Parallel Hierarchical Composition Conflict-Based Search for Optimal Multi-Agent Pathfinding

Hannah Lee, James Motes, Marco Morales, and Nancy M. Amato

***Abstract*—In this letter, we present the following optimal multi-agent pathfinding (MAPF) algorithms: Hierarchical Composition Conflict-Based Search, Parallel Hierarchical Composition Conflict-Based Search, and Dynamic Parallel Hierarchical Composition Conflict-Based Search. MAPF is the task of finding an optimal set of valid path plans for a set of agents such that no agents collide with present obstacles or each other. The presented algorithms are an extension of Conflict-Based Search (CBS), where the MAPF problem is solved by composing and merging smaller, more easily manageable subproblems. Using the information from these subproblems, the presented algorithms can more efficiently find an optimal solution. Our three presented algorithms demonstrate improved performance for optimally solving MAPF problems consisting of many agents in crowded areas while examining fewer states when compared with CBS and its variant Improved Conflict-Based Search.**

***Index Terms*—Multi-robot systems, path planning, parallel algorithms.**

I. Introduction

In multi-agent pathfinding (MAPF) problems, we are given an environment with a set of agents, each with their own respective start and goal positions. At a given timestep, an agent can either move to a neighboring location or wait at its current location. The goal of MAPF is to find a set of actions for each agent, such that no agent collides with another while reaching its goal. MAPF is used in many application domains such as assembly [1], evacuation [2], localization [3], and object transportation [4].

There are three general classes of MAPF algorithms: coupled, decoupled, and hybrid. Coupled approaches compute agent paths in unison. They use a joint space to represent all agent states, allowing many coupled approaches to guarantee feasible and optimal path plans [5], [6]. However, finding an optimal solution for the MAPF problem is NP-hard as the state space grows exponentially with the number of agents [7]. Thus, coupled solvers are not viable options for problems with large numbers of agents because the state space grows to be too large to solve.

Decoupled approaches plan for agents independently and adjust the paths to meet problem constraints or avoid inter-agent conflicts [8], [9]. Unlike coupled approaches, decoupled approaches work with smaller search spaces by planning for individual agents, allowing them to scale to larger MAPF problems. However, planning for individual agents leads to issues with completeness and optimality; thus, decoupled planners are incomplete and provide sub-optimal solutions.

Hybrid approaches leverage the optimality of coupled approaches and the fast computation of decoupled approaches to efficiently find optimal solutions [10], [11]. Conflict-Based Search (CBS) is a state-of-the-art hybrid MAPF algorithm that uses a two-level search method to guarantee optimal solutions. The low-level search finds optimal paths for individual agents. The high-level search explores a constraint tree (CT) that resolves inter-agent conflicts. The high-level search tree, the CT, grows exponentially with the number of found conflicts; therefore, CBS performs poorly for problems with highly coupled agents that result in many inter-agent conflicts that are tedious to resolve [12]. To improve the performance of CBS, four improvements were presented in Improved Conflict-Based Search (ICBS) that improve performance by limiting conflicts and the number of branches within the constraint tree [13]. There exist many variants of CBS [14]–[16] and algorithms that utilize it to solve MAPF problems and other problems such as multi-agent pickup and delivery [17], MAPF for large agents [18], MAPF with delay probabilities [19], and task and motion planning [20].

In this work, we present Hierarchical Composition Conflict-Based Search (HC-CBS), Parallel Hierarchical Composition Conflict-Based Search (PHC-CBS), and Dynamic Parallel Hierarchical Composition Conflict-Based Search (DPHC-CBS). These algorithms are extensions of CBS where the MAPF problem is solved by considering smaller subproblems that are evaluated and hierarchically composed into larger subproblems until the original MAPF problem is formed. Multithreading is used in PHC-CBS and DPHC-CBS to further improve the performance of our algorithms. The presented algorithms provide significant speedup for large MAPF problems while examining fewer states when compared to CBS and ICBS. From our results, we can conclude that the proposed algorithms are a better alternative

Manuscript received February 24, 2021; accepted June 23, 2021. Date of publication July 13, 2021; date of current version July 28, 2021. This letter was recommended for publication by Associate Editor L. Pimenta and Editor M. A. Hsieh upon evaluation of the reviewers' comments. The work of H. Lee was supported in part by the National Science Foundation Graduate Research Fellowship under Grant 2020297899. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The work of M. Morales is funded by the Academia Mexicana de Cultura A.C. *(Corresponding author: Hannah Lee.)*

Hannah Lee, James Motes, and Nancy M. Amato are with the Parasol Lab, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL 61820 USA (e-mail: hannah9@illinois.edu; jmotes2@illinois.edu; namato@illinois.edu).

Marco Morales is with the Parasol Lab, Department of Computer Science, University of Illinois at Urbana-Champaign, Champaign, IL 61820 USA, and with the Instituto Tecnológico Autónomo de México, Mexico City 01080, Mexico (e-mail: moralesa@illinois.edu).

Digital Object Identifier 10.1109/LRA.2021.3096476

for solving large MAPF problems. This benefits many of the variants and applications of CBS mentioned before, as they can leverage the performance improvement from the proposed algorithms over standard CBS.

II. Background

MAPF is a well-studied problem in robotics; however, not much work is available on parallel MAPF algorithms. In [21], the authors present a local collision avoidance algorithm that is parallelized using data-parallelism and thread-level parallelism. In [22], a multi-agent A* algorithm is presented along with a parallel variant. Their A* algorithm leads to super-linear speedup but does not perform well for tightly coupled agents. For general motion planning, there is more extensive work available on parallelized algorithms. For a comprehensive survey, see [23].

