machines

MDPI

Article

Clustered Reverse Resumable A* Algorithm for Warehouse Robot Pathfinding

Gábor Csányi and László Z. Varga *

Faculty of Informatics, ELTE Eötvös Loránd University, Pázmány Péter Sétány 1/C, H-1117 Budapest, Hungary
* Correspondence: lzvarga@inf.elte.hu

Abstract

Robots are widely used to carry goods in automated warehouses. Planning collision-free paths for multiple robots which are continuously given new goals is called Lifelong Multi-Agent Pathfinding. In a lifelong environment, conflicts may emerge among the robots, and continuous replanning is needed. We propose, develop, implement, and evaluate the novel approach called the Clustered Reverse Resumable A* (CRRA*) algorithm to enhance the continuous computation of the shortest path from the changing position of a robot to its goal. The Priority Inheritance with Backtracking (PIBT) algorithm is the currently known most efficient algorithm to handle the pathfinding of thousands of robots in a warehouse. The PIBT algorithm requires that in each step each robot evaluates the distances from its surrounding positions to its goal; therefore, we integrate the CRRA* algorithm with the PIBT algorithm to evaluate CRRA*. The evaluation results show that the CRRA* leads to a significant reduction in computation time, especially in larger warehouses where the obstacles form well-separated spaces. At the same time, the degradation in solution quality is minimal. The CRRA* algorithm is more efficient in larger warehouses than the plain Reverse Resumable A* (RRA*) algorithm. The faster computation of slightly suboptimal paths can be useful in many practical applications, especially in situations where real-time planning is more important than finding the optimal paths. CRRA* can also be used as a heuristic in any multi-agent pathfinding solution to obtain a faster, nearly accurate heuristic.

Keywords: multi-agent path planning; Lifelong Multi-Agent Pathfinding; path planning heuristics; hierarchical search; Reverse Resumable A*

check for updates

Academic Editors: Changwon Kim and Seong-Jin Kwon

Received: 22 October 2025
Revised: 1 December 2025
Accepted: 4 December 2025
Published: 8 December 2025

Citation: Csányi, G.; Varga, L.Z. Clustered Reverse Resumable A* Algorithm for Warehouse Robot Pathfinding. *Machines* **2025**, *13*, 1127. https://doi.org/10.3390/machines13121127

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The Multi-agent Pathfinding (MAPF) problem [1] comprises planning collision-free paths for multiple robots on a given map, which is filled with obstacles. The goal is for each robot to move from its starting position to its designated target in the shortest possible time without colliding with other robots or obstacles. This problem is crucial in many industrial applications, especially in automated warehouses, where reducing human resources can lead to significant efficiency gains and cost reductions [2]. Founded in 2003, Kiva Systems revolutionized warehouses by having robots bring products to workers instead of workers walking to them. In 2012, Amazon acquired the company for USD 775 million. MAPF can also be used to plan the towing of planes at crowded airports [3] or in video games where a team of characters must be controlled simultaneously [4].

Lifelong Multi-Agent Pathfinding (LMAPF) is a more realistic version of the MAPF problem, in which robots are continuously given new goals as they complete the previous one. For example, in an automated warehouse, a robot is immediately given a new task

after delivering a product to a designated location. In most research, robots are treated as point-like objects that can freely move on a grid map to any adjacent square regardless of the direction they are facing. However, this assumption does not reflect reality, as the movement of the robots often requires them to turn in the direction of their forward movement. This is the Lifelong Multi-Agent Pathfinding with Turns (LMAPF-T) problem.

One of the main difficulties of the LMAPF-T problem is that all new goals are not known in advance but are always revealed only after a robot has completed its current task. This is a significant challenge compared to the traditional "one-shot" MAPF problem, where the goal of each robot is initially known, so the entire path planning can be performed in a single step. However, in a lifelong environment, robots reach their goals at different times, which results in the need for continuous replanning. This can lead to conflicts, as robots that receive new goals can intersect or block already planned paths as described in [5]. There are several options for handling such conflicts [6]. The robot remains in place until the desired path becomes free. This is a simple solution, but it can significantly reduce the throughput of the system [7]. All viable approaches require that the path of the robots is frequently recalculated. Instead of recalculation from scratch, the incremental solution of the Reverse Resumable A* (RRA*) algorithm [8] can be used. However, the RRA* approach needs a lot of resources on larger maps and greater number of robots. In this paper we propose, develop, implement, and evaluate the novel approach called Clustered Reverse Resumable A* (CRRA*) algorithm to enhance this computation. Although we target the LMAPF-T problem, the CRRA* algorithm can be used in any other problem where a potential field-like [9] approach is needed, and a robot has to know the shortest paths towards its destination from the surroundings of its current position.

The most recent algorithm to handle the conflict resolution and deadlock avoidance issues in the LMAPF problem is Priority Inheritance with Backtracking (PIBT) [10]. PIBT does not plan the entire path but only determines the next movement at each step, which scales well even for many robots. In PIBT, for each timestep, each robot has to evaluate distances from the surrounding nodes to its goal. The original PIBT saves computation time by preparing distance tables from each robot's goal to every possible location using breadth-first search from the goals. This has considerable computational cost. In order to empirically evaluate the CRRA* algorithm in LMPAF-T problems, we integrate CRRA* into PIBT to enhance this computation with the incremental and clustered approach of CRRA*. The integration requires that we solve several issues, including the correction of the originally published RRA* algorithm.

We empirically evaluate this solution in the realistic scenarios of The League of Robot Runners Competition (https://www.leagueofrobotrunners.org/ (accessed on 30 November 2025)) supported by Amazon Robotics. In the evaluation, we measure the performance of the PIBT algorithm using the CRRA* algorithm with different parameters, and compare the results to the results when the plain RRA* algorithm is used in PIBT.

The rest of the paper is organized as follows. In Section 2, we formally define the LMAPF-T problem, then we discuss related work in Section 3. We present the Clustered Reverse Resumable A* (CRRA*) algorithm in Section 4. In Section 5, we make analytical considerations. In order to be able to evaluate the CRRA* algorithm empirically, we made several additional developments, which are presented in Section 6, including the correction of the Reverse Resumable A* (RRA*) algorithm. In Section 7, we empirically evaluate the CRRA* algorithm and demonstrate when it performs better than the classic RRA* algorithm. Section 8 concludes the paper.

2. Problem Definition

MAPF is a large family of problems, and many variations exist [1]. In this paper, we use the following definition.

The input to LMAPF-T is a grid type of graph $G = (V, E)$, where V is the set of nodes, E is the set of edges, and n robots $A = \{a_1, a_2, ..., a_n\}$, and a set of obstacles. The position of an obstacle can be any node $v \in V$. The position of a robot consists of a node $v \in V$ and an orientation $o \in \{North, South, East, West\}$. The obstacles and the robots cannot be positioned on the same node $v \in V$. Each node $v \in V$ is connected with one edge to at most one node in each orientation. Each robot $a_i \in A$ has a starting node $s_i \in V$ with a starting orientation o_i and an initial goal node $g_i \in V$. The orientation at the goal node is not specified because any orientation is acceptable. Once a robot has reached its current goal, it receives a new one, and the reached goal becomes the new starting point. At each step, a robot either moves forward to an adjacent node in the direction of its orientation, or turns 90 degrees clockwise or counterclockwise, or stays in place. A collision occurs when two robots enter the same node or traverse the same edge at the same time. Let $l_i(t) \in V$ denote the location of agent a_i in discrete timestep t. Agent a_i starts in its initial location $l_i(0) = s_i$. At each step, an agent can move to the adjacent vertex in accordance with the orientation of the agent or wait at its current vertex, that is, $l_i(t+1) = l_i(t)$ or $(l_i(t); l_i(t+1)) \in E$. The agents have to avoid collisions when two agents move to the same vertex or traverse the same edge at the same step. Formally, two agents cannot be in the same location in the same timestep, that is, $\forall a_i, a_j \in A : a_i \neq a_j \Rightarrow \forall t : l_i(t) \neq l_j(t)$, and two agents cannot move along the same edge in opposite directions in the same timestep, that is, $\forall a_i, a_j \in A : a_i \neq a_j \Rightarrow \forall t : l_i(t) \neq l_j(t+1) \vee l_i(t+1) \neq l_j(t)$. The goal is for the robots to reach as many target nodes as possible within a given time, following a collision-free path.

3. Related Work

In this section we discuss MAPF problem solving algorithms to see their requirements for calculating the path of the agents.

Algorithms developed to solve MAPF problems can be divided into several categories, each with its own advantages and disadvantages [11]. They may be suitable for different application areas depending on whether the optimality of the solution or calculation speed is the more important consideration.

Optimal algorithms: These are complete algorithms. If a solution exists, they are guaranteed to find the best one. However, due to their computational complexity, they are extremely slow for larger problems, as finding an optimal solution is NP-hard [12]. There are search-based approaches [13] and compiling-based approaches which reduce MAPF to well-known problems [14–16].

Bounded suboptimal algorithms: They ensure that the result deviates from the optimal by at most a predefined limit but can be faster than optimal algorithms. They are still not fast enough for real-time applications on large problems. See for example [17].

Unbounded suboptimal algorithms: They can solve large problems quickly, but the completeness and the quality of a solution are not guaranteed. These algorithms can be applied in practice for large applications, and we are interested in such applications. The algorithms we discuss below fall into this category.

*3.1. Cooperative A**

Cooperative A* (CA*) [8] is a basic algorithm for solving MAPF problems, which plans the path of each robot sequentially, taking into account the movements of the previously planned robots. To do this, it uses a reservation table, which records which cells a given

robot is in at which time. Later planned robots must avoid these cells, thus ensuring that collisions are avoided.

One of the main disadvantages of the algorithm is that it is very sensitive to the planning order of the agents. An unfavorable order may result in no valid path for a robot, while a better arrangement may result in a more efficient solution. In lifelong scenarios, the algorithm is very sensitive to the emerging conflicts described in [5]. In order to resolve the conflicts, the path of the agents have to be replanned frequently. The path of each robot is planned using the A* algorithm, using Manhattan distance as a heuristic, which does not always give an accurate estimate.

3.2. Priority Inheritance with Backtracking

Priority Inheritance with Backtracking (PIBT) [10] is a suboptimal algorithm that does not guarantee the quality of the solution. Instead of calculating the entire path of the agents in advance, it determines the next moves step by step. With this approach, the PIBT algorithm can handle the emerging conflicts in LMAPF-T; however, the algorithm requires that the robot always knows the distance between each node in its surrounding and the goal location of the robot.

The algorithm works on the basis of the priority order, according to which the agents plan their next move in turn. In the single-shot MAPF problem, the PIBT algorithm might calculate the shortest paths between the start and the goal locations in advance, and then this can be used during the movement of the robots. A robot always chooses from the neighboring locations, preferring the one that is closer to its goal. If another robot already wants to move to the location in a given step, it ignores it because the robot moving there has a higher priority, and its move was planned before. If no one is planning to move to the selected location in a given step and no one is currently staying there, then the robot can move there freely. Otherwise—if no one is planning to move there yet but another robot was already there in the previous step—the robot that is there must also look for a new goal location, inheriting the priority of the original robot, thus ensuring the path of the robot with the higher priority. If a robot cannot find a free move, it signals this to the robot who wants to move in its place, so it steps back and looks for an alternative route. Proper priority assignment ensures that each robot will eventually reach its goal, provided that each vertex of the search graph is part of a cycle, guaranteeing that no agent will reach a dead end.

