Scalable Online Multi-Agent Path Planning: A Hybrid Method with Adaptive Scheduling and Conflict-Minimizing Conflict-Based Search

Qihao Shen[1], Dajun Guo[1], Katherine Ip[1], Guang Hu[1], Yangmengfei Xu[1], Chenyuan Zhang[1, 2]

[1]Faculty of Engineering and IT, the University of Melbourne
[2]Monash University
{qihaos, dajun.guo1, katherine.ip, ghu1,yangmengfeix}@student.unimelb.edu.au, chenyuan.zhang@monash.edu

Abstract

Multi-Agent Path Finding (MAPF) focuses on computing collision-free paths for multiple agents operating within shared environments. The League of Robot Runners (LoRR) competition challenges participants to develop coordination strategies for teams of autonomous robots in dynamic, grid-like settings.

To address the competition problem, we developed an integrated algorithm with distinct scheduling and planning phases. In the scheduling phase, we propose a task allocation strategy that accounts for both unassigned tasks and those already allocated but not yet initiated. Our approach supports dynamic task reallocation whenever an alternative agent can complete a task with a reduced makespan. This iterative process continues until all agents are assigned unique tasks or a predefined time limit is reached.

In the planning phase, we introduce an online variant of Conflict-Based Search (CBS). Unlike classical CBS, which guarantees optimality by expanding the lowest cost node at the expense of high computational overhead, our method employs a conflict-minimization strategy, prioritizing nodes with the fewest unresolved conflicts. This enables rapid generation of feasible solutions under tight runtime constraints, with the ability to incrementally improve plan quality.

Our algorithm does not rely on domain-specific tuning, allowing it to generalize across different maps. In the LoRR competition, our online CBS algorithm achieved fifth place on the planner leaderboard.

Building on this foundation, we further developed two enhanced variants: a PASS-based algorithm and a partitioned-graph approach, both of which demonstrated superior performance on large-scale maps in our experimental evaluations.

1 Introduction

Multi-Agent Path Finding (MAPF) is a fundamental problem in artificial intelligence and robotics, focusing on computing collision-free paths for multiple agents operating in shared environments. This problem has significant real-world applications, including warehouse automation, airport ground operations, and autonomous vehicle coordination. To advance research in this area, the League of Robot Runners (LoRR), sponsored by Amazon Robotics, hosts competitions that challenge participants to develop solutions for coordinating teams of autonomous robots in dynamic, grid-like environments. These competitions emphasize realistic constraints such as turn costs and online task arrivals, thereby promoting the development of scalable and practical solutions for automated logistics and warehouse automation. More specifically, addressing the online configuration of multi-agent path planning and its integration with task allocation requires a more efficient path planning component and an effective task allocation component to augment existing offline methods (Sharon et al. 2015).

In the scheduling phase, the objective is to produce an effective task allocation that facilitates shorter paths in the subsequent planning phase. In online multi-agent systems, tasks must be assigned efficiently within tight time constraints, and the complexity increases significantly with the number of tasks and agents. To address this challenge, we attempted to modify the greedy-assignment algorithm combined with an iterative reallocation mechanism. This approach preserves the speed of greedy assignment while improving the quality of suboptimal allocations in an online style. Specifically, an agent can be reassigned to a different task if it results in a lower completion cost. The aim is to ensure fast and efficient task allocation across all agents.

In the planning phase, the objective is to plan collision–free trajectories for a set of agents operating in a specific environment. A powerful foundation is Conflict-Based Search (Sharon et al. 2015), a hybrid coupled–decoupled algorithm that proceeds in two levels. At the low level, CBS computes an optimal path for each agent under the current set of constraints. At the high level, it detects conflicts between paths of the agents, adds new constraints for conflict-avoiding, and then re-invokes the low-level search to replan the affected agents. This process repeats until either all conflicts are resolved or a predefined time bound is reached. However, classical CBS is inherently offline and always expands the CT node with the lowest total path cost. While this approach guarantees the optimality of the final solution, it often fails to produce a feasible solution under tight time budgets.