A. Conflict-Based Search (CBS)

In CBS, we are given a set of n agents $a_0, a_1, \ldots, a_{n-1}$. Each agent a_i has a start s_i and goal g_i position. An agent's path is the sequence of actions leading from s_i to g_i. The algorithm is tasked with finding a set of n paths that are free of vertex and edge conflicts. Vertex conflicts occur when two agents inhabit the same position. Edge conflicts occur when agents attempt to transition along the same edge at the same time. Conflicts are represented as a tuple. A vertex conflict is represented as $\langle a_i, a_j, v, t\rangle$. This states that agents a_i and a_j occupy vertex v at timestep t. For edge conflicts, the conflicting edge is represented with a pair of vertices, $\langle a_i, a_j, (v_0, v_1), t\rangle$. A solution is the set of all agent paths and is only valid if all paths are conflict-free. The cost of a path is the number of actions required to move from s_i to g_i. The cost of a solution is determined by a defined cost metric. Two common metrics are Sum-of-Costs and Makespan. Sum-of-Costs is the summation of all agent path costs. Makespan is the maximum path cost of all agents. For the purpose of this letter, we will be using Sum-of-Costs as our cost metric.

CBS is a two-level search method. The low-level search consists of a pathfinding algorithm that finds an optimal sequence of actions for an agent a_i to move from s_i to g_i. The high-level search explores a constraint tree (CT) that iteratively resolves conflicts by generating sets of constraints that restrict individual agents [12]. The CT is a binary tree where each node N contains a set of constraints imposed on the agents ($N.constraints$), a solution ($N.solution$), and the cost of $N.solution$ ($N.cost$). Constraints are represented by a tuple $\langle a_i, v, t\rangle$ which states that agent a_i cannot access vertex v at timestep t. Edge constraints contain a pair of vertices denoting the edge. The solution and paths within a CT node must be consistent. A consistent path for agent a_i is a path that adheres to all of a_i's constraints. A consistent solution is a solution composed of only consistent paths. A solution is valid if it is consistent and conflict-free.

The initial root node of the CT contains an empty set of constraints and an initial path for each agent. All following nodes will contain a set of d constraints where d is the depth of the CT. $N.solution$ is found using CBS's low-level search algorithm. $N.cost$ is calculated from $N.solution$. The high-level search of CBS greedily evaluates lowest cost nodes.

Given a CT node N, $N.solution$ is validated by examining every agent's path with respect to every other agent's path for conflicts. If no conflict is found, $N.solution$ is valid and N can be returned as a goal node. If a conflict $\langle a_i, a_j, v, t\rangle$ is found, the node N is resolved by a splitting action.

When a non-goal node N is split to resolve a conflict $\langle a_i, a_j, v, t\rangle$, two new CT nodes, N_i and N_j, are generated as children of N. N_i and N_j both receive a copy of $N.constraints$ and $N.solution$. To resolve the conflict, two constraints are created: $\langle a_i, v, t\rangle$ and $\langle a_j, v, t\rangle$. One of these constraints is placed on N_i and the other on N_j. When N_i receives the constraint $\langle a_i, v, t\rangle$, $N_i.solution$ is no longer consistent. Agent a_i's path is inconsistent while all other agent paths remain consistent, so only a_i's path must be replanned. The same is true for N_j. Once agents a_i and a_j have been replanned within nodes N_i and N_j, respectively, both nodes will have consistent solutions. $N_i.cost$ and $N_j.cost$ are updated to reflect their new solutions. By creating two constraints that disjointly limit a_i and a_j, CBS guarantees optimality by examining both possibilities and ensuring that no states within the search space are lost or overlooked.

B. Improved CBS (ICBS)

ICBS consists of four improvements that can be used separately or in conjunction. The only exception to this is merge and restart (MR), which can only be used when meta-agent CBS is present. For the purpose of this letter, all four improvements are used in conjunction.

- **Meta-agent CBS (MA-CBS):** MA-CBS generalizes CBS by merging small groups of agents into meta-agents when the number of conflicts seen between a pair of agents exceeds a predetermined threshold B. Once a node decides to merge a group of agents, the merged agents are searched and evaluated in a composite state for all following nodes. MA-CBS reduces the runtime of CBS when a good threshold is chosen [24].
- **Merge and Restart (MR):** MR can only be used with MA-CBS. In MA-CBS, agents are merged locally at each node within the CT and remain merged for all following nodes. With MR, when a meta-agent is made, the search is restarted. The CT is cleared and starting from the root node, the merged agents are handled as a meta-agent for the entirety of the search [13].
- **Bypass (BP):** When node N is validated and a conflict $\langle a_i, a_j, v, t\rangle$ is found, BP tries to find an alternative path for one of the conflicting agents, a_i or a_j. For an alternative path to be considered a bypass, the path must have the same cost as the original path without containing the conflict $C_{i,j}$. If node N has a conflicting agent a_i and a valid bypass P_i' is found, N is not split into child nodes. Instead, the P_i within $N.solution$, is replaced with the valid bypass path, P_i', and the node is re-evaluated in the following iteration of CBS without growing the tree [25].
- **Prioritized Conflicts (PC):** In CBS, nodes are split based on the first occurring conflict with respect to the timestep. CBS's CT is sensitive to th set t of ae order in which conflicts are resolved as it influences the number of generated

CT nodes. A poor conflict choice can increase the size of the CT and lead to poor performance. Instead of arbitrarily choosing conflicts, conflicts can be prioritized by categorizing them into three types: cardinal, semi-cardinal, and non-cardinal [13]. A conflict $\langle a_i, a_j, v, t\rangle$ is cardinal if adding *either* of the two constraints, $\langle a_i, v, t\rangle$ or $\langle a_j, v, t\rangle$ results in an increase in the agent's path cost. A conflict is semi-cardinal if adding *one* of the constraints results in an increase in the path cost. A conflict is non-cardinal if *neither* of the two constraints results in an increase in the path cost. The PC improvement splits nodes by first prioritizing cardinal conflicts, then semi-cardinal conflicts, and then finally non-cardinal conflicts. When PC is used in conjunction with BP, BP is only applied to semi-cardinal or non-cardinal conflicts.

