Improving Continuous-time Conflict Based Search*

Anton Andreychuk, 1, 2 Konstantin Yakovlev, 2, 3 Eli Boyarski, 4 Roni Stern 4, 5

1 Peoples’ Friendship University of Russia (RUDN University)
2 Federal Research Center for Computer Science and Control of Russian Academy of Sciences
3 HSE University 4 Ben-Gurion University of the Negev
5 Palo Alto Research Center

andreychuk@mail.com, yakovlev@isa.ru, eli@boyar.ski, sternron@post.bgu.ac.il

Abstract
Conflict-Based Search (CBS) is a powerful algorithmic framework for optimally solving classical multi-agent path finding (MAPF) problems, where time is discretized into the time steps. Continuous-time CBS (CCBS) is a recently proposed version of CBS that guarantees optimal solutions without the need to discretize time. However, the scalability of CCBS is limited because it does not include any known improvements of CBS. In this paper, we begin to close this gap and explore how to adapt successful CBS improvements, namely, prioritizing conflicts (PC), disjoint splitting (DS), and high-level heuristics, to the continuous time setting of CCBS. These adaptions are not trivial, and require careful handling of different types of constraints, applying a generalized version of the Safe interval path planning (SIPP) algorithm, and extending the notion of cardinal conflicts. We evaluate the effect of the suggested enhancements by running experiments both on general graphs and \(2^2\)-neighborhood grids. CCBS with these improvements significantly outperforms vanilla CCBS, solving problems with almost twice as many agents in some cases and pushing the limits of multi-agent path finding in continuous-time domains.

Introduction
Multi-Agent Pathfinding (MAPF) is the problem of finding paths for \(n\) agents in a graph such that each agent reaches its desired goal vertex and the agents do not collide with each other while moving along these paths. Many real-world applications require solving variants of MAPF, including managing aircraft-towing vehicles (Morris et al. 2016), video game characters (Silver 2005), office robots (Veloso et al. 2015), and warehouse robots (Wurman, D’Andrea, and Mountz 2007). Finding an optimal solution to a MAPF problem for common objective functions is NP-Hard (Surynek 2010; Yu and LaValle 2013), but modern optimal MAPF algorithms can scale to problems with over a hundred agents (Sharon et al. 2015; Boyarski et al. 2015; Felner et al. 2018; Lam et al. 2019; Gange, Harabor, and Stuckey 2019; Surynek et al. 2016).

However, such scaling has been demonstrated mostly on the classical version of the MAPF problem (Stern et al. 2019), which embodies several simplifying assumptions such as all actions have the same duration and time is discretized into time steps. MAPF\(_{R}\) (Walker, Sturtevant, and Felner 2018) is a generalization of the classical MAPF problem in which actions’ durations can be non-uniform, agents have geometric shapes that must be considered, and time is continuous, that is, it is not discretized into time steps. Handling continuous time in MAPF\(_{R}\) is challenging because it implies an agent may wait in a location for an arbitrary amount of time, i.e., the number of wait actions is infinite.

Several recently proposed algorithms address the MAPF\(_{R}\) problem or limited variants of it, such as Extended ICTS (E-ICTS) (Walker, Sturtevant, and Felner 2018), CBS with Continuous Time-steps (CBS-CT) (Cohen et al. 2019), and Continuous-time conflict-based search (CCBS) (Andreychuk et al. 2019). A direct comparison between these algorithms is problematic as they make different underlying assumptions.

In this work, we propose several improvements to CCBS that allow it to solve MAPF\(_{R}\) problems with significantly more agents. CCBS is based on the Conflict-based search (CBS) algorithm for classical MAPF, and the improvements we propose for CCBS are based on known improvements of CBS, namely Disjoint Splitting (DS), Prioritizing Conflicts (PC), and high-level heuristics. Adapting the DS technique to the continuous-time setting of MAPF\(_{R}\) requires solving a single-agent pathfinding problem with temporally-constrained action landmarks (Karpas and Domshlak 2009). We show how to efficiently solve this pathfinding problem in our context by applying a generalized version of the SIPP algorithm (Phillips and Likhachev 2011). A naive applying of PC to CCBS is shown to be ineffective, and we propose an adapted version of PC that can cut the number of expanded nodes significantly. The third CCBS improvement we propose is an admissible heuristic function for CCBS that require only a negligible amount of overhead when applied together with the PC technique.

Finally, we evaluate the impact of these improvements individually and collectively on several benchmarks, including both roadmaps and grids. The results show that the number of MAPF\(_{R}\) instances solved by CCBS with all the proposed improvements compared to vanilla CCBS has increased by 49.2% — from 3,792 to 5,659. In some cases, it can even solve problems with approximately twice the number of agents compared to vanilla CCBS and reduce the runtime up to two orders of magnitude.