To overcome this limitation, we introduce an online CBS variant that revises the node-selection heuristic and integrates dynamic subgoaling. Rather than ordering CT nodes by total cost, our method expands the node with the fewest unresolved conflicts at each step. Intuitively, this ”conflict-

minimization" strategy accelerates the discovery of a first feasible (conflict-free) solution, which can then be incrementally refined toward higher quality if more time is available. Moreover, to further improve performance on large-scale maps, we establish a dynamic subgoaling mechanism: For each CT node, we record the time step of its earliest conflict. We then select the one whose earliest conflict time is the latest, and designate this node as the subgoal. This ensures that we can always find a short-term conflict-free solution.

Building upon our online CBS algorithm, we also develop and evaluate two additional enhancements: PASS and Subgraph Partitioning (SP), which both can further optimize online CBS on large-scale maps. Our experimental results show that the online CBS variant and additional enhancements can outperform the default Priority Inheritance with Backtracking (PIBT) algorithm provided by the League of Robot Runners.

In summary, we propose a novel task allocation strategy coupled with a CBS variant that performs planning following task assignment. Notably, unlike many competition-oriented approaches, our algorithm does not incorporate any optimization tailored to the competition maps, highlighting its potential for broader generalizability across diverse environments.

2 Preprocessing Heuristic Table

In this section, we first describe how the layout information is processed during the preprocessing stage of the competition. Specifically, we construct a heuristic table that serves as a shared resource for multiple modules within our framework.

The heuristic table H calculates and stores the shortest actual map distances from any source location to any target location. It does this efficiently using a lazy-initialized breadth-first search for each potential target. Initially, the heuristic table is initialized by iterating through all possible locations on the map. However, the state space in this competition is determined not only by the location of each agent, but also by the direction of each agent. Thus, we additionally experimented with a more accurate distance estimation that includes turning costs in the calculations. That is, a heuristic lookup table for the pair of state tuple (location and direction).

This preprocessing procedure is used for both the scheduling phase and the planning phase in our final submission to the competition. Other preprocessing procedure attempts have been made after the competition to optimize the planning phase, such as Partitioning, Heating Mapping, which are introduced in Section 4.2 and Section 6.

3 Scheduling Phase

In the scheduling phase, we consider a set of free tasks T that consists of all unassigned and unopened tasks and a set of free agents Agt that consists of agents without assigned tasks and agents with unopened tasks. We assign each free agent to a free task with the minimum makespan to that agent. The minimum makespan is estimated based on the actual map distance between the agent and the task, including the cost of turning actions. A look-up table H that was prepared in the pre-processing stage (See 2). Another look-up table M stores the lowest makespan to complete each of the tasks. In the situation where two agents are being assigned to the same task, the agent with the lower makespan will keep the task. The other agent will be assigned to the task with the lowest makespan in the remaining tasks. The process continues until each agent is assigned a unique task or the time limit exceeds. The proposed task schedule is stored in a list P with length equal to the total number of agents in the environment.

Algorithm 1: Task Scheduling

Input : env, Agt, T, H, M, P
Output : P

while $Agt \neq \emptyset$ **do**
  $a \leftarrow A.next\_agent()$;
  $best\_task \leftarrow null$;
  $best\_makespan \leftarrow \infty$;
  for $t \in T$ **do**
    $curr\_makespan \leftarrow H[t.\text{loc}][a.\text{loc}]$;
    if $curr\_makespan < best\_makespan$ **then**
      if *P.find(t) and* $curr\_makespan > L[t]$ **then**
        continue;
      $best\_task \leftarrow t$;
      $best\_makespan \leftarrow curr\_makespan$;
  if $P.find(best\_task)$ **then**
    $b \leftarrow P.get\_index(best\_task)$;
    $A.add(b)$;
  $P[a] \leftarrow best\_task$;
  $L[best\_task] \leftarrow best\_makespan$;
return P;

4 Planning Phase

The planning phase is developed based on the Conflict-Based Search (CBS) algorithm. Two different versions, including the submitted version for the competition and the newly developed version, are presented in this section.

4.1 Submitted Online CBS

This part details the planning algorithm we submitted to the competition, where it secured fifth place on the final leaderboard.

The route for each agent is planned based on a Conflict-Based Search (CBS) algorithm which takes the environment env and pre-goal node p_g_n as input and generates the action A for each agent. However, the original CBS is an offline algorithm which doesn't fit the needs in this application. Thus, we developed an online version of it. Similarly to the traditional CBS, our online CBS also contains two levels: a low-level planner to plan a path for each agent with constraints; and, a high-level validator to detect the conflicts and

strategically call the low-level planner to replan with newly added constraints. The low-level planner used in this work is the Weighted Constrained A* (WCA*). The whole process of the proposed online CBS is presented in Algorithm 2.

