ordered by lowest cost, CCBS is guaranteed to find a lowest-cost feasible solution before terminating. □

Disjoint K-Partite Cliques

So far, we have discussed the approach for combining bicliques with disjoint splitting for resolving a single conflict. It is often the case that multiple agents conflict with the positively constrained action. It is possible (and helpful) to additionally constrain these agents using negative constraints. This is done by executing Algorithm 1 for all agents that have a conflict with the positively constrained action to form two k-partite conflict graphs (KCG), one for agent i and another for agent j. But it is explained more simply as enumerating all actions by all agents which conflict with a_i and a_j respectively and computing unsafe intervals. All other steps are straightforward, and amount to assigning adding negative constraints for all actions that conflict with the positively constrained one. Our implementation chooses the agent with the largest KCG to get the positive constraint.

In order to avoid performing extra conflict checks to discover other agents that conflict with the positively-constrained action, we make use of a conflict count table (CCT) (Walker 2022) which is adapted for MAPF_R to perform bookkeeping on all conflicts. The CCT is set up so that looking up conflicts in the table is indexed on a per-agent, per-action basis to expedite the KCG creation. In a vast majority of cases, the number of partitions in the KCG graphs are much smaller than the number of agents.

We show that DK is correct by an extension of Lemma 1:

Lemma 3. *The use of disjoint k-partite cliques with disjoint splitting never blocks a feasible solution in CCBS.*

Proof. Following the definitions from Lemma 1, DK now adds negative constraints for multiple agents to $\hat{N}_i$. Since the constraints in $\bar{N}_i$ are unchanged, cases 2 and 3 still hold.

Case 1 still holds because Π^* cannot contain any action by any agent which conflicts with a_i, therefore just as the disjoint bicliques procedure ensures that only actions which conflict with a_i are blocked for agent j, DK ensures that only actions which conflict with a_i are blocked for all agents $j \neq i$ (or subset of agents $\in j \neq i$). Thus, DK can never block a feasible solution in any of the three cases. □

Finally, substituting Lemma 3 into Theorem 2, we see that DK is also optimal. In summary, we now have a powerful capability for multi-agent conflict resolution, capable of eliminating even more nodes in the CT.

Empirical Results

We now analyze the enhancements described in the previous section, namely: bypass, which is new to CCBS in this paper, biclique constraints (BC) which is newly formulated in this paper, and disjoint k-partite cliques (DK) which is new. All tests in this section were performed single-threaded, on cloud compute instances that report an Intel Xeon 2.5GHz processor. In addition to the three roadmaps: "Sparse", "Dense" and "Super-dense" and the four grid maps focused on by the original CCBS authors, we test our enhancements in 32 of the MAPF grid benchmarks (Stern et al. 2019) with 2^k neighborhood (Rivera, Hernández, and Baier 2017) connectivities, namely 4-, 8-, 16- and 32-neighborhoods. In accordance with previous benchmark settings, agents are circular, with a radius of $\sqrt{2}/4$, however, any agent radius can be used with this algorithm.

All tests were run by starting with 2 agents and incrementing the number of agents by 2 until the problem instance became unsolvable in under 30 seconds. We thank the original CCBS authors for making their code publicly available. Our implementation is based on theirs and is also freely available[1]. We updated some of the memory management, which made it slightly faster than the original. For this reason, some of our baseline results differ from previously published ones.

We will describe each of our experiments in turn and then discuss them all together. Figure 4 shows the success rates; the percentage of problem instances solvable in under 30 seconds for increasing numbers of agents. The rates are computed over 25 problem instances for each map. All plots, except for the super-dense roadmap are on 32-neighbor grids. We remind the reader that results are optimal in terms of cost (i.e., shortest non-conflicting paths), and that adding a single agent to a problem instance represents a multiplicative increase in the problem instance's computational complexity. More precisely, the upper bound on the computational complexity of MAPF (and by extension MAPF_R) is exponential in the number of agents (Yu and LaValle 2013), therefore, adding one agent to a problem instance increases the exponent of the upper bound by one. Hence, even a small gain (e.g., a few percentage points) can mean that a large reduction in work is actually realized.