*This is a pre-print of the paper accepted to AAAI 2021.
Background and Problem Statement

In a MAPF$_R$ problem [Walker, Sturtevant, and Felner 2018], the agents are confined to a weighted graph $G = (V,E)$ whose vertices $V$ correspond to locations in some metric space, e.g. $\mathbb{R}^2$ in a Euclidean space, and edges $(E)$ correspond to possible transitions between these location. Each agent $i$ is initially located at vertex $s_i \in V$ and aims to reach vertex $g_i \in V$. When at a vertex, an agent can either perform a move action or a wait action. A move action means moving the agent along an edge. We assume that the agent moves in a constant velocity and inertial effects are neglected. The duration of a move action is the weight of its respective edge.

A wait action means the agents stays in its current location. The agent along an edge. We assume that the agent moves in a constant velocity and inertial effects are neglected. The duration of a wait action can be any positive real value. Since we do not discretize time, the set of possible wait actions is uncountable.

A timed action is a pair $(a_i,t_i)$ representing that action $a_i$ (either move or wait) starts at time $t_i$. A plan for an agent is a sequence of timed actions such that executing this sequence of timed actions moves the agent from its initial location to its goal location. The cost of a plan is the sum of the durations of its constituent actions. We assume that after finishing the plan the agent does not disappear but rather stays at the last vertex forever, but this “dummy” wait action does not add up to the cost of the plan.

The plans of two agents are said to be conflict free if the agents following them never collide. A joint plan is a set of plans, one per each agent. A solution to a MAPF$_R$ problem is joint plan whose constituent plans are pairwise conflict-free. The cost of a solution is its sum of costs (SOC), i.e., the sum of costs of its constituent plans. In this work, we are interested in solving MAPF$_R$ problems optimally, i.e., finding a solution with a minimal cost.

CBS (Andreychuk et al. 2019) is a CBS-based algorithm that does so. For completeness, we provide a brief description of CBS and CCBS below.

Conflict-based search (CBS)

CBS [Sharon et al. 2015] is a complete and optimal algorithm for solving classical MAPF problems, i.e., MAPF problems where time is discretized and all actions have the same duration. CBS works by finding plans for each agent separately, detecting conflicts between these plans, and resolving them by replanning for the individual agents subject to specific constraints. A CBS conflict in CBS is defined by a tuple $(i,j,x,t)$ stating that agents $i$ and $j$ have a conflict in location $x$ (either a vertex or an edge) at time $t$. A CBS constraint is defined by a tuple $(i,x,t)$, which states that agent $i$ cannot occupy $x$ at time $t$. To resolve a conflict $(i,j,x,t)$, CBS replans for agent $i$ or $j$ or both, subject to CBS constraints $(i,x,t)$ and $(j,x,t)$, respectively. To guarantee completeness and optimality, CBS runs two search algorithms: a low-level search algorithm that finds paths for individual agents subject to a given set of constraints, and a high-level search algorithm that chooses which constraints to impose and which conflicts to resolve.

CBS: Low-Level Search.

In the basic CBS implementation, the low-level search is a search in the state space of vertex-time pairs. Expanding a state $(v,t)$ generates states of the form $(v',t+1)$, where $v'$ is either equal to $v$, representing a wait action, or equal to one of the locations adjacent to $v$. States generated by actions that violate the given set of CBS constraints, are pruned. CBS runs $A^*$ on this search space to return the lowest-cost path to the agent’s goal that is consistent with the given set of CBS constraints, as required.

CBS: High-Level Search.

The CBS high-level search is a search in a binary tree called the Constraint Tree (CT). In the CT, each node $N$ represents a set of CBS constraints $N$.constraints and a joint plan $N$.II that is consistent with these constraints. Generating a node $N$ involves settings its constraints $N$.constraints and running the low-level search to create $N$.II. If $N$.II does not contain any conflict, then $N$ is a goal. Expanding a non-goal node $N$ involves choosing a conflict $(i,j,x,t)$ in $N$.II and generating two child nodes $N_i$ and $N_j$. Both nodes have the same set of constraints as $N$, plus a new CBS constraint: $(i,x,t)$ for $N_i$ and $(j,x,t)$ for $N_j$. This type of node expansion is referred to as splitting node $N$ over conflict $(i,j,x,t)$. The high-level search finds a goal node by searching the CT in a best-first manner, expanding in every iteration the CT node $N$ with the lowest-cost joint plan.