Algorithm 2: Online CBS

Input : env, p_g_n
Output : A
Initialize: $open \leftarrow \{\}, root\_h \leftarrow \{\}$
1 Extract each agent's $(start, goal, pre\_action)$ from env and p_g_n;
2 **if** $current\_timestamp > 0$ **then**
3 $\quad root\_h \leftarrow$ carry-over from previous timestamp;
4 **for** *each agent* **do**
5 $\quad$ **if** $agent.pre\_action = root\_h[\text{agent}].action[0]$ ***and*** $agent.start = root\_h[\text{agent}].path[1]$ **then**
6 $\quad\quad root\_h[\text{agent}].path.remove\_first()$;
7 $\quad\quad root\_h[\text{agent}].action.remove\_first()$;
8 $\quad$ **else**
9 $\quad\quad$ **if** $start = goal$ **then**
10 $\quad\quad\quad path \leftarrow \{\}, \quad action \leftarrow \{\text{Wait}\}$;
11 $\quad\quad\quad root\_h[\text{agent}] \leftarrow \{path, action\}$;
12 $\quad\quad$ **else**
13 $\quad\quad\quad root\_h[\text{agent}] \leftarrow$ WCA*$(env, start, goal, \{\}, \{\})$;
14 $open$.add$(root_h)$;
15 $found \leftarrow$ false;
16 $goal\_node \leftarrow root\_h$;
17 $sub\_goal \leftarrow root\_h$;
18 farthest timestamp $t^* \leftarrow 0$;
19 **while** $open \neq \emptyset$ **do**
20 $\quad n \leftarrow open$.pop();
21 $\quad c \leftarrow$ get_earliest_conflict_constraint(n);
22 $\quad$ **if** $n.t_{earliest} > t^*$ **then**
23 $\quad\quad t^* \leftarrow n.t_{earliest}, \quad sub\_goal \leftarrow n$;
24 $\quad$ **if** n.num_conflict $= 0$ **then**
25 $\quad\quad$ **if** $\neg found$ ***or*** $n.\text{cost} < goal\_node.\text{cost}$ **then**
26 $\quad\quad\quad goal\_node \leftarrow n, \quad found \leftarrow$ true;
27 $\quad (p_i, p_j, t) \leftarrow c$;
28 $\quad n_i \leftarrow n$ with c added for agent i;
29 $\quad n_j \leftarrow n$ with c added for agent j;
30 $\quad n_i.\text{path}[i] \leftarrow$ WCA*$(env, start_i, goal_i, n_i.\text{V}_{\text{con}}, n_i.\text{E}_{\text{con}})$;
31 $\quad n_j.\text{path}[j] \leftarrow$ WCA*$(env, start_j, goal_j, n_j.\text{V}_{\text{con}}, n_j.\text{E}_{\text{con}})$;
32 $\quad open.\text{push}(n_i); \quad open.\text{push}(n_j)$;
33 **if** $\neg found$ **then**
34 $\quad goal\_node \leftarrow sub\_goal$;
35 Extract action A from $goal\_node$;
36 $p\_g\_n \leftarrow goal\_node$;
37 **return** A;

At the beginning of the processing at each timestamp, the online CBS will generate two empty lists, an open list $open$ and a root $root\_h$, and enter a pre-check process (line 2 to 13). At $timestamp = 0$, the low-level planner WCA* will be called to generate an initial solution and save to $root_h$ as the root node for later processing. At the later timestamp, once the online CBS is run, it will first verify and synchronize each agent's state between the carried-over node from the previous timestamp and the agent's current actual state. If an agent has completed its original task and been assigned a new task, which results in the absence of a feasible path, the low-level algorithm will be called again to compute a new initial solution for that agent.

High level In the high-level phase (Line 19-32 in Algorithm 2), the algorithm, guided by a heuristic function, keeps exploring the open list, which could potentially lead to a better solution. While traditional CBS employs the path cost as its heuristic, the online version must return a feasible solution (conflict-free next moves for all agents) within 1 second or even less. The original heuristic struggles to quickly yield feasible solutions in crowded multi-agent scenarios.