Tables 1, 2 and 3 show the sum total of the maximum number of agents solvable over 25 problem instances in under 30 seconds per map. The statistics with the best result

[1]https://github.com/thaynewalker/CCBS

underlined and those within the 95th percentile of the best are in bold. The label "CCBS" in Figure 4 is CCBS with all enhancements from the original authors except DS. The label "Base" is CCBS with all enhancements, including DS, the previous state-of-the-art. In all of the tables, the column labeled "Base" is also the previous state-of-the-art.

Table 1 shows totals for all grid maps for 4-neighbor grids, totals for each class of maps, and an overall total. 4-neighbor grids in this context are similar to those for "classic" MAPF, except instead of fixed wait actions, agents may wait an arbitrary amount of time. Table 2 shows aggregate group results for the same groupings as Table 1, and the grand total for all remaining connectivity settings, namely 8-, 16- and 32-connected grids. Table 3 shows the results for all settings on probabilistic roadmaps (Andreychuk et al. 2019). "sparse" contains 158 nodes and 349 edges with a mean vertex degree of 4.2, "dense" contains 878 nodes and 7,341 edges with a mean degree of 16.7, and "super-dense" contains 11,342 vertices and 263,533 edges with a mean degree of 100.4.

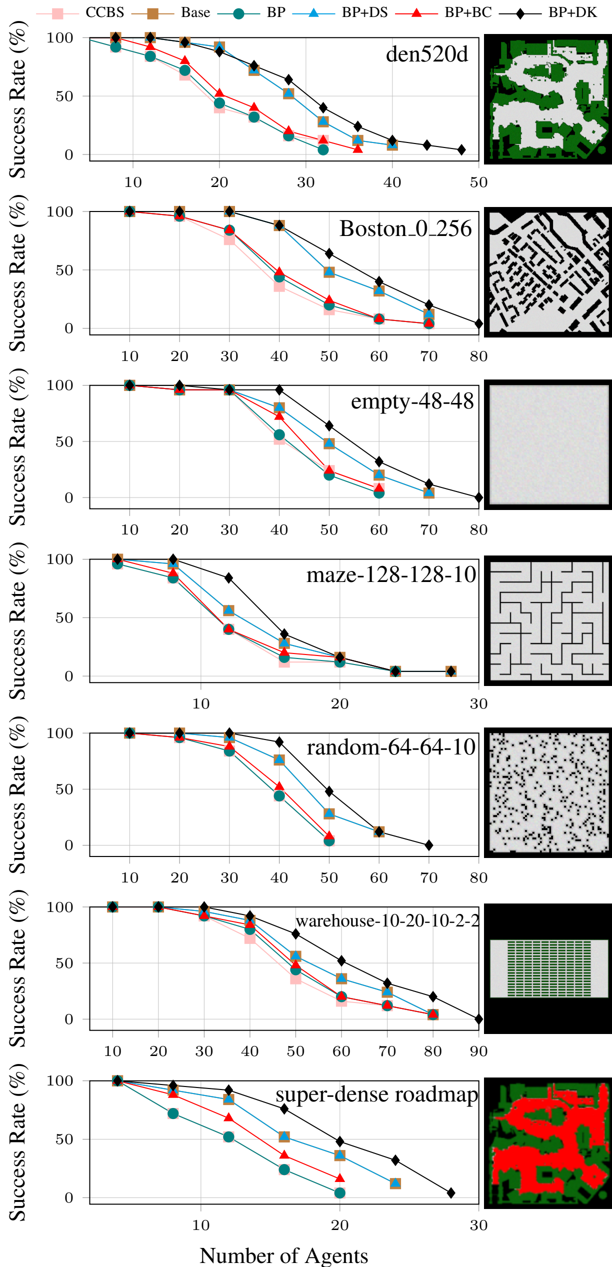

Figure 4: Success rates on 32-neighbor grids and roadmaps.