Continuous-Time Conflict Based Search (CCBS)

To consider continuous time, CCBS (Andreychuk et al. 2019) reasons over the time intervals, detects conflicts between timed actions, and resolves conflicts by imposing constraints that specify the time intervals in which the conflicting timed actions can be moved to avoid the conflict. Formally, a CCBS conflict is a tuple $(a_i,t_i,a_j,t_j)$, specifying that the timed action $(a_i,t_i)$ of agent $i$ has a conflict with the timed action $(a_j,t_j)$ of agent $j$. The unsafe interval of timed action $(a_i,t_i)$ w.r.t. the timed action $(a_j,t_j)$, denoted $[t_i,t_i']$, is the maximal time interval starting from $t_i$ in which performing $a_i$ creates a conflict with performing $a_j$ at time $t_j$. A CCBS constraint is a tuple $(i,a_i,[t_i,t_i'])$ specifying that agent $i$ cannot perform action $a_i$ in the time interval $[t_i,t_i']$. To resolve a CCBS conflict, CCBS generates two new CT nodes, where it adds the constraint $(i,a_i,[t_i,t_i'])$ to one node and the constraint $(j,a_j,[t_j,t_j'])$ to the other.

The low-level planner of CCBS is an adaptation of the SIPP algorithm [Phillips and Likhachev 2011]. SIPP was originally designed to find time-optimal paths for an agent moving among the dynamic obstacles with known trajectories. SIPP runs a heuristic search in the state-space of $(v,[t,t'])$ tuples, where $v$ is the graph vertex and $[t,t']$ is a safe interval of $v$, i.e. a maximal contiguous time interval in which an agent can stay or arrive at $v$ without colliding with a moving obstacle. As numerous obstacles may pass through $v$ there can exist numerous search nodes corresponding to the same graph vertex but different time intervals in the SIPP search tree.

The CCBS low-level search is based on SIPP except for how it handles the given CCBS constraints. Instead of dy-

---

1This assumption is common in the MAPF literature.
Dynamics obstacles, the low-level CCBS computes safe intervals for each vertex \( v \) with respect to the CCBS constraints imposed over wait actions at \( v \). Initially, vertex \( v \) has a single safe interval \([0, \infty)\). Then, for every CCBS constraint \((i, a_i, [t_i, t_i])\) where \( a_i \) is a wait action at vertex \( v \), we split the safe interval for \( v \) to arriving before \( t_i \) and arriving after \( t_i \). CCBS constraints imposed over the move actions are integrated into the low-level search by modifying the constrained actions, as follows. Let \( v \) and \( v' \) be the source and target destinations of \( a_i \). If the agent arrives to \( v \) at \( t \in [t_i, t_i) \) then we remove the action that moves it from \( v \) to \( v' \) at time \( t \), and add an action that represents waiting at \( v \) until \( t_i \) and then moving to \( v' \).

Disjoint Splitting for CCBS

The first technique we migrate from CBS to CCBS is called Disjoint Splitting (DS) (Li et al. 2019b). DS is a technique designed to ensure that expanding a CT node \( N \) creates a disjoint partition of the space of solutions that satisfy the constraints in \( N.constraints \). That is, every solution that satisfies \( N.constraints \) is in exactly one of its children. Observe that this is not the case in CBS: for a conflict \((i, j, v, t)\) there may be solutions that satisfy both \((i, v, t)\) and \((j, v, t)\). This introduces an inefficiency in the high-level search.

To address this inefficiency, CBS with DS (CBS-DS) introduces the notion of positive and negative constraints. A negative constraint \((i, x, k)\) is the regular CBS constraint stating that agent \( i \) must not be at \( x \) at time step \( k \). A positive constraint \((i, x, k)\) means that agent \( i \) must be at \( x \) at time step \( k \). When splitting a CT node \( N \) over a CBS conflict \((i, j, x, k)\), CBS-DS chooses one of the conflicting agents, say \( i \), and generates two child nodes, one with the negative constraint \((i, x, k)\) and the other with the positive constraint \((i, x, k)\). Deciding on which agent, either \( i \) or \( j \) to split on, does not affect the theoretical properties of the algorithm, and several heuristics were proposed (Li et al. 2019b). The low-level search in CBS-DS treats each positive constraint as a special type of fact landmark (Richter, Helmert, and Westphal 2008), i.e., a fact that must be true in any plan.

The CBS-DS low-level search generates a plan that satisfies these fact landmarks by planning to achieve these fact landmarks in ascending order of their time dimension. This effectively decomposes the low-level search to a sequence of simpler search tasks, searching for path one fact landmark to the next one. The agent’s goal is set a the last fact landmark, to ensure the agent reaches it eventually.

Positive and Negative Constraints in CCBS

A CCBS constraint \((i, a_i, [t_i, t_i])\) can be stated formally as follows:

\[
\forall t \in [t_i, t_i) : (a_i, t) \text{ is not in a plan for agent } i
\]

This is a negative constraint from a DS perspective. The corresponding positive constraint is therefore the inverse:

\[
\exists t \in [t_i, t_i) : (a_i, t) \text{ is in a plan for agent } i
\]

In words, this mean agent \( i \) must perform \( a_i \) at some moment of time from the given interval. Since this constraint is over an action, the positive constraint in CCBS is action landmark (Karpas and Domshlak 2009), i.e., the action that must be performed in any solution. Next, we show how the low level search of Continuous-time conflict-based search with disjoint splitting (CCBS-DS) is able to find a plan that achieves all these action landmarks efficiently.