To solve this, instead of sorting the nodes by their costs, we prioritize those with fewer conflicts. With this "greedy" heuristic in the high-level search, a conflict-free solution can be retrieved quickly. When all conflicts cannot be solved within 1 second, we return the first actions of a solution that contains the longest step before reaching its first conflict. Once a feasible solution is found, the algorithm will utilize any remaining time to refine it further for optimality.

When popping a node from the open list, we first check for conflicts. If any exist, we prioritize resolving the conflict occurring at the earliest timestamp.

During the conflict detection phase, a conflict C is represented as a tuple (p_i, p_j, t), where p_i and p_j denote the respective states (including position and orientation) of agents i and j at the time the conflict occurs, and t indicates the timestamp at which the conflict happens. Once a conflict is identified, the system resolves it by generating two new child nodes. In each child node, a constraint is added for one of the conflicting agents: in the first child node, agent i is prohibited from occupying position p_i at timestamp t, while in the second child node, agent j is prohibited from occupying position p_j at the same timestamp. These newly constrained nodes are then passed to the low-level planner to update the paths.

Weighted Constrained A* As a preliminary attempt, we employ Weighted Constrained A^* (Algorithm 3) as our path-planning algorithm for individual agents to retrieve a conflict-free path efficiently. The weight factor w is set to 1.7, and the heuristic function used is $manhattan\ distance$ in our setting.

4.2 Additional techniques with online CBS

Now we will describe the additional modules we introduced after the competition for further enhance the performance of our algorithm.

PASS This is an additional technology that can enhance our online CBS. In the PASS phase (Algorithm 4), we examine the remaining nodes in the open list. For each node, if

Algorithm 3: Weighted Constrained A* (WCA*)

Input : env, $start$, $goal$, V_{con}, E_{con}
Output : $path$, $action$
Initialize: $closed \leftarrow V_{con}$
weight $w \leftarrow 1.7$
$h \leftarrow get\_heuristic(env, start, goal)$
$root.path \leftarrow \{\}$
$root.loc \leftarrow start$
$root.action \leftarrow \{\}$
$open$.push($root$)

while $open \neq \emptyset$ **do**
  $curr \leftarrow open$.pop();
  if $curr \in closed$ **then**
    continue;
  $closed$.push($curr$);
  if $curr$.loc $= goal$ **then**
    return ($curr$.path, $curr$.action[0]);
  for $next \in getNeighborLocs(curr$.loc$)$ **do**
    $action \leftarrow getAction(curr$.loc, $next)$;
    $path' \leftarrow curr$.path $\cup \{next\}$;
    $action' \leftarrow curr$.action $\cup \{action\}$;
    if *violatesConstraints*$(path', E_{con})$ **then**
      continue;
    $h \leftarrow get\_heuristic(env, next, goal)$;
    $new\_node \leftarrow (curr, w \cdot h)$;
    $open$.push(new_node);
return ($root$.path, $\{Wait\}$);

every agent's next action matches the corresponding action in our candidate feasible solution (to be returned), we then:

1. Trim the current timestamp from the node to align it with the next timestamp's state;
2. Store the modified node in a new open list;
3. Pass this updated open list to the subsequent timestamps;

Algorithm 4: PASS

Input : open list $open$, goal node $goal\_node$
Output : $next\_open$
Initialize: $next\_open \leftarrow \{\}$

while $open \neq \emptyset$ **do**
  $node \leftarrow open$.pop();
  foreach *agent* i *in* $node$ **do**
    if
      $node[i].action[1] = goal\_node[i].action[1]$
      and $|node[i].action| \geq 2$ **then**
      $next\_open$.push($node$);
return $next\_open$;

Subgraph Partitioning Since we need to efficiently find feasible solutions on large maps, in addition to using Weighted A^*, we developed a subgraph partitioning (SP) algorithm during the preprocessing phase. This algorithm aims to divide the large map into multiple smaller maps, allowing the WCA* to find a solution much faster. This is done by allowing the agent to find a constrained path towards each border between the small sections that the agent needs to travel in a greedy manner, instead of finding a constrained path to the end.

The subgraph partitioning algorithm is divided into two parts: finding the centroids and generating the Voronoi diagram.