Due to its low complexity, PIBT finds solutions much faster for larger maps and multiple robots than many other algorithms. However, it does not guarantee the quality of the solution, but with a low robot density it is not much behind other solutions.

*3.3. Hierarchical Pathfinding A**

Hierarchical Pathfinding A* (HPA*) [18] is an efficient pathfinding method between a start and a goal location, with some similarity to the approach presented in this paper. It can achieve up to a 10x speedup compared to an A* search, while maintaining a path length close to the optimal one. Although HPA* is a single agent pathfinding method, we consider it here because we will have to frequently replan the shortest path of the individual robots in LMAPF-T and we need an efficient approach.

The algorithm divides the map into smaller, square clusters, and then defines crossing points on their boundaries, creating an abstract graph. During the search, the start and destination points are inserted into this abstract graph, connecting them with the nearest cluster boundaries.

Route planning is performed on the resulting abstract graph using the A* algorithm, which contains significantly fewer vertices than the original detailed graph. Optimal routes

between cluster boundaries can be calculated in advance, so the search in the original graph only needs to be performed between the starting point and its cluster boundary, and between the destination point and its cluster boundary.

This method significantly reduces the size of the search space and speeds up route planning; however, it cannot be applied directly to multiple agents because there are other agents on the map, and they are moving obstacles.

Game developers also use a similar technique, where the map is divided into convex polygons (https://docs.unity3d.com/Packages/com.unity.ai.navigation@2.0/manual/NavInnerWorkings.html (accessed on 30 November 2025)) and the closest one is assigned to the start and destination points, and then the search is performed at this abstract level. Once the series of polygons leading to the destination is available, the game character is always navigated towards the nearest visible corner.

*3.4. Hierarchical Cooperative A**

Hierarchical Cooperative A* (HCA*) [8] attempts to address the heuristic problem of CA* by using a Reverse Resumable A* (RRA*) algorithm to estimate the remaining distance instead of the Manhattan distance. RRA* is a modified A* search that starts backwards from a goal and does not terminate once the robot's current position is reached, and can be continued until a given vertex is expanded. This is advantageous because the shortest path calculated by the A* algorithm is often not traversable due to other robots, so the robot is forced to deviate from it and has to compute the shortest path from its surrounding. Computing the shortest path from the surrounding cannot reuse the previous forward A*, and a new forward A* search would have to be started. Using RRA*, it is possible to determine much more quickly how far a given point is from the target by continuing the initial calculations, thus avoiding the full cost of re-searching.

*3.5. Windowed Hierarchical Cooperative A**

Windowed Hierarchical Cooperative A* (WHCA*) [8] is an improvement of CA* by using the heuristics of HCA* and by making reservations ahead only to a time window. The agents replan and re-reserve when they approach the end of their previous time window. This mitigates the negative effects of the sensitivity to the planning order of the agents in the CA* algorithm.

3.6. Applicability for the LMAPF-T Problem

The above algorithms work well for the MAPF problem. Due to the specifics of the LMAPF-T problem, where the targets are not known in advance, we cannot calculate the distances to each target in advance like in the MAPF problem because we would have to calculate the distances between every possible pair of vertices. On a 256×256 grid map with 40% of the area occupied by obstacles, storing the distance between each pair of vertices in 4-byte integers would require approximately 6.18 GB of memory: $(256 \times 256 \times 0.6)^2 \times$ 4 byte = 6,184,752, 906 byte $\approx$ 6.18 GB. This storage requirement can be halved since the distance from one point to another is the same as the other way around, but due to the consideration of rotations, each vertex would be represented in four different orientations, which would significantly increase the required memory size, not to mention larger maps.

To cope with the dynamic nature of the LMAPF-T problem, we could use the A* algorithm to dynamically determine at each step which of the neighboring squares is closest to the goal, and where it is worth moving next. However, this would require multiple A* runs per robot, which is computationally inefficient. A better option is to use the RRA* algorithm, which searches for the shortest path back from the goal to the current position, and can reuse the calculations made so far to extend to an arbitrary vertex. However, this can also be time consuming for a larger number of robots and a larger

map; therefore, we propose the novel Clustered Reverse Resumable A* (CRRA*) algorithm which combines RRA* with the clustered nature of HPA*. With this combination, the CRRA* can be applied to larger LMAPF-T problems in real-time scenarios compared to the RRA* algorithm.

4. Clustered Reverse Resumable A*

The running time of traditional RRA* increases as the map size increases and the target is further away. The bigger the distance between the goal and the location of the robot, the larger the search space, and the more vertices need to be expanded to reach the desired vertex. The distance has a one-dimension value and the search space has two dimensions, so the search space grows faster than the distance from the goal. If the robot has to deviate from its optimal path because of other robots, and the robot ends up in an area where the target distance is not yet known by RRA*, the search must be continued, which also takes more time if the target is further away.

CRRA* offers a solution to these problems. The essence of the method is that we define clusters, and we store which other clusters we can reach a cluster through. Given a map (V,E), we will create $\mathcal{C} = \{C_1, \ldots, C_k\}$ a set of clusters where $\mathcal{C}$ is a partition of V so that each C_i is a connected subset of vertices. To create the clusters, we will use $\mathcal{R} = \{R_1, \ldots, R_k\}$, a subset of the vertices, so that each R_i will be a particular vertex of C_i called its reference point. We first identify the reference points on the map, and then divide the map into clusters by iteratively expanding the reference points with the nodes closest to them until each point on the map belongs to a reference point.

In order to reduce the computation time of pathfinding for a larger number of robots and on a larger map, it is worth combining preliminary and on-the-fly calculations. If we keep the number of reference points under a limit, then we can calculate the shortest path between every pair of reference points preliminarily, and store this information. We use the clusters to determine the path from a starting point to the target by identifying the sequence of clusters $\langle C_{start}, \ldots, C_{goal} \rangle$ that the robot goes through to reach the goal location. We use the preliminarily computed distances between the reference point to determine this sequence of clusters.

Once the clusters and the above data structures are created, we can use them to calculate the shortest paths when needed. With this method, we do not need to perform a single large RRA* search when determining the robots' path. Instead, we break the task into several smaller RRA* searches, each starting from the next cluster to the current cluster in the sequence of the clusters. This is expected to significantly reduce the number of extended vertices, thus enabling faster computations. It does not necessarily guide the robots along the shortest paths but might be close to it. If we decrease the distance between the reference points, then we approach the case where the distance between every pair of vertices is calculated in advance. If we increase the distance between the reference points, then we approach the case of a single RRA* search starting from the target point. If there is only one reference point on the map, then we obtain the plane RRA* algorithm, which will be our reference in the evaluation.

4.1. Reference Points and Clusters

For the clustering, we use a similar approach to HPA* [18] that covers the map with disjunct rectangular areas, in the sense that no domain knowledge is needed to perform the clustering. However, in our case the clusters will be created from the reference points, and we try to place the reference points in a rectangular grid structure.

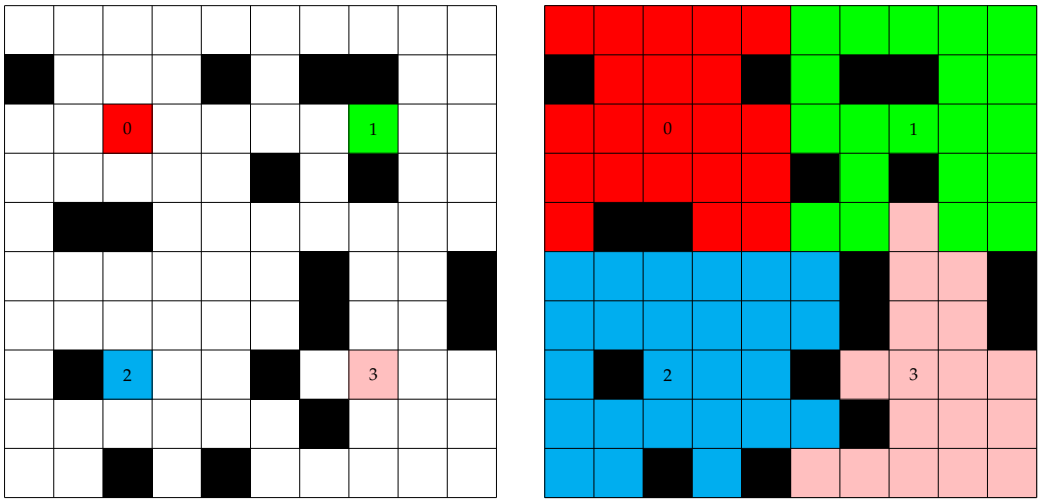
There are several challenges in defining reference points and clusters, and these calculations can be expensive depending on the number of points. Due to obstacles, the

placement of points may not always follow a regular grid structure, which can distort the size of the clusters. When designing clusters, we want to ensure that each point belongs to the reference point closest to it, while also taking rotations into account.

When placing the reference points, we want to ensure that they are as evenly spaced as possible so that the clusters are of approximately the same size and no domain knowledge is needed. To achieve this, the points are placed along a square grid, in every x row and column, starting from the $x \div 2$ row and column. Since there are obstacles on the map, it may not be possible to place a reference point exactly at the predetermined location. In such cases, the closest free position within a smaller surrounding area is selected so that the grid structure and the ratio of clusters are distorted as little as possible.

Then, starting from the selected reference points, we gradually grow the clusters so that the clusters form a Voronoi tessellation [19] of the map. The Voronoi tessellation ensures that any location in a cluster can be reached from any other location in the same cluster. This is not the case in the strict grid based clusters of HPA* [18]. To create the tessellation, we use a modified A* search for each reference point, which already takes into account the rotations in this step since we want to determine which reference point can be reached from a given point in the shortest path. However, it is unimportant from which direction we arrive at the reference point, and we do not know this in advance. Accordingly, we put the reference point with all four orientations into the *Open* set with zero distance so that no direction will be in a privileged role. Since we are not searching to a specific goal, the heuristic value (h) is always zero, so the search always expands the points that are closest to the starting point first. This is actually the same as how Dijkstra's algorithm works, except that here a vertex is only placed in the priority queue when one of its neighbors is expanded and is not already in the queue, rather than initially being placed in it with an infinite distance value.

The clusters are formed as follows: in each step, we iterate over the reference points and run the modified A* search on the current reference point until the distance of the extended points reaches a predefined value. This distance threshold is initially zero and then increases by one in each step. So in the first step, we process all reference points (points at zero distance), in the second step, we assign points at distance 1 from the reference points to the appropriate cluster, and so on. If we extend a point that already belongs to another cluster, we just remove it from the Open set of the currently processed reference point and do not process this point; we go to the next point since we know that this point is already closer to another reference point. If the extended point does not yet belong to any cluster, we assign it to the reference point currently being processed. The A* algorithm guarantees that if we have extended a point, it will definitely contain the shortest distance from the starting reference point. The process continues until all expandable vertices have been processed, i.e., all accessible points on the map have been assigned to a cluster. We store which cluster a given vertex belongs to, as well as which vertices belong to a given cluster, as this information will be useful later. We also calculate the outer boundaries of the individual clusters, i.e., the points that no longer belong to the given cluster. To do this, we not only record the location of the boundary points but also the direction of rotation with which we will enter the cluster from there, as this data will be needed during subsequent RRA* searches. Figure 1 shows an example for clusters created this way.



(**a**) reference points (**b**) clusters

Figure 1. Reference points and the clusters grown from them. The numbers indicate the reference points, and the colors indicate their clusters.