Low-Level Search in CCBS

The low-level search in CCBS-DS sorts the positive constraints in ascending order of their time dimension and plan to achieve each of them in that order. For example, assume there is a single positive constraint \((i, \text{move}(A, B), [t_i, t_i])\).

Then, the low-level search works by first (1) searching for a plan from \( s_i \) to \( A \) that ends in the time range \([t_i, t_i]\), then (2) performing the action landmark (i.e., move from \( A \) to \( B \)), and finally (3) searching for a plan from \( B \) to \( y_t \), (starting immediately after the action landmark is performed).

However, in CCBS-DS there is an additional challenge for the low-level search: there may be more than one plan for to perform each landmark. In our example above, there may be an infinite amount of plans from \( s_i \) to \( A \) that ends in the time range \([t_i, t_i]\). As we show below, choosing the plan that performs the action landmark earliest does not necessarily lead to finding an optimal solution and even lead to incompleteness, especially when there are both positive and negative constraints.

Example Consider the illustration depicted in Figure 1

The low-level search needs to find a plan that satisfies three constraints:

- A positive constraint \((i, \text{move}(A, B), [t_{ab}, t_{ab}])\).
- A negative constraint \((i, \text{wait}(A), [t_{aa}, t_{aa}])\).
- A negative constraint \((i, \text{wait}(B), [t_{bb}, t_{bb}])\).

where \( t_{ab} < t_{aa} < t_{ab} < t_{ab} \). Thus, the negative constraint on waiting at \( A \) (\( \text{wait}(A) \)) creates two safe intervals for \( A \), \( I_1 = [0, t_{aa}] \) and \( I_2 = [t_{aa}, \infty) \) that overlap the interval of the positive constraint. The negative constraint on waiting at
\(B (\text{wait\_at}(B))\) creates two safe intervals for \(B\), \(I_3 = [0, t_{ab}]\) and \(I_4 = [t_{ab}, \infty)\).

Now assume that there are two plans that satisfy the action landmark for the positive constraint, one that reaches \(A\) before \(t_{aa}\) (shown in yellow) and one that reaches \(A\) after \(t_{aa}\) (show in green). Clearly, the lowest-cost plan to achieve the action landmark is the one that reaches \(A\) before \(t_{aa}\), but to find the optimal solution one must use the second plan. Figure 2 illustrates an even more extreme case, where choosing to lowest-cost plan that achieves the action landmark cannot be extended to a full plan, because it reaches \(B\) during its unsafe interval (marked in red).

Generalized SIPP  
There may be an infinite plans that satisfy a given action landmark \(l = (i, \text{move}(A, B), [t, t])\) and it is not sufficient to only find the lowest-cost one. However, it is sufficient to find the lowest-cost plan to reach \(A\) for every safe interval of \(A\) that overlaps with \([t, \infty)\).

To this end, we create a generalized version of SIPP such that: (1) it accepts a set of goal states, one per safe interval of \(A\) that overlaps with \([t, \infty)\), and (2) it outputs a set of plans, one per goal state. To each of these plans, we concatenate the action landmark itself \(\text{move}(A, B)\). These plans may end in different safe intervals in \(B\), which will then be distinct start states when searching for a plan to get from \(B\) to the next landmark. Thus, our generalized SIPP accepts a set of starts states and a set of goal states and outputs a set of plans, one per goal. It works as follows. First, the open list is initialized with all start states. Then, the search proceeds as in regular SIPP, except that the stop criteria is either when the agent arrives to its goal and stays there forever. Thus, even if numerous plans to the preceding landmark were found they all will collapse into a single one, i.e. the one that achieves the final goal at the earliest possible time which is what CCBS requires.

Algorithm 1: Low-level search for CCBS with DS

\[
\text{Input:} \text{Negative constraints } C^(-) \\
\text{Input:} \text{Positive constraints } C^+) \\
\text{Input:} \text{Agent } i \\
1. S \leftarrow \text{ComputeSafeIntervals}(C^(-)) \\
2. L \leftarrow \text{ComputeLandmarks}(C^+) \cup S \\
3. \text{Starts} \leftarrow \{s_i\} \\
4. \text{foreach landmark } l = (i, \text{move}(A, B), [t, t]) \in L \text{ do} \\
5. \text{Goals} \leftarrow \text{computeGoals}(l) \\
6. \text{Plans} \leftarrow \text{SIPP}(\text{Starts}, \text{Goals}) \\
7. \text{Starts} \leftarrow \emptyset \\
8. \text{foreach plan in Plans do} \\
9. \text{Append } \text{move}(A, B) \text{ to plan} \\
10. \text{Add last state in plan to Starts} \\
11. \text{Starts} \leftarrow \text{Prune Plans/Starts if possible} \\
12. \text{return SIPP(Starts, } g_i) \\
\]