Map	Base	BP+DS	DK	BP+DK
Berlin_1_256	1,200	**1,334**	1,200	**1,334**
Boston_0_256	**1,112**	**1,140**	**1,112**	**1,152**
Paris_1_256	1,432	**1,524**	1,434	**1,524**
City Total	3,744	**3,998**	3,746	**4,010**
den520d	**844**	**860**	**844**	**868**
brc202d	580	**604**	586	**606**
den312d	474	**510**	474	**552**
lak303d	474	**538**	476	**552**
orz900d	**626**	566	**622**	564
ost003d	**562**	**582**	**564**	**582**
DAO Total	3,560	**3,660**	3,566	**3,724**
empty-8-8	312	316	314	**338**
empty-16-16	**520**	**520**	**520**	**546**
empty-32-32	886	**920**	886	**918**
empty-48-48	1,058	**1,110**	1,058	**1,144**
Empty Total	**2,776**	**2,866**	**2,778**	**2,946**
lt_gallowstemplar	**632**	**654**	**634**	**658**
ht_chantry	**570**	**588**	566	**592**
ht_mansion_n	680	**726**	680	**746**
w_woundedcoast	664	**698**	670	**706**
DAO2 Total	2,546	**2,666**	2,550	**2,702**
maze-32-32-2	**280**	**278**	**286**	**278**
maze-32-32-4	252	**256**	248	254
maze-128-128-2	**248**	**254**	**252**	**256**
maze-128-128-10	340	**364**	340	**360**
Maze Total	1,116	**1,152**	**1,126**	**1,148**
random-64-64-10	**1,210**	**1,222**	**1,210**	**1,266**
random-64-64-20	**862**	**860**	**852**	**920**
random-32-32-10	614	**674**	614	**674**
random-32-32-20	424	**438**	424	**448**
Random Total	**3,110**	**3,194**	**3,100**	**3,308**
room-64-64-16	**424**	**432**	**424**	**428**
room-64-64-8	290	288	290	**314**
room-32-32-4	286	**298**	286	**306**
Room Total	**1,000**	**1,018**	**1,000**	**1,048**
w-10-20-10-2-2	1,124	**1,372**	1,124	**1,390**
w-10-20-10-2-1	1,022	**1,078**	1,022	**1,066**
w-20-40-10-2-2	1,890	**1,998**	1,890	**1,998**
w-20-40-10-2-1	1,740	1,870	1,870	**2,038**
Warehouse Total	5,776	**6,318**	5,776	**6,492**
Total	23,628	**24,872**	23,642	**25,378**

Table 1: Sum total of agents solved on 4-neighbor grid MAPF benchmarks

Map Type	Base	BP+DS	DK	BP+DK
		8-neighbor grids		
City	4,820	**5,016**	4,982	**5,084**
DAO	4,298	**4,394**	**4,382**	**4,474**
Empty	3,304	**3,560**	3,396	**3,628**
DAO2	3,174	**3,318**	3,242	**3,380**
Maze	**1,304**	**1,292**	**1,298**	**1,336**
Random	3,486	**3,628**	3,514	**3,760**
Room	**1,142**	**1,148**	**1,124**	**1,168**
Warehouse	6,688	**6,990**	**6,722**	**7,280**
Total	28,216	**29,346**	28,660	**30,110**
		16-neighbor grids		
City	3,906	**4,044**	3,958	**4,148**
DAO	3,578	3,616	3,602	**3,800**
Empty	3,252	**3,326**	3,278	**3,366**
DAO2	2,598	**2,662**	**2,624**	**2,776**
Maze	**1,142**	**1,154**	**1,146**	**1,188**
Random	3,080	**3,124**	**3,110**	**3,260**
Room	**1,044**	**1,058**	**1,040**	**1,082**
Warehouse	6,516	**6,650**	6,550	**6,926**
Total	25,116	**25,634**	25,308	**26,546**
		32-neighbor grids		
City	3,310	3,358	3,414	**3,680**
DAO	2,860	2,884	**2,940**	**3,058**
Empty	2,798	2,058	2,872	**3,116**
DAO2	2,190	2,218	2,224	**2,342**
Maze	1,030	1,028	1,060	**1,124**
Random	2,840	2,866	2,940	**3,186**
Room	1,032	1,042	1,040	**1,106**
Warehouse	5,832	6,128	5,926	**6,478**
Total	21,892	21,582	22,416	**24,090**

Table 2: Sum total of agents solved on 8-, 16- and 32-neighbor grid MAPF benchmarks

Map	Base	BP+DS	DK	BP+DK
Sparse	**434**	**434**	**440**	**444**
Dense	604	604	630	**712**
Super-dense	402	402	442	**474**
Total	1,440	1,440	1,512	**1,630**