4.2. Distances Between Reference Points

In the next step, we calculate the shortest paths between the reference point pairs. Since these are completely independent calculations, they can be easily parallelized, and we can perform the calculations for each point on a separate thread.

This search is very similar to the definition of clusters; in this case, we continue until all vertices are expanded, but here we actually run one search per reference point through the entire map, rather than several smaller ones started from all the reference points. So at the end of the run we obtain the cost of reaching all the reference points on the map, but only the other reference points will be of interest to us. For each pair of reference points we save the cost of the path and the sequence of clusters $\langle C_{start}, \ldots, C_{goal} \rangle$ through which we get from one to the other. This will be used later by the robots to find the list of clusters to their targets.

4.3. Concave Clusters

Defining the sequence of clusters $\langle C_{start}, \ldots, C_{goal} \rangle$ between two reference points poses an additional challenge, as obstacles may cause a cluster to be touched more than once on a path, which must be managed. A cluster may be concave, so the shortest path between two reference points passes through it more than once. When we navigate the robot to its goal based on the order of the clusters, we do not want a cluster to appear in the list more than once as explained in the next paragraph.

For example, Figure 2 shows the portion of a path going from the left-hand side of the figure towards the right-hand side of the figure. The order of clusters of this portion of the path in Figure 2 from left to right is $\langle C_0, C_2, C_0, C_3 \rangle$, if we keep cluster 0 in the list more than once. In this case, from cluster C_0, we would have to go to cluster C_2, then from cluster C_2 back to cluster C_0, and then to cluster C_3, as this would be the sequence of clusters. As soon as the robot enters cluster C_2, it would target the closest point in cluster C_0, which would mean that the robot would go back to where it came from since the adjacent location from which it came is closer, even after counting the rotations, than the extension of cluster C_0 on the other side. We avoid this unnecessary back turn by immediately targeting the cluster after the second occurrence of cluster C_0 when the robot enters cluster C_2. In this case, the next cluster after the second occurrence of cluster C_2 is cluster C_3, and the sequence

becomes $\langle C_0, C_2, C_3 \rangle$. Actually, in the example of Figure 2, the path remains the same with sequence $\langle C_0, C_2, C_3 \rangle$, but in special cases it might change.

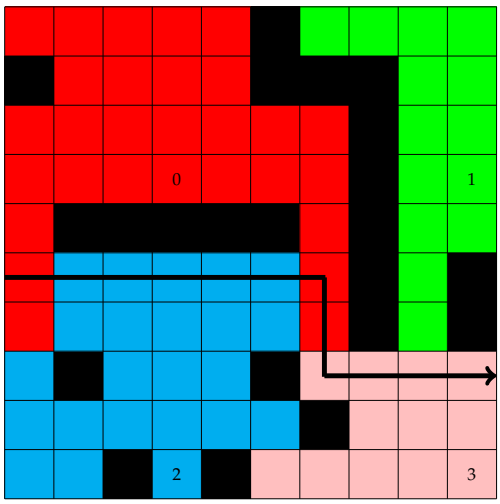


Figure 2. Part of a map with concave cluster. The numbers indicate the reference points, and the colors indicate their clusters. The arrowed line indicates a portion of a path going from the left-hand side of the figure towards the right-hand side of the figure.

So we always keep only the first occurrence of a cluster in the sequence of clusters. The last cluster C_{goal} of the sequence is an exception: it is never removed from the sequence of clusters because it contains the target location of the robot, and it is treated differently from other clusters in the sequence, as described in the next subsection. Regardless of if we keep only the first occurrence of a cluster, it is of course possible that the robot actually enters the same cluster more than once, but this is not a problem, as this is how the path goes.

*4.4. The Algorithm of CRRA**

The CRRA* algorithm is basically a sequence of RRA* searches along the sequence of clusters $\langle C_{start}, \ldots, C_{goal} \rangle$ from the starting cluster of the robot to the goal location.

During the search, the robot is always aware of which cluster it is currently in since every passable point on the map belongs to one of the clusters. Similarly, the robot knows the cluster of its target location. Using the pre-calculated list of clusters of the shortest path between any two reference points, the robot knows the sequence of clusters $\langle C_{start}, \ldots, C_{goal} \rangle$ it must visit on its way to its goal.

In order to reach the next cluster C_i in the sequence, we start an RRA* search from cluster C_i to our current position. During the search, it does not matter which point in cluster C_i we reach; any border point of cluster C_i will be good for us. The closest border point may change if the robot is pushed away from its original shortest path. This is why it was necessary to store the boundaries of the clusters when the clusters were created. We proceed in a similar way as in the case of the traditional RRA* algorithm, with the difference that initially we do not put a single target location in C_i into the *Open* set but the pre-calculated cluster boundaries of C_i together with the entering orientation of the boundary points, and thus start the search. The inside points of cluster C_i are initially put into the *Closed* set since we do not want to expand them during the search because the target position of C_i will definitely not be among them. With this method, we achieve that the algorithm always gives the distance between the robot's current position to the nearest point in cluster C_i. As soon as the robot reaches a cell that already belongs to cluster C_i, the robot looks again to see which cluster is next and starts a new RRA* search from the next cluster C_{i+1}.

When we reach cluster C_{goal} in which the target of the robot is located, we perform a traditional RRA* search to reach the target point, where we do not put the cluster boundaries into the *Open* set but only the target's neighbors with a direction facing the target location. We do this because it does not matter in which direction we reach the target. In this last cluster, the search space will be significantly smaller, and the target is closer, depending on the size of the clusters.

It is important to note that if the shortest path between two reference points passes through the same cluster more than once, we only keep the first occurrence as it is written above. The purpose of this is to avoid pointless backtracking.

5. Considerations for Analytical Evaluation

The evaluation of the CRRA* algorithm has three aspects. The first is the length of the path computed, the second is the computational resource usage of the algorithm, and the third is the overall throughput of an LMAPF-T program using CRRA*. This last aspect is complex because it is a combination of the first two. The throughput depends on both the quality of the solution and the speed of the computation. If the program can compute a somewhat suboptimal solution very fast, then the overall throughput may be better than with an optimal solution which is computed very slowly. In addition, the quality of an LMPAPF-T solution is not just the length of the path of the individual robots but the whole combination of the paths of all robots, which depends on the algorithm using CRRA*.

In this section we make analytical considerations that will conclude that empirical analysis is needed. Therefore we perform experimental analysis later in Section 7.

5.1. Suboptimality of the Path Length with the Clustered Approach

Traditional RRA* is essentially a special case of CRRA*, where a single large cluster is defined. RRA* is a backward A* search that is guaranteed to find the shortest path so that each robot can find the optimal path, even from its new location when it is pushed away from its original path. However, with the introduction of clusters, we lose optimality and the paths may be somewhat longer, but the running time and the memory usage are expected to be significantly reduced.

Unfortunately we cannot prove analytically that the increase in the length of the paths created with CRRA* compared to the paths created with RRA* has an upper limit because for any limit we can create an extreme scenario where the change exceeds the limit. Figure 3 shows an example of how to create such an extreme scenario. The map has a corridor, and two clusters reach into this corridor. The starting point is indicated with "S" and the target is indicated with "T". The length of the shortest path with the non-clustered RRA* would be 1.

"S" is in cluster C_1 and "T" is in cluster C_5. According to CRRA*, if we want to go from cluster C_1 to cluster C_5, then we have to go through clusters $\langle C_1, C_0, C_4, C_5 \rangle$ in this order. This works well for most of the locations in clusters C_1 and C_5; however, in the case of "S" and "T", the shortest path created with CRRA* would be the one indicated with empty circles in Figure 3. The length of this path with turns is 61, so in the cases of "S" and "T", the length of the shortest path with CRRA* is 61 times longer than with RRA*. For any number n, we can create an extreme scenario where the ratio is above n by lengthening the corridor with inserting additional clusters of the type cluster C_2 and cluster C_6 in Figure 3. The same applies to the memory usage and the computation time of the CRRA* algorithm because they also increase with the length of the path.

However, in real scenarios, this extremity is unlikely to happen, so we evaluate the CRRA* algorithm in experimental scenarios. The concise theoretical model for empirical comparison and analysis of paths before and after clustering implementation is that we

measure the length of the path determined by the CRRA* algorithm, and we compare the path length with CRRA* to the path length with RRA*. The measurement is performed with a single robot on each map, so the robot does not have to deal with avoiding others; it only moves along the path determined by the CRRA* algorithm. With a single robot on the map, the RRA* algorithm computes the optimal shortest path. The measurements are executed on the paths of several consecutive tasks.

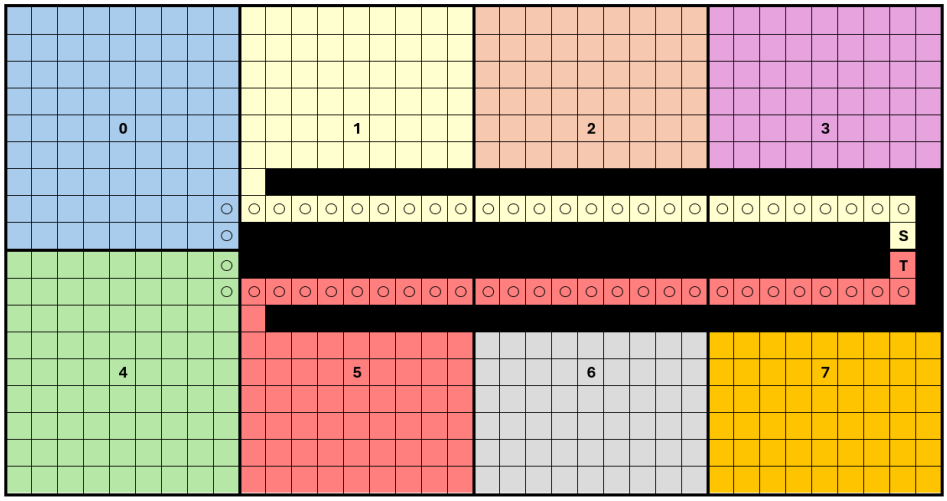


Figure 3. Extreme scenario for analytical evaluation. The numbers indicate the reference points, and the colors indicate their clusters. The shortest path created with CRRA* from starting point "S" to target point "T" is indicated with the empty circles.

5.2. Advantages of the Clustered Approach over Other Approaches

Here we discuss why the clustered approach is better than other approaches.

5.2.1. Advantages over the Waypoint Approach

Alternatively, instead of dividing the map into clusters, we could have worked with a simplified graph consisting solely of some selected reference points called waypoints, like in [5]. In this simplified graph, the weights of the edges would be the distances between the reference points measured on the original map. During route planning, we would connect the start and destination points to the nearest reference points, and compute a route consisting of waypoints, like in the fixed waypoint approach of [5]. Then we would use RRA* searches between the waypoints: we would start an RRA* search from the next waypoint of the route when we reach the currently targeted waypoint. This approach has two disadvantages compared to the clustered approach.

One disadvantage is that finding the closest reference points to the starting and target positions is not necessarily ideal. For example, if our starting point is closer to our target point than the the closest reference point to our starting point, then we would start in the direction of this nearest reference point, and it might easily happen that our path turns back after reaching the first reference point, so the agent would take unnecessary steps. The advantage of the clustered approach over the waypoint approach is that all these additional calculations are not needed; the direction to start is already determined in advance based on the target cluster and the starting position.