Pseudo Code  
Finally, we can describe the pseudo-code for the CCBS-DS low-level search. It accepts a list of negative and positive constraints for an agent \(i\). Let \text{SIPP}(\text{Starts}, \text{Goals}) denote our generalized SIPP. Initially, the low-level search computes the safe intervals of every vertex based on the negative constraints. Then, it computes the action landmarks based on the positive constraints. These landmarks are sorted by time, and then it iterates over these landmarks. For each action landmark \(l = (i, \text{move}(A, B), [t, t])\), it computes the safe intervals of \(A\) that intersect with \([t, t]\). Every such safe interval is considered a goal for our generalized SIPP. When all such goals are added, we run our generalized SIPP to find a set of plans, one per goal. Then, for each found plan we concatenate the action move \(\text{move}(A, B)\) to its end. If \(B\) is not reachable within a safe interval then the plan is discarded. If two or more concatenated plans safely reach \(B\) in the same interval \(I_{safr_k}\) we prune such plans leaving the only one that reaches this interval earlier. A node \((B, I_{safr_k})\) now becomes one of the start nodes for the subsequent search and is added to Starts.

Note that the number of concatenated plans satisfying each landmark \(l\) is proportional to the number of the negative constraints associated with the wait actions for the target vertex of \(l\). Consequently, if no wait actions are prohibited at this vertex only one plan to this landmark will be present after pruning (no matter with how many different start and goal nodes the search was initialized). In general, in the process of the iterative invocation of the modified SIPP and plan pruning, numerous plans constructed so far might eventually collapse to a single one. This definitely happens when one is planning to the final goal. The reason is that this goal is defined by a single graph vertex and a single time interval ending with \(\infty\) (recall that we assume that the agent arrives to its goal and stays there forever). Thus, even if numerous plans to the preceding landmark were found they all will collapse into a single one, i.e. the one that achieves the final goal at the earliest possible time which is what CCBS requires.

Prioritizing Conflicts

Prioritizing Conflicts (PC) \cite{boyarski2015} is the second CBS enhancement we migrate to CCBS. PC is a heuristic for choosing which conflict to resolve when expanding a CT node. Such a heuristic is needed when the CBS high-level search expands a node \(N\) having multiple conflicts in its joint plan (N.II). Different ways to choose conflicts in practice often lead to CT of different sizes, thus have a significant effect on the overall runtime. PC systematically prioritizes conflicts by classifying each conflict as either cardinal, semi-cardinal, and non-cardinal. A conflict \((i, x, t)\) is called cardinal iff splitting a CT node \(N\) over it results in two child nodes whose cost is higher than the cost of \(N\). A conflict is semi-cardinal iff the cost of only one child increases while the cost of the other does not. A conflict that is not cardinal or semi-cardinal is non-cardinal. CBS with PC prefers cardinal conflicts to semi-cardinal and semi-cardinal to non-cardinal. This way of prioritizing conflicts results in a significant reduction of the expanded CT nodes compared to vanilla CBS and makes the algorithm much faster in practice.

In MAPF, most conflicts are cardinal, i.e. the agents involved in that conflicts are not able to find the paths that respect the corresponding constraints and are of the same cost as before. The reason for that is that the ability to perform wait actions of arbitrary duration paired with non-uniform
costs of the move actions reduces the symmetry in MAPF$_R$ setting. Thus, differentiating the conflicts based just on their cardinality type is insufficient.

To this end, we propose a generalized version of PC that introduces a finer-grained prioritization of conflicts, by introducing the notion of cost impact. Intuitively, the cost impact of a conflict is how much the cost of the solution is increased when it is resolved. More formally, for a CT node $N$ with a CCBS conflict $\text{Con} = (a_i,t_i,a_j,t_j)$, let $N_i$ and $N_j$ be the CCBS nodes obtained by splitting over this conflict, and let $\delta_i$ be the difference between the cost of $N$ and $N_i$. We define the cost impact of the conflict $\text{Con}$, denoted $\Delta(\text{Con})$, as $\min(\delta_i, \delta_j)$. Our adaptation of PC to CCBS chooses to split a CT node on the conflict with the largest cost impact. This follows the same rationale as PC, as we prioritize the resolution of conflicts that will reveal the highest unavoidable cost that was so far hidden in conflicts.

**Heuristics for High-Level Search**

To guarantee optimality, the high-level search in CBS explores the CT tree in a best-first fashion, expanding in every iteration the CT node in OPEN with the smallest cost. Felner et al. [2018] and Li et al. [2019a] introduced admissible heuristics to the CBS high-level search. These heuristics estimate the cost difference between the cost of a node and the cost of the optimal solution. Both heuristics are admissible, i.e., they are a lower bound on the actual cost difference, and therefore can be safely added to the cost of a CT node when choosing which node to expand next from OPEN. Indeed, these heuristics were shown to significantly decrease the number of the expanded CT nodes and improve the performance of CBS.