Table 3: Sum total of agents solved on roadmaps

Discussion

In Figure 4, the BP enhancement (teal circle) success rate is not significantly better or worse than CCBS alone (p square). It never performs worse than CCBS in any of the 32-neighbor maps. This is also corroborated by the results for the roadmaps in Table 3, BP does not improve performance in these cases. On the other hand, in Table 1, which is on 4-neighbor maps, we see that adding BP to Base (previous state-of-the-art) yields statistically significant gains in nearly every map. Table 2 shows a similar trend for all 8-neighbor grids, but as the connectivity is increased to 16- and 32-neighbor grids, the improvement from BP becomes less significant. The cost symmetries in these settings decreases as the connectivity increases. Ultimately, in the road maps, which have practically no cost symmetries, we see no improvement with BP. The results also show that there is no significant decline in performance in the higher connectivities when using BP. From this we learn that (1) BP offers significant performance improvements when there are many cost symmetries in the graph, and the performance benefits are directly proportional to the amount of cost symmetries. (2) The cost of BP is not significant, even with no symmetries in the map. (3) BP is complimentary to DS and BC.

The trend in Figure 4 shows that BC provides a consistent improvement over CCBS in 32-neighbor grids. It is complimentary to DS, as evidenced by the fact that BP+DK performs better than both BP+DS and BP+BC. The trend in all tables shows that the performance improvement of DK is correlated to the mean vertex degree in the graph. The amount of improvement is especially significant in the dense and super-dense road maps, where BP offers no benefit. In these roadmaps, the branching factor is high, with many edge crossings, a situation which is conducive to large bicliques. Still, the cost of DK is not significant in planar graphs. From this we learn that (1) DK offers significant performance improvements when many edges cross in the graph. (2) BP and DK are complimentary, as evidenced by BP being stronger in settings with many cost symmetries and DK being stronger in settings with few cost symmetries. With CCBS, because an agent may have multiple different wait actions at a location, BCGs larger than 1x1 are possible, thus a small benefit over Base is shown with BC in Table 1.

Because the BP enhancement is never significantly detrimental to performance, and provides a benefit even when there are few cost symmetries, it can be used generally. The DK enhancement tends to work best in many cases where BP does not, and it also has no significant execution cost, hence it can be used generally.

Finally, combining BP with DK consistently beats state-of-the-art by statistically significant margins. Compared to the previous state-of-the-art, our enhancements allow solutions for up to 10% more agents and in grid maps and for up to 20% more agents in super-dense roadmaps in the same amount of time.

Conclusion

We have tested re-formulated enhancements with CCBS, namely: Bypassing (BP) and Biclique Constraints (BC). We have formulated novel enhancements for CCBS, namely: Disjoint Bicliques (DB) and Disjoint K-Partite Cliques (DK), leading to a new state-of-the-art algorithm: CCBS+DK. We found BC alone to be less effective. We found BP to be most effective in graphs with many cost symmetries. We found DB and DK to be most effective in graphs with few cost symmetries and densely crossing edges. We have shown that BP and DK are complimentary and that using them together allows a statistically significant improvement over state-of-the-art generally for all MAPF benchmarks and sparse to super dense graphs.

Acknowledgments

The research at the University of Denver was supported by the National Science Foundation (NSF) grant number 1815660 and Lockheed Martin Corporation. Research at the University of Alberta was funded by the Canada CIFAR AI Chairs Program. We also acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Research at Ben Gurion University was supported by BSF grant number 2021643.

References

Andreychuk, A.; Yakovlev, K.; Atzmon, D.; and Stern, R. 2019. Multi-Agent Pathfinding with Continuous Time. In *International Joint Conference on Artificial Intelligence*, 39–45.

Andreychuk, A.; Yakovlev, K.; Boyarski, E.; and Stern, R. 2021. Improving continuous-time conflict based search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 11220–11227.

Andreychuk, A.; Yakovlev, K.; Surynek, P.; Atzmon, D.; and Stern, R. 2022. Multi-agent pathfinding with continuous time. *Artificial Intelligence*, 305.