The other disadvantage of the waypoint approach is that the waypoint approach would create "traffic jams" around the reference points. Since each robot would try to follow the optimal path between the waypoints, and they would try to approach the waypoints, significant congestion could arise on these roads between waypoints and around the reference points. In contrast, in the cluster solution, the robots do not move along a single narrow path but follow a wider strip, the thickness of which depends on the

cluster size. This distributes the robots more evenly across the map, thus making more efficient use of the available space.

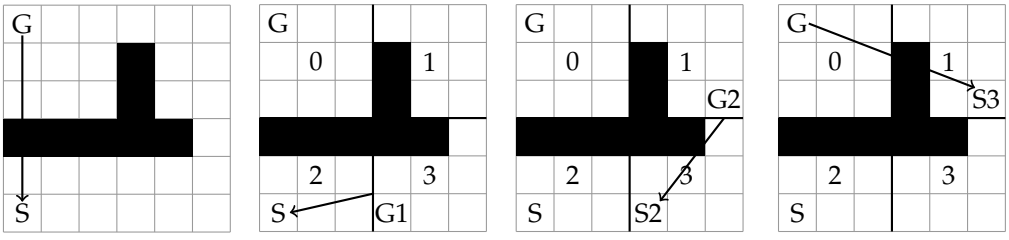
5.2.2. Advantages over the Hierarchical Approach

This cluster approach is conceptually closer to the hierarchical approach of the HPA* algorithm [18] but differs in that the clusters are defined in a different way. The hierarchical approach creates the clusters along a pure grid, while CRRA* creates Voronoi tessellation clusters where only the centers of the clusters are placed along a grid. Both approaches are domain knowledge independent. The hierarchical approach may create clusters which have disconnected parts, while the Voronoi tessellation clusters do not have disconnected parts. While the shortest paths between clusters are pre-computed and stored in HPA*, this would not work in our case due to the presence of multiple robots because if a robot is pushed away from the pre-computed path, then the pre-computed path cannot be used. Therefore, it is better to apply CRRA*, which can efficiently handle situations when a robot deviates from the optimal path inside the cluster—for example, because another robot pushed it away.

5.2.3. Advantages over Non-Clustered Conventional RRA* Algorithm

The RRA* algorithm uses the Manhattan distance on the whole map as a heuristic to estimate the distance to the target. The algorithm always expands the vertex that has the smallest distance from the starting point, increased by the heuristic, first. However, depending on the design of the map, this may not necessarily expand the best vertices. For example, if the target is surrounded by obstacles that need to be avoided, the heuristic does not take this into account, so it initially expands vertices that lead to dead ends. Only then does it start expanding in the correct direction, which can lead to a significant number of unnecessary extensions, especially on large and complex maps. However, by using clusters, we have defined in advance the list of clusters where the agent should go in order to reach its goal, so it will not expand vertices in the wrong direction, or only to a small extent. This allows for more efficient route finding, reducing unnecessary calculations.

This is illustrated in Figure 4. The starting position of the robot is S, and the goal of the robot is G. The non-clustered RRA* in Figure 4a executes an A* search from node G to the node S, which will discover all the nodes of the map until it finds S. If the clustered CRRA* has four clusters on the map (with reference points 0, 1, 2, 3 on the map), then CRRA* knows that the shortest path from S to G is through clusters $\langle C_2, C_3, C_1, C_0 \rangle$. CRRA* first executes a smaller A* search from the border of cluster C_3 to S; see Figure 4b. When the robot reaches G1, CRRA* discards this search space and executes a smaller A* search from the border of cluster C_1 to S2; see Figure 4c. Finally, when the robot reaches G2, CRRA* discards this search space and executes a smaller A* search from G to S3; see Figure 4d.



(**a**) RRA* (**b**) CRRA* search 1 (**c**) CRRA* search 2 (**d**) CRRA* search 3

Figure 4. Non-clustered versus clustered heuristic.

Because the CRRA* algorithm executes shorter distance searches than the RRA* algorithm, the search space is smaller and the CRRA* algorithm uses less memory than the

RRA* algorithm. This is an important improvement if the map is large and there are several robots on the map. The shorter distance search also means that the CRRA* algorithm can give an answer faster than the RRA* algorithm, which is important in real-time applications.

6. Additional Developments Needed for the Experimental Evaluation

As we said, we want to evaluate experimentally the performance of the CRRA* algorithm in LMAPF-T problems by integrating CRRA* into PIBT. In this section, we discuss the additional issues that we had to solve to be able to use PIBT with CRRA* in LMAPF-T.

6.1. Turns

The original PIBT algorithm is without turns, and in our evaluations we must somehow account for rotations. There are two possible solutions to this.

If we determine the next target location that an agent wants to move to, the movement will take at most three steps, first turning in the right direction and then moving forward. However, this means that a location that was free three steps ago may be occupied in the meantime, as another agent may have been moving forward. If we were to calculate the movements of all agents several steps in advance, like in [20], we could more accurately determine whether the target square will actually be free by the time the agent reaches it. However, this would lead to an explosive increase in the number of possible states, significantly increasing the computational effort and thus the planning time.

A second, more computationally efficient solution is to ignore rotations in the initial path planning, and first we simply determine which location to move to next. Then, if an agent's movement requires rotation but another agent wants to move in the meantime, the agent waits in place until the space is actually free. This method preserves the simplicity of PIBT while also allowing for rotations. The disadvantage is that in some cases, agents have to wait to execute their movements, which can reduce the efficiency of the system.

To implement rotations, we choose the second option from the two options above, as it is computationally less demanding, easier to implement, and does not require multiple steps to look ahead. To decide whether a robot can immediately move to the desired location or needs to rotate first, after designing a PIBT without rotation, we iterate over the robots and assign the appropriate actions (rotate, stay, or move forward).

If the original PIBT plan leaves the robot in place, we keep it as is. If the target place is not in the direction the robot is facing, we first assign it a rotation action instead of moving forward. If a robot is already facing the correct direction and the target space is free (or another robot is standing on it, whose forward action we have already defined), it can simply move forward. However, if the target space is occupied and the robot standing there does not plan to move forward, the robot remains in place. If the robot standing there does not yet have a defined next move, its movement must be decided first. If it can move forward, then the original robot can also move; otherwise, it must wait. Due to rotations, it is possible that by the time an agent reaches the right direction, a robot with a higher priority is already occupying the space it is targeting, so it must turn in a new direction.

It may happen that robots try to move in a circle one after the other, which can lead to an infinite cycle, as everyone would be waiting for the one in front of them. To avoid this, we trace the planning chain and if we reach a robot that is already in it, we recognize the cycle. In this case, all robots involved can move forward in the circle at the same time.

6.2. Graph Constraint

The completeness of the PIBT algorithm requires that every point in the graph must be part of a circle; otherwise, it is possible that a high-priority robot's destination is at the

end of a dead end while a lower-priority robot is blocking its path in the dead end. In this case, the lower-priority robot can become stuck at the end of the dead end because the higher-priority robot pushes it but it cannot leave due to its own lower priority, so neither of them can continue their movement. Circles on the map ensure that there is always an accessible path for each robot, avoiding such dead ends. In the case of our evaluation maps, this condition is not given, and the paths may contain dead ends. Figure 5 shows such a situation. The lower-priority robot $R2$ facing east is pushed into location $A1$, and the higher-priority robot $R1$ wants to move onto its goal location $A1$, so $R1$ forces $R2$ to go away, but $R2$ can go away only by forcing $R1$ to step back. Because $R1$ has higher priority than $R2$, this cannot happen, and $R1$ is stuck in the dead-end location. Therefore, a solution is needed that handles such situations.

As mentioned above, on the maps that we examined, it is not guaranteed that every point is part of a circle, so dead ends can occur, in which the robots can become stuck. To prevent this, we identify those points on the map that form dead ends, i.e., cells that have only one neighbor, which is neither an obstacle nor a cell marked as a dead end. The maps we used only have dead ends of one depth, in which case the method described here can be applied.

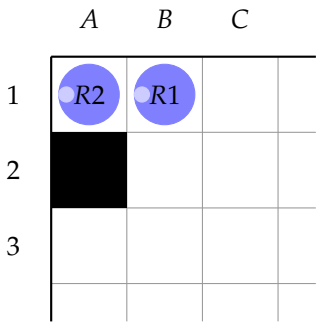
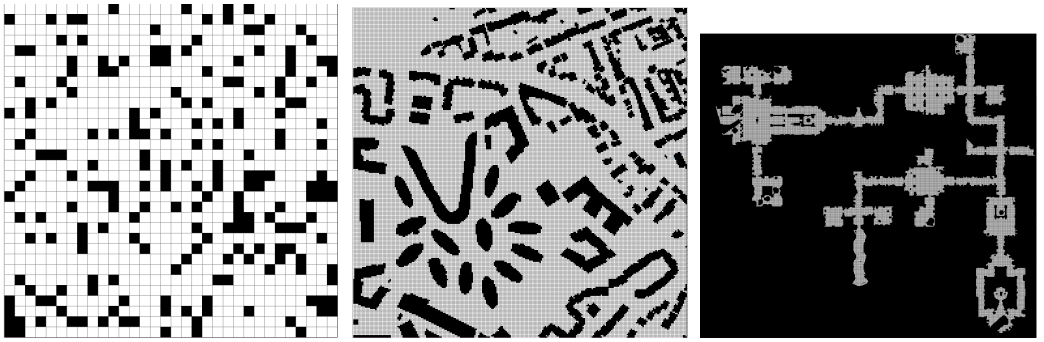
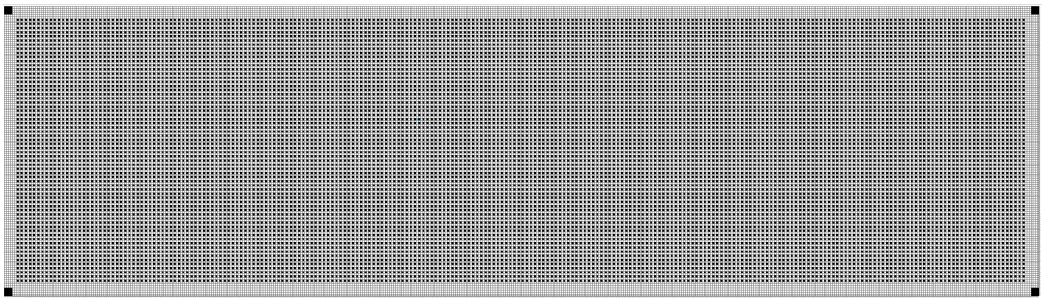


Figure 5. Dead-lock situation in a dead-end cell in the top left corner of the random scenario map of Figure 6. $R1$ and $R2$ are the robots facing east as indicated by the dots in the circles representing the robots.

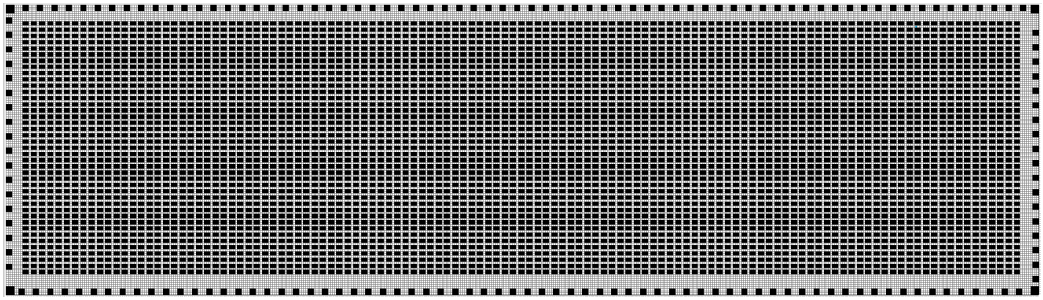


(**a**) random 32 × 32 (**b**) city 256 × 256 (**c**) game 481 × 530



(**d**) sortation 140 × 500

Figure 6. *Cont.*



(**e**) warehouse 140 × 500

Figure 6. Maps used in the evaluation.