Drawing from these works we suggest two admissible heuristics for CCBS. The first admissible heuristic, denoted $H1$, is based on solving the following linear programming problem (LPP). This LPP has $n$ non-negative variables $x_1, \ldots, x_n$, one for each agent. Each conflict $\text{Con}_{i,j}$ between agents $i$ and $j$ in the CT node for which we are computing the heuristic introduces the LPP constraint $x_i + x_j \geq \Delta(\text{Con}_{i,j})$. The objective to be minimized is $\sum_{i=1}^{n} x_i$. By construction, for any solution to this LPP, the value $\sum_{i=1}^{n} x_i$ is an admissible heuristic since for every conflict $\text{Con}_{i,j}$ the solution cost is increased by at least $\Delta(\text{Con}_{i,j})$.

The second admissible heuristic we propose, denoted $H2$, follows $h_1$ the approach suggested in [Felner et al. 2018]. There, the heuristic was based on identifying disjoint cardinal conflicts, which are cardinal conflicts between disjoint pairs of agents. As discussed above, in CCBS most conflicts are cardinal but their cost impact can vary greatly. Therefore, in the $H2$ heuristic we aim to choose the disjoint cardinal conflicts that would have the largest cost impact. We do so in a greedy manner, sorting the conflicts in $N$.II in descending order of their cost impact. Then, conflicts are picked one by one in this order. After a conflict is picked, we remove from the conflict list all conflicts that involve at least one of the agents in this conflict. This process continues until all the conflicts are either picked or removed. The $H2$ heuristic is the sum of the cost impacts of the chosen conflicts. By construction the chosen conflicts are disjoint and so $H2$ is admissible. While $H2$ is less informed than $H1$ (the one computed by solving LPP), it is faster to compute. We observed experimentally that the practical difference between these heuristics was negligible – an average difference of 1%. Thus, in for the experimental results below we used $H2$ and refer to it as $H$.

**Empirical Evaluation**

We have incorporated all the CCBS enhancements described so far and evaluated different versions of CCBS in different MAPF$_R$ scenarios involving general graphs (roadmaps) and grids. Specifically, we evaluated the basic CCBS, CCBS with PC (CCBS +PC), CCBS with DS (CCBS-DS), CCBS with both DS and PC (CCBS +DS +PC), and CCBS with all the improvements (CCBS +DS +PC + H). In the conducted experiments all agents are assumed to be disk-shaped with radius equal to $\sqrt{2}/4$.

In each run of the evaluated algorithm, we recorded the algorithm’s runtime, the number of expanded CT nodes, and whether the algorithm was able to find a solution under a time limit of 30 seconds or not. One of the primary evaluation metrics in our evaluation is the Success rate, which is the ratio $N_{sol}/N_{all}$, where $N_{sol}$ is the number of instances the algorithm managed to solve under the imposed time cap and $N_{all}$ is the total number of instances.

**Implementation Details**

Conflict detection in MAPF$_R$ is more involved than in classical MAPF and is more computationally intensive. To compensate for that we have implemented the following approach to cache the intermediate conflict detection results and speed up the search. We detect all the conflicts in the root CT node and store them with the node. After choosing a conflict and performing a split we copy all the conflicts to a successor node except the ones involving the agent that has to re-plan its path. After such re-planning newly introduced conflicts (if any) are added to the set of conflicts for that CT node. To compute the cost impacts of the conflicts for versions of CCBS that use PC or the high-level search heuristic $H$, we run the low-level search explicitly to resolve these conflicts and acquire the needed cost increase values. For versions of CCBS without PC, we picked the latest conflict out of all conflicts associated with the node. Our preliminary tests have showed that this ad-hoc strategy leads to better results when compared to choosing a conflict randomly or choosing the earliest conflict.

To speed-up the low-level search, we pre-compute a set of heuristics, $h_1, \ldots, h_n$ each of which estimates costs-to-go to a particular goal $g_i$. To compute $h_1$, we run Dijkstra’s algorithm with $g_i$ as the source node. Indeed, such heuristics are more informative compared to e.g. Euclidean distance. When DS is used, the low-level search performs multiple
searches to achieve the landmarks created by the positive constraints. When searching for the intermediate goals associated with each landmarks, we implemented a Differential Heuristic (DH) (Goldenberg et al. 2011) that uses the precomputed heuristics $h_1, \ldots, h_n$ as pivots.

**Evaluation on the Roadmaps**

In the first set of experiments we have evaluated CCBS on 3 different roadmaps, referred to here as sparse, dense and super-dense. The sparse roadmap contains 158 nodes and 349 edges, the dense roadmap contains 878 nodes and 7,341 edges, and the super-dense roadmap contains 11,342 vertices and 263,533 edges. All of these graphs were automatically generated by applying a roadmap-generation tool from the Open Motion Planning Library (OMPL) (Sucan, Moll, and Kavraki 2012) on the den520d map from the game Dragon Age Origin (DAO). This map is publicly available in the MovingAI MAPF benchmark (Stern et al. 2019).