When selecting the centroids, we employ a Max-Min Sampling strategy (Algorithm 5) to choose k points from a set of points. The goal is to ensure that these k points are as dispersed as possible in the space, maximizing the minimum distance between any two points. The core logic of this strategy is that, in each iteration, the newly selected point is the one that is farthest from all the currently selected points.

To be specific, in the initial stage, we maintain a set called $selected$ to determine whether a point has been chosen, and a set $interest\_points$ to store the selected centroids. If we want to select k points, in each of the k iterations, we will traverse all the possible unselected points p, compute the distance between p and each point in $interest\_points$, and record the shortest distance as min_dist. Finally, we select the point p with the largest min_dist in this iteration and add it to $interest\_points$. After all k iterations, we return

Algorithm 5: Max-Min Sampling

Input : Set of points $points$, Environment env,
Number of centroids k
Output : Selected centroids $interest\_points$
Initialize: $interest\_points \leftarrow \{\}$,
$selected \leftarrow$ array of size $|points|$
initialized to $false$,
Randomly select a point p from $points$,
$interest\_points$.push(p)
$selected[p] \leftarrow true$.

for $i \leftarrow 1$ **to** $k - 1$ **do**
  $max\_min\_dist \leftarrow -1$
  $selected\_idx \leftarrow -1$
  foreach $j \in \{0, 1, 2, \ldots, |points| - 1\}$ **do**
    if *not* $selected[j]$ **then**
      $min\_dist \leftarrow \infty$
      foreach $p \in interest\_points$ **do**
        $dist \leftarrow$ distance$(env, points[j], p)$
        if $dist < min\_dist$ **then**
          $min\_dist \leftarrow dist$
      if $min\_dist > max\_min\_dist$ **then**
        $max\_min\_dist \leftarrow min\_dist$
        $selected\_idx \leftarrow j$
  if $selected\_idx \neq -1$ **then**
    $interest\_points$.push($points[selected\_idx]$)
    $selected[selected\_idx] \leftarrow true$
return: $interest\_points$;

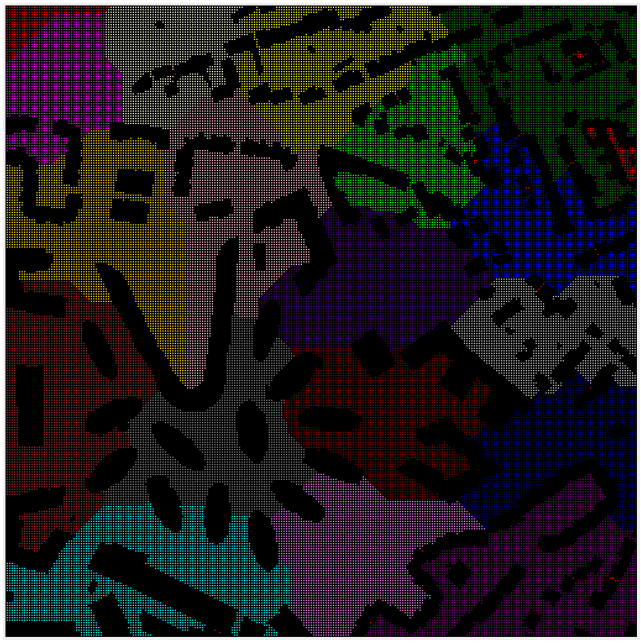

Figure 1: City map by Subgraph Partition

Algorithm 6: Generate Voronoi Diagram

Input : Environment env, Number of rows $rows$, Number of columns $cols$, Set of centroids $centroids$
Output : Voronoi Map $voronoi\_map$
Initialize: $map \leftarrow env.map$
$voronoi\_map \leftarrow$ array of size $rows \times cols$ initialized to -1

```
for i ← 0 to rows − 1 do
    for j ← 0 to cols − 1 do
        if map[i × cols + j] = 0 then
            min_dist ← ∞
            label ← −1
            for k ← 0 to |centroids| − 1 do
                dist ← distance(env, i × cols + j, centroids[k].x × cols + centroids[k].y)
                if dist < min_dist then
                    min_dist ← dist
                    label ← k
            voronoi_map[i × cols + j] ← label
return: voronoi_map;
```

$interest\_points$.

The Voronoi diagram (Choset and Burdick 1995) is an algorithm that divides the plane into several regions, where any point within a region is closer to its designated centroid than to any other centroids.