A robot can get into a dead end if its goal is there, or if a robot with a higher priority pushes it there. If such a situation arises during the planning, then at each step we gradually increase the priority of the robot stuck in the dead end to a greater extent than for the other robots until it gets out of the dead end. This is similar to the breakout algorithms [21]. This way, after a while, it will definitely have a higher priority than the one that trapped it, and sooner or later it will be able to move out. It will keep this increased priority until it reaches its next goal, to avoid the situation where a robot with a higher priority pushes it back into the dead end again. In the case of deeper dead ends, it is possible that several agents are in it at the same time, which would complicate the situation and another method would have to be devised to handle the situation.

6.3. Setting Priorities

The PIBT article [10] does not specify the method by which initial priorities are assigned to robots. In our evaluation, we examine two different strategies.

In the first case, the priorities (a number between zero and one) are randomly assigned, and then incremented by one at each step until the robot reaches its goal, then the priority is reset to its original value. This ensures that the robot that completed its last goal the longest will have the highest priority. The initial random priority essentially only matters in tie situations. The increment per step is also used in the article, as it ensures that sooner or later, each robot will reach its goal. The one with the highest priority can always move to the place it deems best, so it will definitely reach its goal and its priority will drop so that someone else will have the highest priority and thus reach their goal, and so on.

The second approach gives higher priority to robots that are closer to their target. The distance is determined when the task is assigned, and the robots maintain this value throughout the journey, so if a robot gets closer to its target, we still plan based on the original priority. This method helps robots that are closer to the target reach their target faster, so it is expected that more tasks can be completed in a given time.

During the evaluation tests, we compare the two strategies and see which one proves to be the more efficient solution.

6.4. RRA Algorithm Correction*

The pseudocode for the RRA* algorithm in [8] contains an inaccuracy. Algorithm 1 shows the corrected RRA* algorithm. The dotted parts are the same as in the original algorithm in [8].

Algorithm 1 Reverse Resumable A* with correction.

```
1:  procedure INITIALIZERRA*(O, G)
2:      ...
3:  end procedure
4:
5:  procedure RESUMERRA*(N)
6:      while Open ≠ ∅ do
7:          P ← pop(Open)
8:          Closed ←add P
9:          for all Q ∈ reverse(SUCCESSORS(P)) do
10:             Q.g ← P.g + COST(P, Q)
11:             Q.h ← MANHATTAN(Q, O)
12:             if Q ∉ Open and Q ∉ Closed then
13:                 Open ←add Q
14:             end if
15:             if Q ∈ Open and f(Q) < f(Q in Open) then
16:                 Open ←update Q
17:             end if
18:         end for
19:         if P = N then
20:             return success
21:         end if
22:     end while
23:     return failure
24: end procedure
25:
26: procedure ABSTRACTDIST(N, G)
27:     ...
28: end procedure
```

The function ResumeRRA* should not return with a success value when we put in the *Closed* set the vertex that was the target of the continued search but when its neighbors have already been expanded and placed in the *Open* set or updated. So the "if" part of the code in lines 19–21 must be after the loop in lines 6–18 as shown in Algorithm 1. They are inaccurately in reversed order in the original publication in [8].

Returning too early can cause problems in cases where one of the neighbors of the target vertex is only connected to the graph by a single edge. If we do not process the neighbors before returning from the function, it will never be included in the Open set. So if we want to query this particular vertex, we would have to traverse the entire graph, which would result in significant computational overhead, and we would end up getting the wrong answer that the vertex is not reachable.

Even if the target vertex's neighbor is connected to the graph by more than one edge, it is still possible that we do not obtain the actual shortest path because we have omitted an edge during the calculation, which may be part of the shortest path.

6.5. Integrating CRRA into the PIBT Algorithm*

The CRRA* algorithm is a good choice for determining the distance to the target of an agent. A backward search from the target can be used to determine the distance of a given point from the target, and the previous calculations can be reused for additional points.

Integrated into the PIBT algorithm, this is performed as follows: before we go through the robots according to the priorities and determine their next steps, we first check in a separate cycle whether any robot has just reached its goal. If so, we set the new goal for it, and at the same time start a CRRA* search from the new goal to the robot's current position.

After that, regardless of whether there was a new target or not, each robot calculates the distance of the locations adjacent to its own position from its target. This effectively results in a continuation of the CRRA* search.

Since these calculations are independent for each robot, they can be accelerated by parallelization. If these calculations were not performed at the beginning of the PIBT planning step, parallelization would not be possible since the distances would always have to be calculated by the robot that is next in the priority queue or that has to make room for a robot with a higher priority. By calculating the distances in parallel at the beginning of the step, this data is already available during the planning.

The distances are of course determined by taking the rotations into account, which is relevant from the point of view of the CRRA* search because the goal can be reached from any direction. To achieve this, we initially do not put the goal vertex into the *Open* set but its neighbors in a direction from which we can reach the goal in one step, and then start the search in this way. For the other vertices, we also determine the rotation direction of the neighbors in such a way that it is correct towards the goal from the given location.

7. Empirical Evaluation

In the empirical evaluation, we examine how choosing different cluster sizes affects the runtime, the number of completed tasks, the number of extended vertices, and how much longer the paths with CRRA* are. The baseline for the evaluation is the performance with the plain RRA* algorithm.

In application domains where humans are involved, the exact same experiment cannot be repeated because humans cannot be ”reset” to the same initial condition; therefore, humans are divided into groups, and the different methods are tested on the different groups, and then the results on the different groups are compared. In the case of machine learning techniques, the same set of data is used for training and testing the techniques; therefore, the set of data has to be grouped into training subsets and testing subsets. In such cases, the comparative experiments are performed on the different groups, and the right methods are needed to create the groups, like in [22–24]. Methods for comparative experiments include randomized comparative experiments for controlled studies where subjects are randomly assigned to different groups to minimize bias. Another method is quasi-experimental design, where existing groups are used. In comparative case studies, two or more groups are compared to identify similarities or differences.

In our application domain we are in a good situation because we can use the same set of data and the same experimental setup to evaluate the different techniques because the exact same data and the exact same experimental setup can be recreated to execute the exact same test with the different techniques. We are not using machine learning techniques, so there is no need to group the data into training data groups and testing data groups. We compare the results with the different techniques on the same experimental data and setup.

7.1. Experimental Evaluation Setup

The algorithm was implemented in C++, and the source code is available at the link in Appendix A. The evaluation was executed using the simulation environment of The League of Robot Runners competition (https://www.leagueofrobotrunners.org/ (accessed on 30 November 2025)). Our algorithm tells the robots their next movement, and the simulation environment of The League of Robot Runners competition executes the moves of the robots.

In The League of Robot Runners competition, supported by Amazon Robotics, the robots’ routes must be planned in real time, with only one second available for each step for the robots to choose a target and calculate the next actions. The job of the robots is

to run infinite errands, which they accomplish by visiting different locations on the grid map. The errands must be completed in a specific order. A sequence of such errands is called a task. The objective of the robots is to complete as many tasks as possible, as quickly as possible, in a given time frame. Solutions can be submitted in three categories: pathfinding only, task assignment only, and a combination of these. If the competitor does not start in the combined category, then a basic algorithm is provided for the other part. The evaluation of the algorithm presented in this paper works according to the parameters of the pathfinding category of this competition, telling the robots how to move, and the task selection is provided by the basic scheduling algorithm of the competition.

The simulations were run on a server with the following specifications:

- Processor: AMD EPYC 7763.
- Memory: 128 GB.
- Operating system: Ubuntu 24.04.1 LTS.
- Number of CPUs: 32.

We ran the evaluations on five different maps, with different layouts from The League of Robot Runners competition shown in Figure 6:

- Random: A small, randomly arranged map measuring 32 × 32.
- City: A 256 × 256 map that models a section of the Paris street network.
- Game: A 481 × 530 map consisting of rooms and narrow corridors connecting them.
- Sortation: A 140 × 500 map with a sortation center layout.
- Warehouse: Another 140 × 500 map with a warehouse layout.

The evaluation was performed with different cluster sizes. As a reference, we used the plain RRA* algorithm, which treats the entire map as a single cluster, thus always determining the shortest path between the current and the target locations.

The size of the clusters is determined by the density of the reference points. The tests were run with the reference points placed every 4, 8, 16, 32, and 64 units. These distances roughly correspond to the diameters of the clusters. Table 1 summarizes how many clusters were formed on each map at different reference point distances.

Table 1. Number of clusters depending on the distance between reference points.

Reference Point Distance	Random	City	Game	Sortation	Warehouse
4	64	3458	3295	4371	4349
8	16	965	872	1054	1054
16	4	256	237	279	279
32	1	64	99	64	64
64	1	16	37	16	16

Since the size of the random map is 32 × 32, we obtain one cluster in the case of reference points placed at distances of 32 and 64. This is practically the same as the case without clusters, so the results of the measurements are also identical.

The game map exhibits a higher number of clusters at reference point distances 32 and 64 compared to other maps, and it seems to be a little bit odd. However, the game map is the biggest map among those used in the evaluation, so it is not exceptional that it has a higher number of clusters compared to the others. The odd thing seems to be that the game map does not have a higher number of clusters at distances 4, 8 and 16 compared to other maps. However, this is normal because the game map has large areas covered with obstacles, much larger than the other maps, and at shorter reference point distances, a big proportion of the reference points fall into an area where there is no non-obstacle area

in their grid cell. These reference points do not grow a cluster. At bigger reference point distances, a bigger proportion of the reference points touch at least a little non-obstacle area of the game map, and a bigger proportion of the reference points grow a cluster.

7.2. The Length of the Paths

First, we investigated how the size of the clusters affects the length of the planned paths. To do this, we placed a single robot on each map, so it did not have to deal with avoiding others, it only moved along the path determined by the CRRA* algorithm. During the measurements, we observed how many steps a robot takes to complete 20 tasks.

One of the adjustable parameters of the League of Robot Runners competition program is how many possible tasks the scheduler algorithm foresees, from which it selects the one that the given robot works on next. The number of visible tasks is defined as a given multiple of the number of robots. In this case, this value was 1.5, which resulted in a single robot only seeing one task that it must complete next, and the scheduler had no other choice. This often leads to long paths, as the goal may be located at the other end of the map. This was advantageous in this test, as we specifically wanted to examine the impact of clusters on the path length.

We used the task files from the example scenarios of The League of Robot Runners competition (https://github.com/MAPF-Competition/Start-Kit/tree/72b409f52f4ca76a29b9dce849ed5f624dae0293/example_problems (accessed on 30 November 2025)), and the first robot listed in the agents file was used in the simulations.

We compared the path length with CRRA* to the path length with RRA*. Because there is only one robot on the map, the RRA* algorithm computes the optimal shortest path, while CRRA* computes the suboptimal path due to clustering. Based on the data in Table 2, the CRRA* solution does not significantly increase the length of the paths in most cases. The largest difference was observed on the random map, where the reference points placed every 8 units resulted in a path that was approximately 19% longer. On the other maps, the difference was smaller: on the city map, the path was 8% longer; on the game map, 5% longer; on the sortation map, 2% longer; and on the warehouse map, the path was 6% longer in the worst clustered case compared to the solution without clusters.