III. Methods

We present three algorithms: Hierarchical Composition Conflict-Based Search (HC-CBS), Parallel Hierarchical Composition Conflict-Based Search (PHC-CBS), and Dynamic Parallel Hierarchical Composition Conflict-Based Search (DPHC-CBS). These are extensions of CBS where instead of solving a given MAPF problem all at once, the algorithms consider smaller, more easily managed subproblems. Subproblems are MAPF problems where only a subset of agents from the full problem are considered. These subproblems are solved and hierarchically composed into larger subproblems until the original MAPF problem is reformed. The information from subproblems is used to generate a starting point for larger subproblems. By solving less related subproblems independently, solutions can be found faster and earlier within the search by targeting strongly coupled agents and reducing the number of explored states.

The order in which subproblems are solved and merged is determined by a composition function. This will be further discussed in Section III-A. Agent group composition determines which agents should be grouped together into a subproblem. For HC-CBS and PHC-CBS, the ordering of subproblems is determined each time a layer of subproblems has finished executing. PHC-CBS is a parallel implementation of HC-CBS where multithreading is used to parallelize the execution of subproblems. DPHC-CBS differs in that it organizes subproblems dynamically as subproblems finish.

Subproblems are solved using CBS and the information from their completed CTs is merged to create larger subproblems through a process called crossing CTs. Crossing CTs takes information from each subproblem and combines it to provide a starting basis for larger subproblems.

The three proposed algorithms each use agent group composition and crossing CTs to solve MAPF problems. In this methods section, we will discuss agent group composition, crossing constraint trees, and then move on to discuss HC-CBS, PHC-CBS, and DPHC-CBS.

A. Agent Group Composition

Agent group composition functions are used to organize subproblems to be merged based on an estimate of the amount of computational work required to solve the merged subproblems. The organized subproblems are passed on to crossing CTs to be merged and composed into larger subproblems. The goal of the composition function is to organize subproblems such that difficult agents are organized together. We want to resolve strongly paired agents in smaller subproblems before the CT and subproblems grow to be large and cumbersome. If more work is accomplished in smaller subproblems and smaller CTs, less work will need to be done in larger problems. The organization of these agents is determined by estimating the amount of computational work a pair of subproblems will require.

The exact amount of computational work required by a pair of subproblems cannot be calculated without finding a solution to the combined MAPF problem. Without the full CT, we don't know how conflicts will be resolved or how resolved conflicts may impact agent paths. If the subproblems are poorly paired, more work is distributed to larger subproblems, leading to performance that is worse than CBS due to added overhead from grouping and merging. Agent group composition attempts to force the work needed to resolve highly coupled agents into smaller subproblems. As a CT grows, highly coupled agents can cause many nodes within the CT to have to split for the same conflict [12]. By resolving these conflicts within the smaller subproblem CTs, the number of nodes generated from inter-agent conflicts is reduced. Therefore, it is important that agents are intelligently grouped to resolve highly coupled agents early on and distribute work to smaller subproblems.

To group agents, we test three composition functions: random, conflict, and cardinal conflict composition. The work estimation values that are calculated by these functions is used to organize the subproblems for HC-CBS, PHC-CBS, and DPHC-CBS. For HC-CBS and PHC-CBS, the work estimation values are used to organize subproblems in layers. In DPHC-CBS, the values are used to organize subproblems as they are evaluated and returned.

Random agent composition groups agents randomly without considering their paths or inter-agent conflicts. Conflict composition and cardinal conflict composition are more intelligent. However, these composition heuristics still provide very rough estimations as they only observe the initial paths of agents. In Section IV we will show that even with naive composition heuristics, the three proposed algorithms produce significant improvements in memory and execution time in congested environments.

For conflict composition and cardinal conflict composition, we evaluate each subproblem's set of paths with every other subproblem's set of paths for conflicts. For conflict composition, we count and store the number of internal conflicts between every possible subproblem pair. For cardinal conflict composition, we do the same but only consider cardinal conflicts. A conflict is cardinal if adding any of the two constraints and invoking the low-level search on the constrained agent results in an increase in the agent's path cost [13]. We take these conflict counts and sort the subproblems in descending order of internal conflicts and begin pairing based on the greatest number of conflicts. The generated subproblem pairings are independent of one another. An agent only ever exists within a single subproblem for each level of the hierarchy. When creating a new layer in

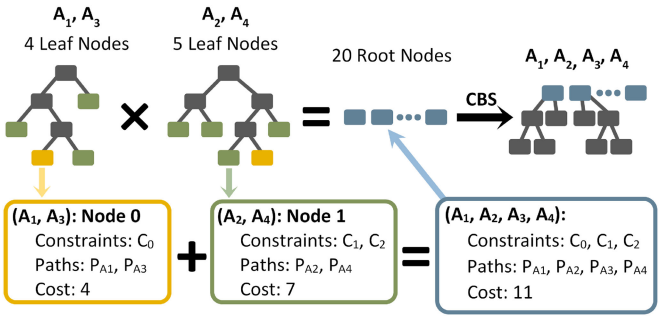


Fig. 1. Crossing CTs takes every combination of leaf nodes from two CTs and merges them to create root nodes. Leaf nodes and goal nodes are shown in green and yellow, respectively. Root nodes are shown in blue. Crossing nodes consists of unioning the nodes' constraints and paths and updating the costs. CBS then begins its search from the root nodes.

the hierarchy, once an subproblem pair is made, the agents that compose the subproblem pair cannot exist within any of the other pairs for the new layer. For example, if (sub_0, sub_1) share 4 conflicts, (sub_0, sub_2) share 3 conflicts, (sub_1, sub_2) share 5 conflicts, the paired agent groupings would be (sub_1, sub_2) and (sub_0), where each subproblem would contain a disjoint set of agents.