For each roadmap, 25 different scenarios were generated. Each scenario is a list of start-goal vertices, chosen randomly from the graph. Then, we pick the first $n = 2$ start-goal pairs and create a MAPF$_R$ instance for $n$ agents. If the evaluated algorithm solves this instance within the 30 seconds time limit, we proceed by increasing $n$ by 1 and creating a new MAPF$_R$ instance. This is repeated until the evaluated algorithm is not able to solve the instance in 30 seconds. We then proceed to the next scenario.

The results are shown in Fig. 3. Consider first the success rate plots (left). The first clear trend we observe is that all the proposed CCBS improvements are significantly better than the baseline CBS in almost all cases. E.g., on the dense roadmap CCBS +DS +PC +H manages to achieve 0.8 success rate for the instances with 20 agents, while CCBS success rate for this number of agents is only 0.1.

Next, consider the relative performance of CCBS with different combinations of improvements. In general, the most advanced version of the algorithm, i.e. CCBS +DS +PC +H, outperforms the competitors on sparse and dense roadmaps. However on the super-dense this is not the case. On this roadmap, CCBS +DS +PC +H is dominated by CCBS +DS which was able to solve 25 agents while the former – 20. Indeed, in this roadmap the PC component on its own is ineffective, as can be seen when comparing the basic CCBS and CCBS +PC. We explain this behavior by observing that this roadmap has a very high branching factor (every vertex has almost 50 neighbors on average). This helps to eliminate conflicts by finding an appropriate detour of nearly the same cost. Thus the cost impacts, which are computationally intensive to compute, are very low and provide limited value in differentiating between the conflicts.

Next, consider the runtime and expanded CT nodes plots in Figure 3. These plots are built in the following fashion. Each data point $(x, y)$ on a plot says that an algorithm was able to solve $x$ problem instances within $y$ seconds/CT nodes expansions. For example, on the dense roadmap CCBS solved only 276 instances in less than 1 second, CCBS +PC – 340 instances, while CCBS +DS +PC +H – 404. In general, the closer the line to $x$-axis and the longer it is – the better. The values at the end of the lines show the exact numbers of the solved instances.

The general trend for runtime and high-level expansions are similar to the ones for the success rate: CCBS +DS +PC
Figure 4: Success rates for CCBS and its modifications on different $2^k$-connected grids.

+H is the best on sparse and dense roadmaps and CCBS +DS – on the super-dense one. Also, these results highlight the benefit of our improvements over vanilla CBS, where our best CCBS version is up to 2 orders of magnitude faster in some cases.

We also analyzed separately the impact of adding the high-level heuristic ($H$) on the instances that involve large numbers of CT expansions. We took the results of 100 instances with the highest values of expanded CT nodes solved by CCBS +DS+PS and CCBS +DS+PC+H averaged the number of expansions and compared them. The number of expansions for CCBS +DS+PC+H was lower by 26.5%, 21.6% and 17.8% for sparse, dense and super-dense roadmaps respectively. Thus, adding heuristic proved to be a valuable technique, especially for the hard instances involving large number of expansions.

Evaluation on Grids

The second set of experiments we conducted was on 8-connected ($2^3$) and 32-connected ($2^5$) grids from the MovingAI MAPF benchmark (Stern et al. 2019). Specifically, we used a 16x16 empty grid (16x16 empty), a warehouse-like grid (warehouse-10-20-10-2-2), and a grid representation of the den520d DAO map mentioned above. Here we used the 25 scenario-files supplied by the MAPF benchmark for each grid. The results of the second series of experiments are shown in Fig.4

Here we can see that in almost all cases the best results were obtained by CCBS with all our enhancements (CCBS +DS +PC +H). Comparing the results on grids with different connectedness, the one can notice the same trend as observed for roadmaps with respect to the benefit of PC and DS: increasing the branching factor makes PC less effective and DS more effective. This benefit for DS is explained by the fact that positive constraints help to reduce the branching factor by reducing the amount of possible alternative trajectories to one. Thus, higher branching factor means stronger pruning by positive constraints.

Finally, we calculated the ratios of expansions between vanilla CCBS and the other versions on the 100 instances in each grid that incurred the highest number of CT node expansions by the basic CCBS and were solved by all CCBS variants. The corresponding median values are presented in Table I. As one can see, the least amount of expansions in all the cases is required for the most advanced version, i.e. CCBS+DS+PC+H. Also, we observe that in most cases additional connectivity of the grid makes all the enhancements less beneficial.

Table 1: The ratio of expanded CT-nodes between CCBS and its modifications on grids (the lower – the better).