Table 2. Number of steps required to complete 20 tasks with the CRRA* algorithm. The bottom row is with the plain RRA* algorithm. The highest values in the columns are indicated with bold.

Reference Point Distance	Random	City	Game	Sortation	Warehouse
4	647	4531	9156	**3726**	**5300**
8	**677**	**4599**	**9304**	3704	5122
16	601	4557	9164	3710	5038
32	-	4595	9124	3706	5044
64	-	4561	9134	3694	5044
Plain RRA*	569	4265	8888	3658	5008

7.3. Memory Usage

During the measurements of the previous subsection, we also examined how many vertices the algorithm extends on average until reaching a target. Note that the CRRA* algorithm uses the RRA* algorithm only to the next cluster. It is clear from the data in Table 3 that the number of extended vertices is the highest in the case of the plain RRA* without clusters, and the smaller the clusters we use, the fewer vertices are processed. We mentioned earlier that in the case of the CRRA* solution, at the beginning of the search, all vertices of the starting cluster are initially placed in the *Closed* set. Since these vertices are

not extended during the search, they are not included in the measured values. The data presented here do not yet include the offsets caused by the other robots. It is expected that fewer extended vertices will result in faster planning time when several robots work on the same map.

Table 3. Average size of the set of closed vertices per target with the CRRA* algorithm. The bottom row is with the plain RRA* algorithm.

Reference Point Distance	Random	City	Game	Sortation	Warehouse
4	403	4200	8876	2992	3055
8	508	4242	8750	2886	3748
16	539	4757	9272	2842	3614
32	-	6868	11,238	3384	4139
64	-	10,881	14,579	5098	6241
Plain RRA*	665	24,957	46,762	24,990	25,054

7.4. Task Completion Throughput

Here we examine how many tasks the agents can complete in 5000 simulation steps of The League of Robot Runners scenarios, and how much time it takes to plan a single move on average. Note that the simulation steps take 1 s, and if the pathfinding finishes earlier, than we do not use the remaining time. The remaining time can be used for optimizations like the Large Neighborhood Search method in [25], but for the evaluation we do not do this because we focus only on the evaluation of CRRA*. We performed the evaluation with different numbers of agents because map congestion affects performance. The number after the map name in the tables indicates how many agents participated in the given test.

Both types of prioritization mentioned above were used in the experiments: the prioritization based on the robots' distance from the goal, and the prioritization based on the time since the last completed goal.

For the following tests, we used the example scenarios (https://github.com/MAPF-Competition/Start-Kit/tree/cdd0356af57a394d4d242a5cc4896e4086b3df31/example_problems (accessed on 30 November 2025)) from The League of Robot Runners competition to define the number of robots, their initial positions, the errands of the tasks to be performed, and other parameters.

Tables 4 and 5 show the results of simulations on less crowded maps. It can be observed that the introduction of clusters on the random map did not result in an acceleration in the planning time. Here, due to the small size of the map, the non-clustered case is also fast enough, and running RRA* between many smaller clusters does not speed up the calculation because the targets are sufficiently close to each other.

The biggest speedup in the average planning time is observed on the city and game maps, where the average planning time is reduced to almost a third when the reference points are placed at a distance of 4 and 8 units. The layout of these maps is less regular than the warehouse-like layout, so the Manhattan distance-based heuristic of the RRA* algorithm is less accurate. This can result in a lot of unnecessary vertex extensions in the case without clusters. On the other hand, on maps with a more regular layout, the Manhattan distance gives a good approximation, so the advantage of the clustered approach is smaller in this regard. This explains the improvement in the computation time on the city and game maps. On the warehouse map, there is no speedup if we place the reference points at a distance of 4 units. This is because the size of the block of shelves, which are obstacles, is close to 4, and the pathfinding of the robots is very constrained at this cluster size. Apart from this, it can be observed that the larger the clusters, the longer the planning takes.

Table 4. Average planning time (ms) in 5000 steps with few robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal. The numbers in brackets show the number of cases when the planning exceeds 1 s.

Reference Point Distance	Random 100	City 500	Game 500	Sortation 800	Warehouse 800
4	6.22	15.80	16.63	17.82	45.20
	6.21	15.35	16.46	18.34	56.06
8	6.54	14.94	16.00	20.93	25.46
	6.43	15.03	15.32	24.23	26.16
16	7.04	19.83	20.02	27.28	28.36
	7.03	19.41	20.02	30.87	30.36
32	-	28.94	25.32	41.16	33.71
	-	28.35	25.55	43.63	36.02
64	-	38.59	29.61	65.35	49.42
	-	38.28	30.81	64.80	52.13
Plain RRA*	6.59	39.29	43.06 (1)	43.01	47.90
	6.40	39.67	42.04	84.52	67.85

Table 5. Number of completed tasks in 5000 steps with few robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal.

Reference Point Distance	Random 100	City 500	Game 500	Sortation 800	Warehouse 800
4	12,423	13,449	6179	22,539	11,885
	12,904	13,648	6227	23,590	13,365
8	12,796	13,595	6212	22,800	20,040
	12,865	13,645	6391	23,234	20,607
16	11,995	13,467	6493	21,837	18,545
	13,146	13,648	6594	22,627	18,908
32	-	13,410	6457	20,473	17,529
	-	13,472	6574	21,672	18,598
64	-	13,679	6345	17,950	18,777
	-	13,761	6416	20,192	19,410
Plain RRA*	13,744	14,697	6487	6726	15,809
	14,564	14,301	6604	11,054	17,155

Regarding the number of tasks completed, the throughput of the clustered version is slightly less than the plain RRA* on the random, city, and the game scenarios. An interesting phenomenon can be observed on the sortation map: in the case without clusters, significantly fewer tasks are completed. The reason for this is that the map simulates a warehouse environment, where the tasks are located on the edge of the map and in the inner areas arranged in a square grid. The robots pick up the packages between the inner shelves and then go to the edge of the map to deliver them. As a result, a lot of agents go straight to the edge of the map, or try to move along the edge of the map towards the delivery point, so they have to turn the least on the way, which results in congestion. In the clustered case, on the other hand, they are better distributed in the inner locations, and congestion occurs less.

On the warehouse map, placing the reference points at a distance of 4 units causes a significant decrease in the number of completed tasks. Here, in narrow passages, the

robots follow each other in longer lines, while in the non-clustered cases and with larger clusters, this is less noticeable. With a small cluster size, the cluster includes only a single corridor, and the algorithm definitely wants to go through on that corridor, while with larger clusters, it is better able to find an alternative route to reach the next cluster.

Tables 6 and 7 show the results of the measurements with a medium number of agents. Tables 8 and 9 show the results of the simulations with a high number of agents. With a high number of agents, the planning exceeded the allowed 1 s time frame several times, mostly in the non-clustered case.

Table 6. Average planning time (ms) in 5000 steps with a medium number of robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal. The numbers in brackets show the number of cases when the planning exceeded 1 s.

Reference Point Distance	Random 200	City 2000	Game 2000	Sortation 2000	Warehouse 2000
4	7.52	46.00	49.42	35.58	80.30
	7.56	46.09	50.93	41.87	109.83
8	7.96	55.44	42.34	46.55	52.19
	8.05	56.97	43.77	66.32	61.58
16	8.56	58.58	55.86	56.81	60.60
	8.71	58.57	57.78	77.22	70.81
32	-	77.31	68.27	79.26	67.54
	-	77.80	68.25	116.25	77.41
64	-	103.85	83.89	108.51	90.69
	-	102.25	85.25	132.18	104.22
Plain RRA*	7.84	95.82 (1)	97.25 (1)	74.58 (1)	79.67
	7.92	99.06	100.52 (1)	114.38	127.15

Table 7. Number of completed tasks in 5000 steps with a medium number of robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal.

Reference Point Distance	Random 200	City 2000	Game 2000	Sortation 2000	Warehouse 2000
4	10,586	55,228	15,947	44,666	27,030
	11,499	56,250	15,370	57,032	33,398
8	11,158	49,820	17,099	73,842	58,445
	12,258	50,851	17,601	75,659	59,682
16	10,615	53,452	23,604	68,265	51,143
	13,796	52,331	22,476	71,238	53,837
32	-	56,496	25,346	59,006	44,586
	-	59,651	25,031	58,413	51,386
64	-	55,266	22,801	42,640	51,312
	-	54,452	24,475	47,626	53,586
Plain RRA*	13,783	59,637	23,743	15,939	41,085
	15,960	58,560	23,135	21,050	48,440

Table 8. Average planning time (ms) in 5000 steps with many robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal. The numbers in brackets show the number of cases when the planning exceeded 1 s.

Reference Point Distance	Random 600	City 5000	Game 5000	Sortation 5000	Warehouse 5000
4	12.78	102.21	97.75	77.08	131.18
	12.79	106.33	99.11	82.36	195.09
8	13.39	100.97	91.22	79.90	91.10
	13.42	105.73	93.68	96.12	107.97
16	13.99	114.24	107.85	96.66	102.97
	14.12	123.95	113.66	129.61	140.41
32	-	132.89	117.67	127.28	114.16
	-	138.29	125.58	161.83	141.95
64	-	149.23 (1)	132.17	155.22 (1)	134.02
	-	158.42 (1)	136.37	194.99	170.04
Plain RRA*	13.51	148.55 (1)	157.90 (33)	116.87 (3)	118.33 (2)
	13.66	151.80 (1)	156.61 (2)	144.21 (2)	204.37 (1)

Table 9. Number of completed tasks in 5000 steps with many robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal.

Reference Point Distance	Random 600	City 5000	Game 5000	Sortation 5000	Warehouse 5000
4	7789	104,812	73,381	68,855	65,907
	7914	99,061	58,116	65,472	70,519
8	8416	106,964	109,757	95,793	82,262
	8109	111,028	88,270	92,473	84,319
16	8206	163,661	87,443	112,732	89,957
	8165	114,063	74,979	135,875	129,734
32	-	120,290	106,903	92,458	90,766
	-	108,717	72,725	105,596	106,135
64	-	156,837	94,010	64,068	82,132
	-	107,228	77,775	77,944	102,272
Plain RRA*	9047	119,074	118,665	38,281	97,616
	8743	117,653	70,243	41,116	117,255

It can be observed that the cluster approach did not bring any speedup in these cases on the random map either. On the other maps, the speedup is less than in the experiments with fewer agents. The number of tasks completed on the random, city, and game maps decreases more significantly with a medium number of agents than we experienced in the previous cases with a lower number of agents. According to the measurements, the speedup visibly decreases with an increase in the number of robots. On more crowded scenarios, the number of conflicts increases, as a result of which more steps are required to reach a goal. Thus, during the 5000 steps, a robot can complete fewer tasks, and fewer RRA* calculation initializations are required per robot in the non-clustered case. For this reason, the advantages of the clustered approach are less evident. Nevertheless, smaller clusters are still able to significantly reduce the planning time.

During the planning process, we do not take into account the paths of other robots, so each agent tries to follow the shortest path for itself without taking into account collisions.

Other robots are taken into account and collision avoidance is managed by the PIBT algorithm. As a result, congestion develops in certain parts of the paths, while in other, less frequented areas, the robot density remains low. These congestions significantly reduce the number of completed tasks, as the robots are often forced to wait.

Regarding the prioritization methods, the priority order based on the time since the last completed goal proved to have higher throughput in several cases based on the measurements. However, the results show that neither approach is clearly better, as their throughput may depend on the density of the robots and the design of the map.