Algorithm 1: Hierarchical Composition CBS Finds an optimal, conflict-free path for a set of n agents. For HC-CBS, omit the lines in grey. For PHC-CBS and DHC-CBS, include the lines in grey.

Input: Set of individual agent problems A, thread limit T
Output: Optimal path solution P

```
layer L ← SOLVE(A)
subproblems ← COMPOSE(L)
runningJobs ← 0
do
  L ← ∅
  for S0, S1 in subproblems do
  if runningJobs ≥ T then
    wait until runningJobs < T
  runningJobs ← runningJobs + 1
  L ← L ∪ LAUNCHJOB(CROSSCTS(S0, S1))
  if algorithm is PHC-CBS then
    wait for all threads to finish
  subproblems ← COMPOSE(L)
while |subproblems| > 1 or runningJobs > 1
P ← subproblems[0].Paths
return P
```

B. Crossing Constraint Trees

When we start HC-CBS, PHC-CBS, and DPHC-CBS, we begin by considering the set of single-agent subproblems that make up the MAPF problem. We compose the subproblems into pairings and evaluate them with crossing constraint trees. Then, for all following subproblems, the process of composing subproblem pairs and evaluating with crossing CTs is repeated until the entire MAPF problem is solved.

Crossing constraint trees is a two-step process that starts by generating root nodes to provide a starting basis for the CBS call that is used to solve the parent problem. When the parent is being evaluated, it begins its search from the root nodes provided by its children instead of starting from scratch. Root nodes are generated by crossing the leaf nodes of each child's CT. Then, these root nodes are used to solve the MAPF problem using CBS.

The leaf nodes of a CT consist of the goal node and all unevaluated nodes. When generating root nodes, the information from every single combination of leaf nodes from each child's CT is taken and merged. If child one has n leaf nodes and child two has m leaf nodes, crossing CTs results in $n \times m$ root nodes. Crossing a pair of leaf nodes to create a root node requires unioning the constraints and paths from each child and updating the cost. Passing these root nodes to CBS results in a Constraint Forest, but as it functions the same as a constraint tree, we will refer to it as a CT for consistency. An example of this is shown in Fig. 1. After the root nodes have been generated, CBS is called to begin its high-level search starting from the root nodes. The process of crossing constraint trees is repeated for all internal DT nodes until a solution for the entire problem is found.

When CBS is used to solve a MAPF problem, the constraints and paths that reside within the CT are only relevant to the agents within that problem. CTs from different subproblems contain information that is independent of all other subproblems. Therefore, we can merge leaf nodes of different CTs by simply copying the constraints and paths within each node because nodes of different CTs hold independent information. By crossing every possible combination of leaf nodes from both CTs, crossing constraint trees maintains optimality by ensuring that we do not lose states within the search space, allowing us to explore all possible solutions.

C. Lazy Crossing Constraint Trees

Copying information from all leaf nodes to generate root nodes can be expensive when there are a large number of leaf nodes. To maintain optimality, given n and m leaf nodes, we must generate all $n \times m$ root nodes. If n and m are large, a significant amount of time is spent copying information and generating root nodes. This can lead to poor performance in larger problems. To combat this, we present an optimization to the crossing constraint trees method.

When CBS evaluates generated root nodes, it may not need to access all the root nodes given. This is especially true for larger problems. Thus, some root nodes are never accessed after being made and the time spent copying and generating information for these root nodes is wasted. To reduce the amount of wasted time, we can lazily execute node crossings such that the copying of paths and constraints is only done when the crossed node is about to be evaluated.

Root nodes are lazily generated using an intermediate structure that is used to store the leaf nodes from each child's CT. Then, instead of crossing the nodes by copying, references to the combination of nodes is stored instead. A root node's references indicate which nodes from what subproblem need to be crossed.

In CBS, when the root node is about to be evaluated for conflicts, the node references are used to look up which nodes to cross from the intermediate structure. Once the leaf nodes are retrieved from the intermediate structure, the nodes are crossed by unioning the paths and constraints and then evaluated.

D. Hierarchical Composition CBS (HC-CBS)

HC-CBS solves the MAPF problem in layers. The first layer consists of single agent subproblems that are merged and solved to produce the second layer consisting of two-agent subproblems. The subproblems are repeatedly organized and solved using the composition function and crossing constraint trees, layer by layer, until we have a layer consisting of only a single subproblem, which is our fully solved MAPF problem.

The algorithm for HC-CBS can be seen in Algorithm 1. We start by solving the single agent subproblems (line 1). This creates the base layer L for our group of subproblems where each subproblem consists of a single agent and a single node containing the agent's path. We then repeatedly merge and solve subproblem pairs using crossing CTs (`CrossCTs`, line 10). As subproblems are solved through crossing constraint trees, the newly composed subproblems are stored in L to create the next layer of subproblems to solve. The new layer is then organized using the composition function `Compose` on line 13. This is repeated until the subproblems have all been merged into a single subproblem, which is the fully solved MAPF problem. Thus, we return its path as our solution.