We initialize a $voronoi\_map$ of size $rows * cols$, with all values set to -1, representing unassigned regions (Algorithm 6). Then, we iterate through each cell, and only if the cell value is 0 (indicating it is valid), we proceed to find the nearest centroid. For each unassigned cell, we use the $distance$ function to compute its distance to all centroids. If the distance is smaller than the current minimum distance, we update min_dist and the corresponding centroid index. Next, we store this centroid's index in the corresponding position of $voronoi\_map$. When the loop finishes, we return the complete $voronoi\_map$, where each point is labeled with the index of its nearest centroid.

5 Evaluation

The algorithm's performance is evaluated under three maps with default settings: `random-32-32-20` (`Random` map with 100 agents), `brc202d_500` (`Game` map with 500 agents), and `paris_1_256_250` (`Random` map with 250 agents). Performance is measured by the total number of tasks completed within $5,000$ timestamps.

5.1 Scheduling Phase

The evaluation was conducted using the default PIBT planning algorithm and the default heuristic table. In addition, we tested our approach with a modified heuristic that incorporates both turning cost and map distance, providing a more accurate estimate of traversal cost. The results are compared against the competition's default scheduling algorithm, which uses a greedy task allocation strategy without task reallocation.

	Default	Reallocation	Modified Heuristic
random	8405	10035	10029
city	3556	4188	4309
game	3613	4679	4786

Table 1: Experiment results for scheduling phase

The results in Table 1 show that task reallocation significantly improves the performance of the greedy scheduling algorithm. Allowing agents to reallocate tasks enables better matching between agents and tasks over time. However, using a more accurate heuristic that considers turning actions does not lead to a significant performance improvement. This suggests that actual map distance alone is already a sufficiently effective heuristic for task scheduling in this setting.

5.2 Planning Phase

	default	Online CBS	PASS	SP
random	8405	9909	9287	9853
city	3556	4081	4109	4126
game	3613	×	×	3943

Table 2: Experiment results for planning phase, where PASS is the online CBS with the PASS phase, and SP is the online CBS with both PASS and Subgraph Partitioning phases.

From experiment results in table 2, we can observe that our uploaded online CBS outperforms the default PIBT algorithm on both the random and city maps, which validates that our algorithm performs as expected. On the random map, the results of PASS are clearly worse than those of online CBS and SP. This may be because, during the pass phase, we did not estimate the quality of the nodes that are passed to the next timestamp. As a result, we pass all nodes with valid actions to the next timestamp without filtering. If the next timestamp has enough time to resolve all the nodes, this approach can naturally approach the optimal solution. However, if congestion occurs at the next timestamp and we have not applied a cost threshold to the passed nodes, it can delay the time it takes for the high-level search to find a feasible solution or improve the solution quality.

On the city map, the result of PASS is comparable to that of online CBS. This may be because the city map is larger than the random map, so an agent requires more timestamps to complete its task, giving us more time to approach an optimal solution. In contrast, the random map is smaller, and agents quickly complete their tasks and move on to the next one, which amplifies the disadvantage of PASS.

Subgraph partition performs well on both the random and city maps. On the random map, we set the number of center points $k = 1$, making it essentially the same as online CBS. Minor differences in results may stem from tie-breaking in A*, where the search might randomly choose paths that lead to more congestion later on. This is a potential direction for future improvement. On the city map, we set $k = 100$, so with a larger map, we have more time to find better solutions.

However, on the large map (game), our online CBS frequently fails, making it difficult to obtain a good solution. In contrast, the default algorithm has a mechanism that forces agents to stop when conflicts occur, which helps maintain feasibility even under congestion. As a result, it can still provide a decent solution on large maps. Meanwhile, subgraph partition outperforms the default algorithm on the game map, demonstrating that our algorithm design can effectively plan paths and obtain feasible solutions in large-scale scenarios.

6 Future Directions

While we have introduced a range of techniques in this report, many remain in the early stages of development and present opportunities for further refinement and enhancement. In particular, our current algorithm was unable to handle the warehouse scenario within the competition's time constraints (1 second) due to the large number of agents, despite significant optimizations to convert the original CBS into an online version. This limitation is partly because we did not incorporate any domain-specific optimizations that exploit the symmetric structure of warehouse environments. Future work should focus on further improving efficiency while remaining agnostic to layout-specific information.