7.5. Task Completion Efficiency

The previous subsection shows the average planning times and the number of completed tasks. The faster planning sometimes produces somewhat fewer tasks, but it is hard to tell if this compromise is worthwhile or not. Here we want to combine the two values into a single value that measures the efficiency. We define an efficiency measure as the number of completed tasks with 1 s planning time effort, i.e., $1000 * Tasks \div (AvgTime * 5000)$ where the *Tasks* is the completed tasks in 5000 steps and *AvgTime* is the average planning time (ms) during 5000 steps from the above tables.

The computed efficiency measures are shown in Tables 10–12 with few robots, a medium number of robots, and many robots, correspondingly. Each table is followed by the trend curves of the data in the corresponding table on Figures 7–9.

On the random map scenarios, there is no efficiency increase. Cluster size 4 is not suitable in the warehouse scenarios because of the aforementioned reason.

Otherwise, in the scenarios with few robots, the CRRA* with a cluster size of 16 or less is much more efficient than the plain RRA*. The efficiency is often two or three times better, and it is even more than three times better on the sortation map at cluster size 8.

In the scenarios with a medium number of robots, the CRRA* is also more efficient than the plain RRA*. The efficiency is often about two times better, and it is about more than seven times better on the sortation map at cluster size 8.

Table 10. Number of completed tasks with 1 s planning time in the case of few robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal.

Reference Point Distance	Random 100	City 500	Game 500	Sortation 800	Warehouse 800
4	399.45	170.24	73.31	178.66	52.59
	415.59	177.82	75.66	158.99	47.68
8	391.31	181.99	77.65	239.78	157.42
	400.16	181.57	83.43	192.41	157.55
16	340.77	135.82	64.86	233.40	130.78
	374.00	140.63	65.87	209.67	124.56
32	-	92.67	51.00	145.28	105.92
	-	95.04	51.46	130.50	103.26
64	-	70.89	42.86	82.55	75.99
	-	71.90	41.65	79.95	74.47
Plain RRA*	417.12	74.81	30.13	65.51	66.01
	455.13	72.10	31.42	57.02	50.57

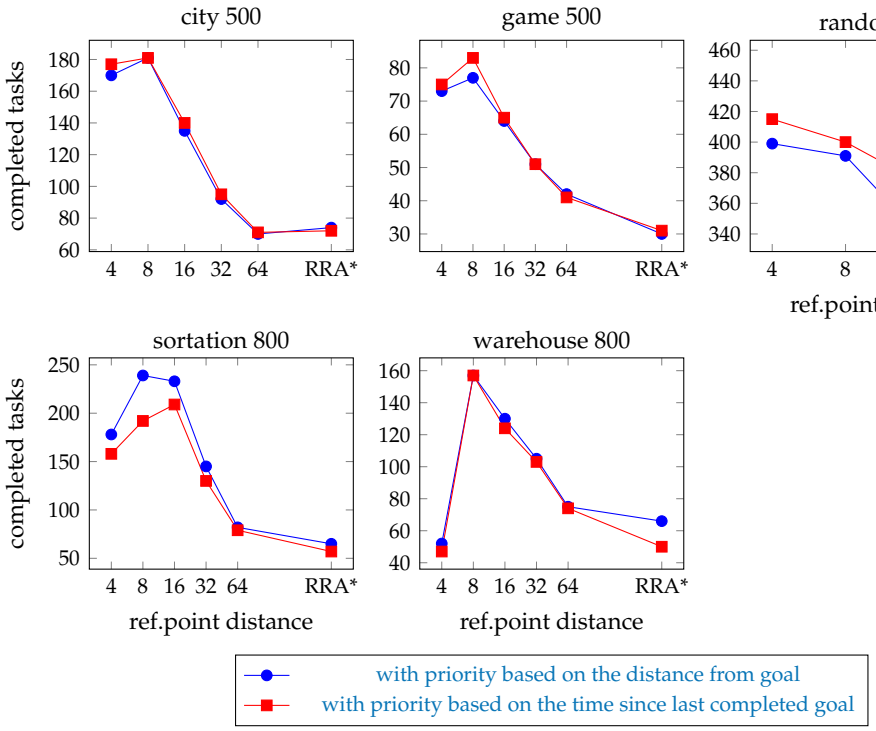


Figure 7. Number of completed tasks with 1 s planning time in the case of few robots. Trend curves from the data of Table 10.

In the scenarios with many robots, the CRRA* is still more efficient than the plain RRA*, but the efficiency gain can be observed mainly at cluster size 8 and 16 . This is probably because if the cluster size is bigger, then there is more space in a cluster for the increased number of robots to find alternative routes. The efficiency at the best cluster size is close to two times better on the city map, about fifty percent better on the game map, and about three times better on the sortation map. There is only a slight efficiency gain on the warehouse scenario, and there is even an efficiency loss at cluster size 64.

Table 11. Number of completed tasks with 1 s planning time in the case of medium number of robots. The gray rows are with priority based on the distance from goal, and the white rows are with priority based on the time since the last completed goal.

Reference Point Distance	Random 200	City 2000	Game 2000	Sortation 2000	Warehouse 2000
4	281.54	240.12	64.54	251.07	67.32
	304.21	244.09	60.36	272.42	60.82
8	280.35	179.73	80.77	317.26	223.97
	304.55	178.52	80.42	228.16	193.84
16	248.01	182.49	84.51	240.33	168.79
	316.79	178.70	77.80	184.51	152.06
32	-	146.15	74.25	148.89	132.03
	-	153.34	73.35	100.50	132.76
64	-	106.43	54.36	78.59	113.16
	-	106.51	57.42	72.06	102.83
Plain RRA*	351.61	124.48	48.83	42.74	103.14
	403.03	118.23	46.03	36.81	76.19

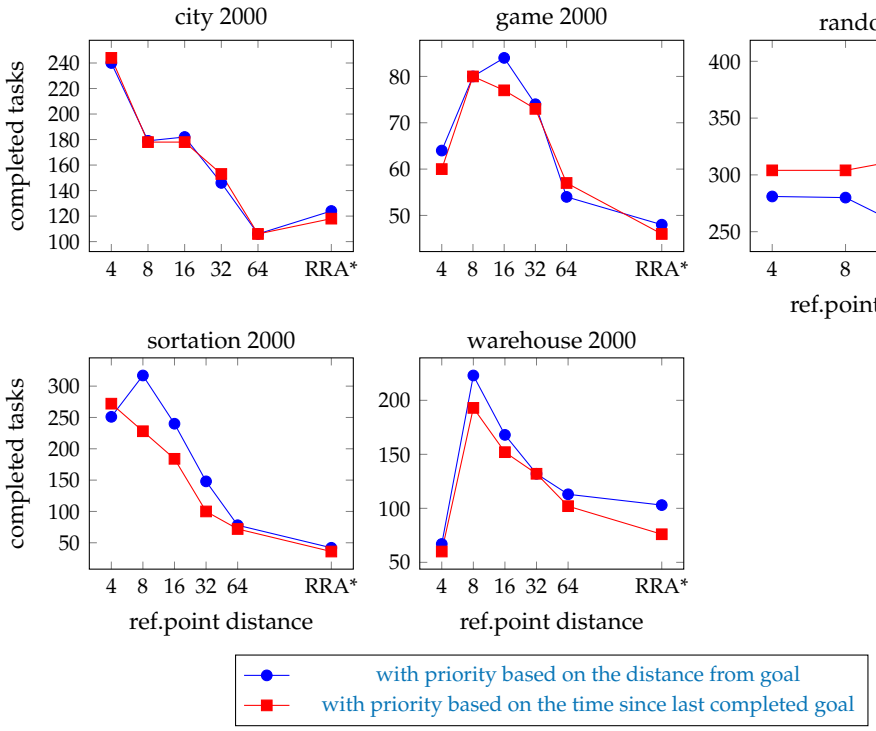


Figure 8. Number of completed tasks with 1 s planning time in the case of medium number of robots. Trend curves from the data of Table 11.

Table 12. Number of completed tasks with 1 s planning time in the case of many robots. The gray rows are with priority based on the distance from the goal, and the white rows are with priority based on the time since the last completed goal.

Reference Point Distance	Random 600	City 5000	Game 5000	Sortation 5000	Warehouse 5000
4	121.89	205.09	150.14	178.66	100.48
	122.70	186.33	117.28	158.99	72.29
8	125.71	211.87	240.64	239.78	180.60
	120.85	210.02	188.45	192.41	156.19
16	117.31	286.52	162.16	233.25	174.72
	115.65	184.05	131.94	209.67	184.79
32	-	181.04	181.70	145.28	159.02
	-	157.23	115.82	130.50	149.54
64	-	210.20	142.26	82.55	122.57
	-	135.37	113.81	79.95	120.29
Plain RRA*	133.93	160.32	150.30	65.51	164.99
	128.01	155.01	89.70	57.02	114.75

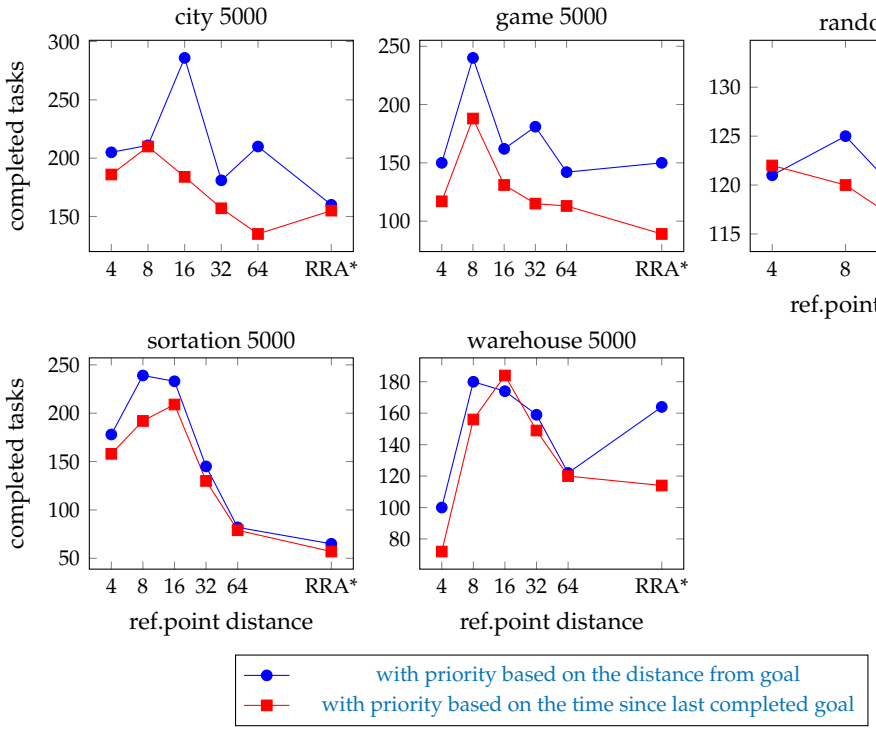


Figure 9. Number of completed tasks with 1 s planning time in the case of many robots. Trend curves from the data of Table 12.

8. Conclusions

The LMAPF-T problem is crucial in many industrial applications, especially in automated warehouses, where reducing human resources can lead to significant efficiency gains. We contributed to this research field with the novel CRRA* algorithm, and we presented the operation of the CRRA* algorithm together with its advantages and disadvantages. CRRA* is an enhancement of the RRA* algorithm. For the experimental evaluation of the CRRA* algorithm, we used it in the PIBT algorithm to sort the neighboring locations based on the distance from the target. In doing so, we had to solve several challenges, including the introduction of rotations into the PIBT algorithm and the relaxation of the graph constraint. We have corrected the originally published plain RRA* algorithm.