E. Parallel and Dynamic Parallel Hierarchical Composition CBS (PHC-CBS and DPHC-CBS)

PHC-CBS and DPHC-CBS are a direct extension of HC-CBS. To parallelize HC-CBS, the algorithm is multithreaded so that each merged subproblem is evaluated by a single thread. In Algorithm 1, the gray lines indicate the lines that must be added to parallelize HC-CBS. For both PHC-CBS and DPHC-CBS, threads are launched (`LaunchJob`) when subproblem pairs are about to be merged and evaluated using crossing constraint trees (`CrossCTs`).

When subproblems are stored in the `subproblem` list, they are in the "ready" state. "Ready" state subproblem pairs are removed and launched into the "running" state when a new thread is launched to cross their constraint trees. When threads finish solving the merged subproblem, the merged subproblem is put into L. This moves subproblems from "running" to "done." Subproblems that are in L are then moved back into "ready" after they are are organized by the composition function on line 12.

The number of threads running at any given time is limited using a thread limit T and a counter, `runningJobs`, that tracks the number of running jobs to prevent thrashing between different subproblems. Thrashing can lead to unwanted overhead that leads to parallel runtimes that are significantly worse than that of HC-CBS. When threads are launched, the running job count is incremented. When threads finish their execution of crossing CTs, the running job count is decremented to allow new jobs to be launched.

In PHC-CBS, the MAPF problem is solved in distinct layers. Once threads for the current list of subproblems are launched, we must wait until they have all finished before the next layer can be solved (lines 11, 12). For PHC-CBS, the runtime of each layer is bounded by the slowest subproblems because we must wait for the subproblems of each layer to completely finish executing before moving on to the next subproblem. Because the subproblems are organized based on an estimation of the required computational work, the subproblems take varying amounts of time to run. The hardest subproblem in a layer could take significantly more time than the easiest subproblem. This imbalance in runtimes is a consequence of differing CT sizes.

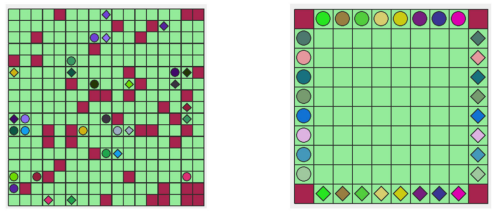
In DPHC-CBS, subproblems are merged dynamically as jobs finish. Whenever a "done" state subproblem is stored in L, DPHC-CBS will organize the subproblems in L using its composition functions. The organized subproblems are moved into `subproblems` to indicate that they are "ready" for evaluation. Then, DPHC-CBS will launch as many jobs as possible from `subproblems`, given the current number of running jobs and the thread limit. Unlike PHC-CBS, it does not wait for threads to finish. While threads are running, it continues to sample and organize L as subproblems finish. The dynamic nature minimizes idle waiting time, which was the main disadvantage of PHC-CBS. However, as the subproblems are organized and merged dynamically, there is added overhead from calling the composition function every time a batch of subproblems has finished.

IV. Experiments

We used three sets of experiments to show the performance of the presented algorithms. These are simulated experiments that do not involve real robots. For all figures, our algorithms are shorthanded from HC-CBS, PHC-CBS, and DPHC-CBS to HC, PHC, and DPHC, respectively, for the sake of space.

The first experiment is a composition function experiment that shows the performance of our presented algorithms with the three composition functions on a random environment. The second and third experiments test the runtime and memory performance of our algorithms against the baseline algorithms CBS and ICBS. For the second experiment, we test the runtime and memory performance of our algorithms with the improvements of ICBS applied. HC-CBS, PHC-CBS, and DPHC-CBS with the ICBS improvements are defined as I+HC, I+PHC, and I+DPHC. Results are given as overall runtime, the number of evaluated nodes, and the total nodes in the CT. All algorithms are run for 20 iterations and the results are averaged from these iterations. For the proposed algorithms, the number of evaluated nodes is calculated from the number of nodes evaluated across all CTs. The total nodes are calculated as the number of nodes in the final CT.

The environments used for these experiments are shown in Fig. 2. The random environments are used in the first and third experiments. There are three sets of random environments. These are generated with an obstacle density of 10%, 15%, and 20% and sized such that there are at least 15 free positions for each agent. Each environment was generated once and the experiments were run 20 times in the same environment. An example of a 15% obstacle density environment is shown in



(a) (b)

Fig. 2. The sample environments shown are labeled as (a) random env. and (b) cross env. Red squares represent obstacles, green squares are the free space, circles are start positions for agents, rhombuses are goal positions. Start and goal positions are paired to an agent by color.

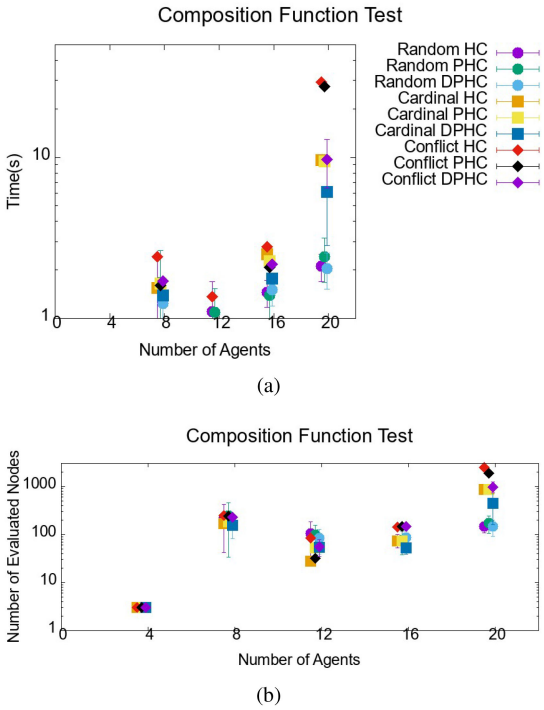


Fig. 3. The three composition functions: random, conflict, and cardinal conflict composition were run on random environments populated with a 15% obstacle density with the HC-CBS, PHC-CBS, DPHC-CBS. Runtime is shown in (a) and the number of evaluated nodes is shown in (b).