Atzmon, D.; Stern, R.; Felner, A.; Wagner, G.; Barták, R.; and Zhou, N.-F. 2018. Robust Multi-Agent Path Finding. In *International Conference on Autonomous Agents and Multiagent Systems*, 1862–1864.

Botea, A.; Bouzy, B.; Buro, M.; Bauckhage, C.; and Nau, D. 2013. Pathfinding in games. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Boyarski, E.; Felner, A.; Sharon, G.; and Stern, R. 2015a. Don't Split, Try To Work It Out: Bypassing Conflicts in Multi-Agent Pathfinding. In *International Conference on Automated Planning and Scheduling*, 47–51.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Betzalel, O.; and Shimony, S. E. 2015b. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In *International Joint Conference on Artificial Intelligence*, 223–225.

Choudhury, S.; Solovey, K.; Kochenderfer, M. J.; and Pavone, M. 2021. Efficient large-scale multi-drone delivery using transit networks. *Journal of Artificial Intelligence Research*, 70: 757–788.

Kottinger, J.; Almagor, S.; and Lahijanian, M. 2022. Conflict-Based Search for Multi-Robot Motion Planning with Kinodynamic Constraints. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 13494–13499. IEEE.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019a. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In *IJCAI*, volume 2019, 442–449.

Li, J.; Harabor, D.; Stuckey, P. J.; Felner, A.; Ma, H.; and Koenig, S. 2019b. Disjoint Splitting for Multi-Agent Path Finding with Conflict-Based Search. In *International Conference on Automated Planning and Scheduling*, 279–283.

Li, J.; Harabor, D.; Stuckey, P. J.; Ma, H.; Gange, G.; and Koenig, S. 2021a. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence*, 103574.

Li, J.; Surynek, P.; Felner, A.; Ma, H.; and Kumar, T. K. S. 2019c. Multi-Agent Pathfinding for Large Agents. In *AAAI Conference on Artificial Intelligence*, 7627–7634.

Li, J.; Tinka, A.; Kiesel, S.; Durham, J. W.; Kumar, T. S.; and Koenig, S. 2021b. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 11272–11281.

Phillips, M.; and Likhachev, M. 2011. Sipp: Safe interval path planning for dynamic environments. In *International Conference on Robotics and Automation*, 5628–5635. IEEE.

Rivera, N.; Hernández, C.; and Baier, J. A. 2017. Grid Pathfinding on the 2k Neighborhoods. In *AAAI Conference on Artificial Intelligence*, 891–897.

Roldán-Gómez, J. J.; González-Gironda, E.; and Barrientos, A. 2021. A survey on robotic technologies for forest firefighting: Applying drone swarms to improve firefighters' efficiency and safety. *Applied Sciences*, 11(1): 363.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence Journal*, 219: 40–66.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Kumar, T. K. S.; Boyarski, E.; and Barták, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In *International Symposium on Combinatorial Search*, 151–159.

Walker, T. T. 2022. *Multi-Agent Pathfinding in Mixed Discrete-Continuous Time and Space*. Ph.D. thesis, University of Denver. Language: English.

Walker, T. T.; and Sturtevant, N. R. 2019. Collision Detection for Agents in Multi-Agent Pathfinding. *arXiv preprint arXiv:1908.09707*.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2018. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. In *International Joint Conference on Artificial Intelligence*, 534–540.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2020. Generalized and Sub-Optimal Bipartite Constraints for Conflict-Based Search. In *AAAI Conference on Artificial Intelligence*.

Wen, L.; Liu, Y.; and Li, H. 2022. CL-MAPF: Multi-agent path finding for car-like robots with kinematic and spatiotemporal constraints. *Robotics and Autonomous Systems*, 150: 103997.

Yakovlev, K.; and Andreychuk, A. 2017. Any-Angle Pathfinding for Multiple Agents Based on SIPP Algorithm. *arXiv preprint arXiv:1703.04159*.

Yu, J.; and LaValle, S. M. 2013. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In *AAAI Conference on Artificial Intelligence*, 1443–1449.

Zhang, H.; Li, J.; Surynek, P.; Koenig, S.; and Kumar, T. K. S. 2020. Multi-Agent Pathfinding with Mutex Propagation. In *International Conference on Automated Planning and Scheduling*.