For the scheduling phase, a direction for future improvement is to introduce a task queue that allows each agent to maintain a list of candidate tasks, sorted by increasing estimated cost. Instead of evaluating all tasks at every timestamp, agents can reference their precomputed queues to make faster and more efficient allocation decisions. These queues would be updated periodically to reflect changes in task availability and agent positions. This approach can potentially reduce computational overhead by decoupling cost estimation from task allocation, thereby enabling more scalable and responsive scheduling.

In the current planning phase, our high-level search adopts a simple greedy heuristic when expanding nodes: nodes with fewer conflicts are given higher priority, based on the assumption that resolving low-conflict nodes first will accelerate progress toward a feasible solution. While this approach is intuitive and lightweight, it lacks deeper consideration of the cost to solve each conflict. As a direction for future improvement, we propose to incorporate ideas from LRTA*(k) (Learning Real-Time A*), as introduced by Hernández and Meseguer (2005). Instead of relying solely on static conflict counts, we intend to dynamically update the cost of each node in the open list during every iteration. This includes not only the immediate cost-to-go estimate but also predictions about possible future constraints the agent may encounter.

By estimating the cost of the potential conflict-free solution of a search node, we can make more informed decisions during high-level node expansion. Such a framework has the potential to significantly improve the planner's efficiency in exploring promising (efficient) feasible solutions earlier, thereby moving toward globally better solutions faster.

Besides the high-level CBS, the low-level search to find a constraint-free path can also be optimized. Instead of `WCA*`, some other search algorithm can be explored, such as Focal Search (Barer et al. 2014). In addition, the integration of the Subgraph Partitioning can also be improved by generating optimal paths between the subgraphs in the preprocessing phase using Graphs of Convex Sets.

Another potential future direction is to use the preprocessing time to examine the map and generate some "high-way" for the agents to move from one area to another in a large map. Since agents on the high-way are following the same direction, there would be no conflict for these agents. The planning algorithm just needs to take care of those agents that are not on the high-way, which would significantly reduce the search time. The idea to generate this high-way is by the frequency of a state that appears in the optimal solutions for any other state pairs. This idea is aligned with the betweenness centrality (Brandes 2001). An example of the generated heatmap for the `Game` map is provided in Figure 2.

7 Conclusion

In conclusion, we presented the algorithm developed for our participation in the LoRR competition. In the scheduling phase, we introduced a dynamic task allocation strategy capable of assigning and reassigning tasks to agents in real time. For the planning phase, we proposed an online variant of Conflict-Based Search (CBS) that prioritizes nodes with the fewest unresolved conflicts to efficiently generate feasible solutions. We further extended this planner with two enhanced variants: a PASS-based approach and a partitioned-graph-based method. Both competition results and experi-

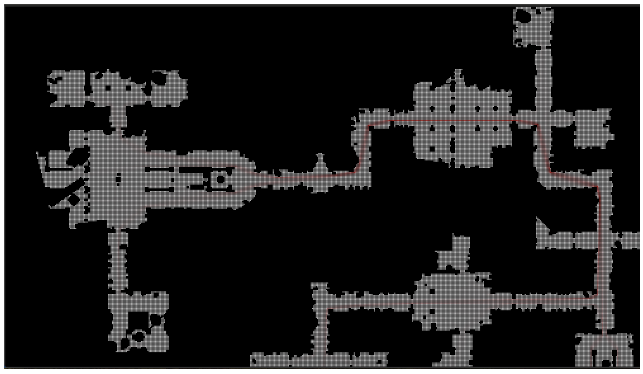

Figure 2: Partial of the betweenness centrality heatmap of Game map.

mental evaluations demonstrate the strong performance and generalizability of our approach in the absence of layout-specific fine-tuning.

References

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the international symposium on combinatorial Search*, volume 5, 19–27.

Brandes, U. 2001. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2): 163–177.

Choset, H.; and Burdick, J. 1995. Sensor based planning. I. The generalized Voronoi graph. In *Proceedings of 1995 IEEE international conference on robotics and automation*, volume 2, 1649–1655. IEEE.

Hernández, C.; and Meseguer, P. 2005. LRTA*(k). In *Proceedings of the 19th international joint conference on Artificial intelligence*, 1238–1243.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial intelligence*, 219: 40–66.