Fig. 2(a). The cross environment, Fig. 2(b), is used for the second experiment.

A. Composition Function Tests

The three composition functions were tested on HC-CBS, PHC-CBS, and DPHC-CBS on random environments with an obstacle density of 15%. From the results shown in Fig. 3(a), the algorithms perform best when run with a randomized composition function. For the random environments, the random composition function performs well because agents are uniformly scattered throughout the environment and it is unlikely for patterns within paths to exist.

Randomized composition is a good option for environments that lack structure and pattern. An intelligent subproblem composition function is unnecessary if the environment does not have a natural grouping of agents. When compared against conflict and cardinal conflict composition, random composition also has the least overhead because it does not need to evaluate any of the returned nodes.

The performance of HC-CBS, PHC-CBS, and DPHC-CBS is very sensitive to the subproblem compositions. The composition functions determine which nodes are merged and how the high-level search across CT nodes is performed. Poor compositions can unnecessarily expand the CT and prevent the high-level search from exploring more meaningful branches of the CT. The large differences in evaluation time (Fig. 3(a)) and in the number of evaluated nodes (Fig. 3(b)) between different composition functions displays how sensitive these algorithms are to different subproblem groupings, especially as the number of agents increases.

Composition functions should be applied based on the problem's features. If the composition function can capture the relationship between agent paths, work can be intelligently distributed into subproblems. Random composition may be most successful in environments that lack structure. For an environment that contains agents that frequently collide with many other agents, cardinal conflict composition can help group problematic agents together. Conflict composition may be more useful in environments where agents only collide with small subsets of agents.

B. Runtime and Memory Performance

The runtime and memory results for HC-CBS, PHC-CBS, and DPHC-CBS with and without the ICBS improvements are shown in Fig. 4. The results from the 10%, 15%, and 20% obstacle density random environments are shown in Figs. 5 and 6. The cross environments were run with the conflict composition function and the random environments were run with the random composition function.

We show significant improvements in both runtime and memory in congested environments. Our algorithms are most useful when many inter-agent conflicts need to be resolved and when the composition function distributes meaningful work to its threads. In congested environments, such as environments with narrow passages or with high-activity open spaces, like the cross environments, there are typically many conflicts that make the CBS CT grow rapidly. Improvements in runtime and memory are greatest in congested environments because the composition functions can leverage the conflicts to compose meaningful subproblems that make it easier to solve the MAPF problem. In open, non-congested environments and small agent problems, the added overhead from organizing subproblems outweighs the benefits.

DPHC-CBS introduces more overhead than HC-CBS and PHC-CBS because it reorders its subproblems as they are returned. Thus, the composition function is called more frequently compared to the layer evaluation of HC-CBS and PHC-CBS. Reordering composes better subproblems, but there is noticeable overhead in small agent problems. As agent numbers increase, PHC-CBS and DPHC-CBS perform well when compared to CBS and ICBS. The performance of PHC-CBS and DPHC-CBS with respect to HC-CBS is dependent on the

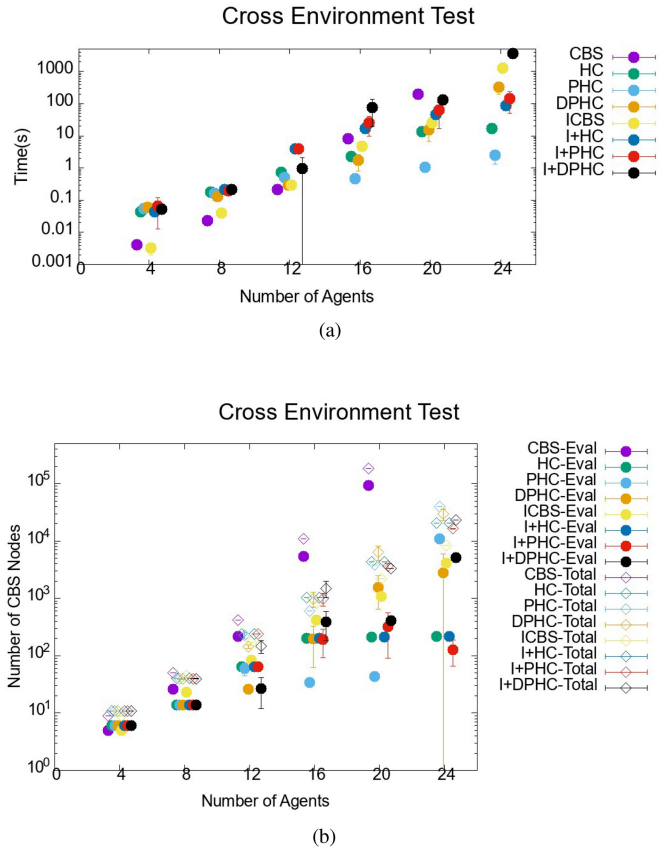


Fig. 4. Average (a) Runtime and (b) Memory performance with standard deviation of HC-CBS, PHC-CBS, DPHC-CBS on the cross environment.