We pointed out that it is not possible to prove analytically that the increase in the length of the paths created with CRRA* compared to the paths created with RRA* always has an upper limit because for any limit we can create an extreme scenario where the increase exceeds the limit. Future research could improve it to better handle these corner cases and take into account the topology of the maps during the clustering process, although with this we might have to give up the domain knowledge independency of the CRRA* algorithm. However, in real scenarios this extremity is unlikely to happen, and CRRA* works well for most of the cases, so we evaluated the CRRA* algorithm with different cluster sizes in experiments using the scenarios of the League of Robot Runners competition. The experimental results showed that the CRRA* solution does not significantly increase the length of the paths in most cases on larger maps. Regarding the memory usage, the number of vertices examined by the CRRA* algorithm can be reduced to less than 20% of the vertices examined by the RRA* algorithm. The biggest speedup in the average planning time in the LMAPF-T problem with our PIBT implementation is observed on the city and game maps, where the average planning time is reduced to almost a third when the clusters are placed at 4 and 8 units. The average planning time on the sortation and on the warehouse maps can be halved. The evaluation results show that the CRRA* leads to a

significant reduction in computation time, especially on larger maps where the obstacles are not in a regular grid structure but rather form well-separated spaces. At the same time, the degradation in solution quality is minimal. The faster planning sometimes produces somewhat fewer tasks; therefore, we combined the two values into an efficiency measure as the number of completed tasks with 1 s planning time effort. This efficiency measure shows that the CRRA* algorithm is clearly more efficient on larger maps than the RRA* algorithm.

We do not have access to a physical environment where we could have tested the CRRA* algorithm. The robots that we are investigating in this paper have the following movement capabilities: move forward, turn left, or turn right. Many warehouse robots are like this. Because of these movement capabilities, these warehouse robots move on a grid, and the grid-like map is a good model of the movements of the physical robots. The fundamental challenge of transitioning the pathfinding algorithms from simulation to physical warehouse robots is related to the time requirement of the movements. Such robots can move faster on a straight path than on a path that includes turns because a turn movement requires a deceleration before and an acceleration after the turn movement. According to the discussions at the Virtual Expo after the last League of Robot Runners competitions, the next version of the competition system is likely to include this feature of physical warehouse robots. Our future research includes the testing of the CRRA* algorithm with this kind of League of Robot Runners testing environment as well, when the new simulation system of the League of Robot Runners competition will be available.

The faster computation of slightly suboptimal paths with the CRRA* algorithm can be useful in many practical applications, especially in situations where real-time planning is more important than finding the optimal paths. Although the algorithm was used in combination with PIBT, it can be useful not only in this context. It can also be used as a heuristic in any multi-agent pathfinding solution, for example, in Windowed Hierarchical Cooperative A* (WHCA*). Manhattan distance often inaccurately estimates the remaining distance to the target, and RRA* can be expensive in large spaces. Using CRRA*, we can obtain a faster computed, nearly accurate heuristic.

Author Contributions: The paper is based primarily on the G.C.'s dissertation. L.Z.V. supervised and directed the work, contributed to the evaluation of the results, and edited the final form of the paper. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The availability of the implementation is described in Appendix A. The example scenarios used for the evaluations are linked in the text of the paper at the relevant places.

Acknowledgments: The authors are thankful to their affiliated organization for supporting this research.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

CA*	Cooperative A*
CRRA*	Clustered Reverse Resumable A*
HCA*	Hierarchical Cooperative A*
HPA*	Hierarchical Pathfinding A*
LMAPF	Lifelong Multi-agent Pathfinding
LMAPF-T	Lifelong Multi-agent Pathfinding with Turns
MAPF	Multi-agent Pathfinding
PIBT	Priority Inheritance with Backtracking

RRA*	Reverse Resumable A*
WHCA*	Windowed Hierarchical Cooperative A*

Appendix A. Source Code of the Implementation

The source code of CRRA* integrated with PIBT is available in the github repository at https://github.com/Csanyi/LORR_2024 (accessed on 30 November 2025). The implementation with distance-to-target priority is available on the *main* branch, while the prioritization based on the last completed target is available on the *random_priority* branch. A Large Neighborhood Search (LNS) addition is available on the *lns* branch. The source code at the link also contains the framework of the League of Robot Runners starter kit. The own solution is available in the source files *src/PIBT/**, *src/RRA_star/**, *src/map_utils/**, *src/MAPFPlanner.cpp*, as well as in the header files in the *inc* folder.

The framework was provided by The League of Robot Runners competition (https://github.com/MAPF-Competition/Start-Kit (accessed on 30 November 2025)).

References

1. Stern, R.; Sturtevant, N.R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T.T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T.K.S.; et al. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. In Proceedings of the Twelfth International Symposium on Combinatorial Search, SOCS 2019, Napa, CA, USA, 16–17 July 2019; AAAI Press: Washington, DC, USA, 2019; pp. 151–159.
2. Wurman, P.R.; D'Andrea, R.; Mountz, M. Coordinating Hundreds of Cooperative, Autonomous Vehicles in Warehouses. *AI Mag.* **2008**, *29*, 9.
3. Morris, R.; Pasareanu, C.; Luckow, K.; Malik, W.; Ma, H.; Kumar, T.K.S.; Koenig, S. Planning, Scheduling and Monitoring for Airport Surface Operations. In Proceedings of the AAAI-16 Workshop on Planning for Hybrid Systems (PlanHS), Phoenix, AZ, USA, 13 February 2016.
4. Botea, A.; Bouzy, B.; Buro, M.; Bauckhage, C.; Nau, D. *Artificial and Computational Intelligence in Games (DFU-Vol. 6)*; Schloss Dagstuhl–Leibniz-Zentrum für Informatik: Wadern, Germany, 2013; Chapter Pathfinding in Games, p. 12. [CrossRef]
5. Varga, L.Z. Empirical Analysis of Hierarchical Pathfinding in Lifelong Multi-Agent Pathfinding with Turns. *Systems* **2025**, *13*, 331. [CrossRef]
6. Morag, J.; Stern, R.; Felner, A. Adapting to Planning Failures in Lifelong Multi-Agent Path Finding. In Proceedings of the International Symposium on Combinatorial Search, Prague, Czech Republic, 14–16 July 2023; Volume 16, pp. 47–55. [CrossRef]
7. Jakab, O.; Varga, L.Z. Options to Speed-Up Search in Lifelong Multi-Agent Pathfinding. In *Advances in Practical Applications of Agents, Multi-Agent Systems, and Digital Twins: The PAAMS Collection*; Springer Nature: Cham, Switzerland, 2024; pp. 121–133. [CrossRef]
8. Silver, D. Cooperative Pathfinding. In Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, Marina Del Rey, CA, USA, 1–2 June 2005; Volume 1, pp. 117–122. [CrossRef]
9. Pertzovsky, A.; Stern, R.; Felner, A.; Zivan, R. Enhancing Lifelong Multi-Agent Path-finding by Using Artificial Potential Fields. In Proceedings of the 24th International Conference on Autonomous Agents and Multiagent Systems, Richland, SC, USA, 19–23 May 2025; AAMAS '25, pp. 2711–2713.
10. Okumura, K.; Machida, M.; Défago, X.; Tamura, Y. Priority inheritance with backtracking for iterative multi-agent path finding. *Artif. Intell.* **2022**, *310*, 103752. [CrossRef]
11. Wu, M.; Yan, W.; Hasan, H.; Jamaluddin, R.A. A Review of Multi-Agent Path Finding Algorithms. In Proceedings of the 2023 11th International Conference on Information Systems and Computing Technology (ISCTech), Qingdao, China, 30 July–1 August 2023; IEEE: New York, NY, USA, 2023; pp. 69–73. [CrossRef]
12. Yu, J.; LaValle, S. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In Proceedings of the AAAI Conference on Artificial Intelligence, Bellevue, WA, USA, 14–18 July 2013; Volume 27.
13. Felner, A.; Stern, R.; Shimony, S.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N.; Wagner, G.; Surynek, P. Search-Based Optimal Solvers for the Multi-Agent Pathfinding Problem: Summary and Challenges. In Proceedings of the International Symposium on Combinatorial Search, Pittsburgh, PA, USA, 16–17 July 2017; Volume 8, pp. 29–37. [CrossRef]
14. Pavel, S.; Ariel, F.; Roni, S.; Eli, B. Efficient SAT Approach to Multi-Agent Path Finding Under the Sum of Costs Objective. In *ECAI 2016*; IOS Press: Amsterdam, The Netherlands, 2016. [CrossRef]
15. Yu, J.; LaValle, S.M. Optimal Multirobot Path Planning on Graphs: Complete Algorithms and Effective Heuristics. *IEEE Trans. Robot.* **2016**, *32*, 1163–1177. [CrossRef]

16. Gómez, R.N.; Hernández, C.; Baier, J.A. Solving Sum-of-Costs Multi-Agent Pathfinding with Answer-Set Programming. In Proceedings of the AAAI Conference on Artificial Intelligence, New York, NY, USA, 7–12 February 2020; Volume 34, pp. 9867–9874. [CrossRef]
17. Cohen, L.; Uras, T.; Kumar, T.K.S.; Xu, H.; Ayanian, N.; Koenig, S. Improved solvers for bounded-suboptimal multi-agent path finding. In Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, New York, NY, USA, 9–15 July 2016; AAAI Press: Washington, DC, USA, 2016; IJCAI'16, pp. 3067–3074.
18. Botea, A.; Müller, M.; Schaeffer, J. Near Optimal Hierarchical Path-Finding. *J. Game Dev.* **2004**, *1*, 1–30.
19. Shahabi, C.; Sharifzadeh, M. *Encyclopedia of Database Systems*; Springer: Boston, MA, USA, 2009; Chapter Voronoi Diagrams; pp. 3438–3440. [CrossRef]
20. Okumura, K.; Tamura, Y.; Défago, X. winPIBT: Extended Prioritized Algorithm for Iterative Multi-agent Path Finding. *arXiv* **2019**. [CrossRef]
21. Hirayama, K.; Yokoo, M. The distributed breakout algorithms. *Artif. Intell.* **2005**, *161*, 89–115. [CrossRef]
22. Chen, J.; Fan, F.; Wei, C.; Polat, K.; Alenezi, F. Decoding driving states based on normalized mutual information features and hyperparameter self-optimized Gaussian kernel-based radial basis function extreme learning machine. *Chaos Solitons Fractals* **2025**, *199*, 116751. [CrossRef]
23. Chen, J.; Cui, Y.; Wei, C.; Polat, K.; Alenezi, F. Driver fatigue detection using EEG-based graph attention convolutional neural networks: An end-to-end learning approach with mutual information-driven connectivity. *Appl. Soft Comput.* **2026**, *186*, 114097. [CrossRef]
24. Chen, J.; Cui, Y.; Wei, C.; Polat, K.; Alenezi, F. Advances in EEG-based emotion recognition: Challenges, methodologies, and future directions. *Appl. Soft Comput.* **2025**, *180*, 113478. [CrossRef]
25. Li, J.; Chen, Z.; Harabor, D.; Stuckey, P.J.; Koenig, S. MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. In Proceedings of the AAAI Conference on Artificial Intelligence, Virtual, 22 February–1 March 2022; Volume 36, pp. 10256–10265. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.