Conclusions and Future Work

In this work, we have proposed three improvements to CCBS, an algorithm for finding optimal solutions to MAPF$_R$ problems in which time is continuous. The first CCBS improvement we proposed, called DS, changes how CT nodes are expanded by introducing positive and negative constraints. To implement this improvement, we modified the CCBS low-level search and applied a generalized version of SIPP with multiple start and goal nodes. The second improvement, called PC, prioritizes the conflicts to resolve by computing the cost of the solution that resolves them. The third CCBS improvement we proposed is two admissible heuristics for the high-level search.

In a comprehensive experimental evaluation, we observed that using these improvements, CCBS can scale to solve much more problems than the basic CCBS, solving in some cases almost twice as many agents. Allowing CCBS to scale to larger problem is key to applying it to a wider range of real-world applications and also as a foundation for more generate MAPF settings in which the underlying graph is also changing rapidly.
References

Andreychuk, A.; Yakovlev, K.; Atzmon, D.; and Stern, R. 2019. Multi-Agent Pathfinding with Continuous Time. In Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI 2019), 39–45.

Boyarski, E.; Felner, A.; Stern, R.; Sharon, G.; Tolpin, D.; Beitzalel, O.; and Shimony, S. E. 2015. ICBS: Improved Conflict-Based Search Algorithm for Multi-Agent Pathfinding. In the International Joint Conference on Artificial Intelligence (IJCAI), 740–746.

Cohen, L.; Uras, T.; Kumar, T. S.; and Koenig, S. 2019. Optimal and bounded-suboptimal multi-agent motion planning. In Symposium on Combinatorial Search (SoCS).

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. S.; and Koenig, S. 2018. Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. In the International Joint Conference on Automated Planning and Scheduling (ICAPS), 83–87.

Gange, G.; Harabor, D.; and Stuckey, P. J. 2019. Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation. In the International Conference on Automated Planning and Scheduling (ICAPS), 155–162.

Goldenberg, M.; Sturtevant, N. R.; Felner, A.; and Schaeffer, J. 2011. The Compressed Differential Heuristic. In Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI 2011), 24–29.

Karpas, E.; and Domshlak, C. 2009. Cost-Optimal Planning with Landmarks. In IJCAI, 1728–1733.

Lam, E.; Bodic, P. L.; Harabor, D. D.; and Stuckey, P. J. 2019. Branch-and-Cut-and-Price for Multi-Agent Pathfinding. In International Joint Conference on Artificial Intelligence (IJCAI), 1289–1296.

Li, J.; Felner, A.; Boyarski, E.; Ma, H.; and Koenig, S. 2019a. Improved Heuristics for Multi-Agent Path Finding with Conflict-Based Search. In Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-2019), 442–449. doi:10.24963/ijcai.2019/65.

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 (ICAPS), volume 29, 279–283.

Morris, R.; Pasareanu, C. S.; Luckow, K. S.; Malik, W.; Ma, H.; Kumar, T. K. S.; and Koenig, S. 2016. Planning, Scheduling and Monitoring for Airport Surface Operations. In Planning for Hybrid Systems, Papers from the 2016 AAAI Workshop.

Phillips, M.; and Likhachev, M. 2011. SIPP: Safe interval path planning for dynamic environments. In Proceedings of The 2011 IEEE International Conference on Robotics and Automation (ICRA 2011), 5628–5635.

Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks Revisited. In AAAI, volume 8, 975–982.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multiagent path finding. Artificial Intelligence Journal 218: 40–66.

Silver, D. 2005. Cooperative Pathfinding. In the First Artificial Intelligence and Interactive Digital Entertainment Conference, 117–122.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. S.; et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In Proceedings of the 12th Annual Symposium on Combinatorial Search (SoCS 2019), 151–158.

Şucan, I. A.; Moll, M.; and Kavraki, L. E. 2012. The Open Motion Planning Library. IEEE Robotics & Automation Magazine 19(4): 72–82. doi:10.1109/MRA.2012.2205651. https://ompl.kavrakilab.org.

Surynek, P. 2010. An Optimization Variant of Multi-Robot Path Planning Is Intractable. In AAAI, 1261–1263.

Surynek, P.; Felner, A.; Stern, R.; and Boyarski, E. 2016. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In ECAI.

Veloso, M. M.; Biswas, J.; Coltin, B.; and Rosenthal, S. 2015. CoBots: Robust Symbiotic Autonomous Mobile Service Robots. In the International Joint Conference on Artificial Intelligence (IJCAI), 4423–4429.

Walker, T. T.; Sturtevant, N. R.; and Felner, A. 2018. Extended Increasing Cost Tree Search for Non-Unit Cost Domains. In IJCAI, 534–540.

Wurman, P. R.; D’Andrea, R.; and Mountz, M. 2007. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. In the AAAI Conference on Artificial Intelligence (AAAI), 1752–1760.

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