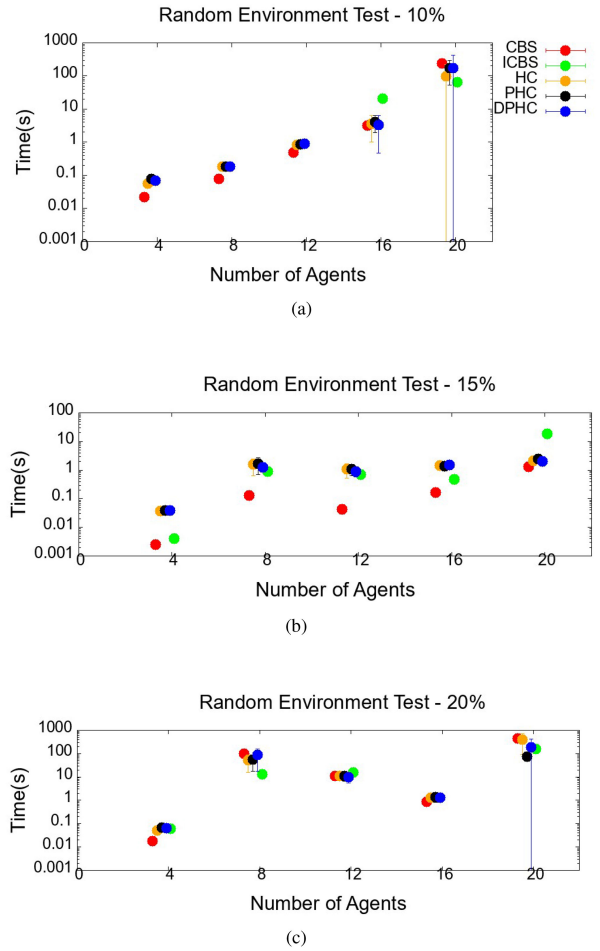


Fig. 5. Average Runtime with standard deviation for random environments populated with (a) 10%, (b) 15%, and (c) 20% obstacle densities.

work distributed to the launched threads. If threads are not given meaningful work, the added overhead of managing shared memory and launching threads makes the parallel implementations perform worse than HC-CBS.

The HC-CBS algorithms don't necessarily benefit from the improvements of ICBS. Although the improvements from ICBS can reduce the number of nodes within the CT, the additional overhead that is introduced to reduce the memory usage is significant. Our algorithms implicitly capture many of the node reduction methods that are present in the ICBS improvements. Our algorithms group agents together, similar to MA-CBS, and target highly coupled agents like the PC improvement. Because of this, the ICBS improvements don't contribute enough in reducing the number of evaluated nodes to make up for the added overhead in runtime.

PHC-CBS performs better than DPHC-CBS when subproblems are relatively balanced in difficulty. The runtime of each layer that is solved by PHC-CBS is bounded by the slowest subproblem. If the subproblems are heavily unbalanced, threads spend a significant amount of time waiting for their sibling thread to finish. In cases like this, DPHC-CBS is more effective because jobs are dispatched based on available subproblems, allowing threads to run one after another without waiting. The parallelization of these methods benefits most when the work is distributed to smaller subproblems. Thus, both PHC-CBS and DPHC-CBS are very dependent on finding good subproblem compositions.

The node counts for HC-CBS, PHC-CBS, and DPHC-CBS are much lower than that of CBS and ICBS. However, the difference between the total number of nodes within the CT to the number of evaluated nodes is much greater for the HC-CBS algorithms. When CTs are crossed, we cross every combination of leaf nodes to generate the root nodes. The generated root nodes contain nodes that are never evaluated; however, these nodes are still included in the CT to maintain optimality. This is why the difference in evaluated nodes and total nodes is so large for the HC-CBS algorithms.

V. Conclusion

We present the multi-agent pathfinding algorithms Hierarchical Composition Conflict-Based Search, Parallel Hierarchical Composition Conflict-Based Search, and Dynamic Parallel Hierarchical Composition Conflict-Based Search that can solve large multi-agent pathfinding problems faster and with less memory than Conflict-Based Search and Improved Conflict-Based Search. We show that our method, which solves the MAPF problem by evaluating smaller subproblems and hierarchically

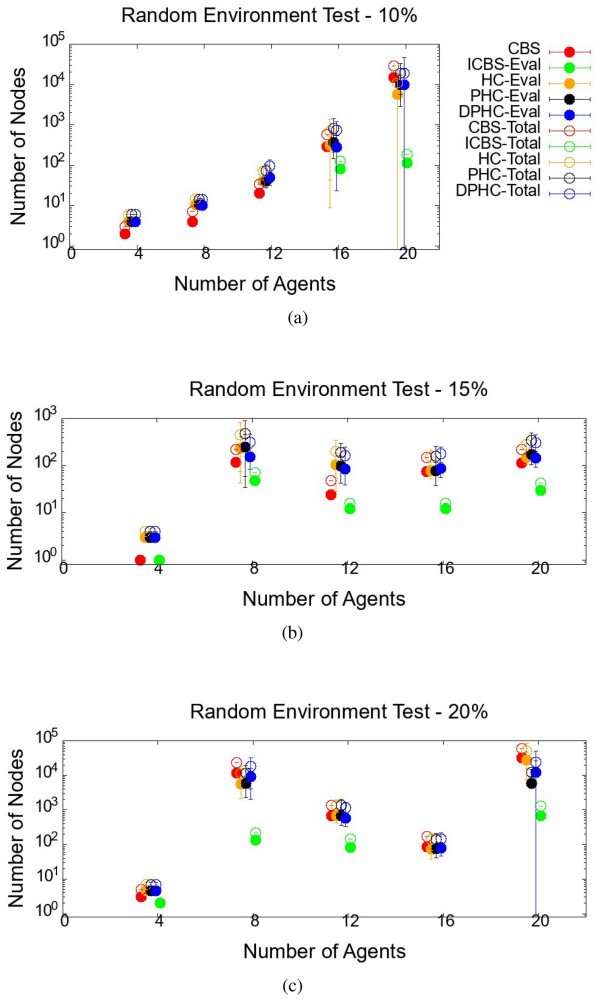


Fig. 6. Average number of evaluated and total CBS nodes with standard deviation for random environments populated with (a) 10%, (b) 15%, and (c) 20% obstacle densities.

composing them into larger problems, achieves improved performance on large problems in congested environments. Future work will seek to explore the importance of different composition functions and optimizations that can improve parallel performance.

REFERENCES

[1] D. Halperin, J.-C. Latombe, and R. Wilson, "A general framework for assembly planning: The motion space approach," *Algorithmica*, vol. 26, no. 3, pp. 577–601, 2000.

[2] S. Rodríguez and N. M. Amato, "Behavior-based evacuation planning," in *Proc. IEEE Int. Conf. Robot. Automat.*, 2010, pp. 350–355.

[3] D. Fox, W. Burgard, H. Kruppa, and S. Thrun, "A probabilistic approach to collaborative multi-robot localization," *Auton. Robots*, vol. 8, no. 3, pp. 325–344, 2000.

[4] D. Rus, B. Donald, and J. Jennings, "Moving furniture with teams of autonomous robots," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. Hum. Robot Interact. Cooperative Robots*, vol. 1, 1995, pp. 235–242.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.

[6] T. Standley, "Finding optimal solutions to cooperative pathfinding problems," in *Proc. 24th AAAI Conf. Artif. Intell.*, 2010, pp. 173–178.

[7] J. Yu and S. M. LaValle, "Structure and intractability of optimal multi-robot path planning on graphs," in *Proc. 27th AAAI Conf. Artif. Intell.*, 2013, pp. 1443–1449.

[8] M. Saha and P. Isto, "Multi-robot motion planning by incremental coordination," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, 2006, pp. 5960–5963.

[9] D. Silver, "Cooperative pathfinding," in *Proc. 1st Conf. Artif. Intell. Interactive Digit. Entertainment*, 2005, pp. 23–28.

[10] G. Wagner and H. Choset, "M*: A. complete multirobot path planning algorithm with performance bounds," in *Proc. IEEE Int. Conf. Intell. Robots. Syst.*, 2011, pp. 3260–3267.

[11] S. Bhattacharya and V. Kumar, "Distributed optimization with pairwise constraints and its application to multi-robot path planning," *Robot.: Sci. Syst. VI*, vol. 177, pp. 177–184, 2011.

[12] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Conflict-based search for optimal multi-agent pathfinding," *Artif. Intell.*, vol. 219, pp. 40–66, 2015.

[13] E. Boyarski *et al.*, "ICBS: Improved conflict-based search algorithm for multi-agent pathfinding," in *Proc. 24th Int. Joint Conf. Artif. Intell.*, 2015, pp. 740–746.

[14] A. Andreychuk, K. Yakovlev, D. Atzmon, and R. Stern, "Multi-agent pathfinding (MAPF) with continuous time," *IJCAI Inter. Joint Conf. Artifi. Intell.*, pp. 39–45, 2019.

[15] A. Felner *et al.*, "Adding heuristics to conflict-based search for multi-agent path finding," in *Proc. 28th Int. Conf. Automated Plan. Scheduling*, 2018, pp. 83–87.

[16] I. Solis, J. Motes, R. Sandström, and N. M. Amato, "Representation-optimal multi-robot motion planning using conflict-based search," *IEEE Robot. Automat. Lett.*, vol. 6, no. 3, pp. 4608–4615, Jul. 2021.

[17] H. Ma, J. Li, T. Kumar, and S. Koenig, "Lifelong multi-agent path finding for online pickup and delivery tasks," in *Proc. 16th Inter. Conf. Auton. Agents Multiagent Syst.*, 2017, pp. 837–845.

[18] J. Li, P. Surynek, A. Felner, H. Ma, T. S. Kumar, and S. Koenig, "Multi-agent path finding for large agents," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, no. 01, 2019, pp. 7627–7634.

[19] H. Ma, T. S. Kumar, and S. Koenig, "Multi-agent path finding with delay probabilities," in *Proc. AAAI Conf. Artif. Intell.*, vol. 31, no. 1, 2017, pp. 3605–3612.

[20] J. Motes, R. Sandström, H. Lee, S. Thomas, and N. M. Amato, "Multi-robot task and motion planning with subtask dependencies," in *Proc. IEEE Int. Conf. Robot. Automat.*, vol. 5, no. 2, 2020, pp. 3338–3345.

[21] S. J. Guy *et al.*, "Clearpath: Highly parallel collision avoidance for multi-agent simulation," in *Proc. ACM SIGGRAPH/Eurographics Symp. Comput. Animation*. New York, NY, USA: ACM, 2009, pp. 177–187.

[22] R. Nissim and R. I. Brafman, "Multi-agent a* for parallel and distributed systems," in *Proc. ICAPS Workshop Heuristics Search Domain-Independent Plan.*, 2012, pp. 43–51.

[23] D. Henrich, "Fast motion planning by parallel processing - a review," *J. Intell. Robot. Syst.*, vol. 20, no. 1, pp. 45–69, 1997.

[24] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant, "Meta-agent conflict-based search for optimal multi-agent path finding," *SoCS*, vol. 1, pp. 39–40, 2012.

[25] E. Boyrasky, A. Felner, G. Sharon, and R. Stern, "Don't split, try to work it out: Bypassing conflicts in multi-agent pathfinding," in *Proc. 25th Int. Conf. Automated Plan. Scheduling*, 2015, pp. 47–51.