Symbolic Planning and Multi-Agent Path Finding in Extremely Dense Environments with Unassigned Agents

Bo Fu[*], Zhe Chen[*], Rahul Chandan, Alexandre Ormiga Galvao Barbosa, Michael Caldara, Joey W. Durham, and Federico Pecora

Amazon Robotics
300 Riverpark Dr, North Reading, MA 01864
{bofu, zhecm, rcd, aormiga, caldaram, josepdur, fpecora}@amazon.com

Abstract

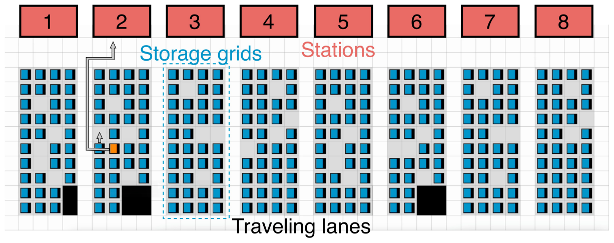
We introduce the Block Rearrangement Problem (BRaP), a challenging component of large warehouse management which involves rearranging storage blocks within dense grids to achieve a goal state. We formally define the BRaP as a graph search problem. Building on intuitions from sliding puzzle problems, we propose five search-based solution algorithms, leveraging joint configuration space search, classical planning, multi-agent pathfinding, and expert heuristics. We evaluate the five approaches empirically for plan quality and scalability. Despite the exponential relation between search space size and block number, our methods demonstrate efficiency in creating rearrangement plans for deeply buried blocks in up to 80×80 grids.

1 Introduction

Amazon Robotics fulfillment centers are large warehouses that utilize robotic Automated Storage and Retrieval Systems to store inventory and fulfill customer orders. As illustrated in Figure 1, fulfillment centers store inventory items in shelves (blocks) arranged into storage grids separated by travel lanes. Robot drives can lift and move these blocks. When a station at the periphery requests an assigned block for item picking or stowing, robot drives move that block, transport it to the storage grid's boundary and then utilize travel lanes to bring it to the station. The system leverages dense storage grids to achieve efficient space utilization and lower fulfillment costs. This creates a challenge: assigned blocks may be deeply buried and obstructed by other blocks, requiring complex rearrangement plans.

We define the Block Rearrangement Problem (BRaP) as follows. Given a grid layout containing assigned blocks, unassigned blocks, empty cells, and obstacles, as illustrated in Figure 2a, the objective is to find the lowest-cost plan to move each assigned block to a set of desired goal locations. Note that the blocks are also referred to as agents in other problem settings (Stern et al. 2019). These goal locations can be either contiguous or disjoint. When goal locations are at the grid boundary, assigned blocks are rearranged for direct travel lane access. Alternatively, when goal locations

(a) Photo



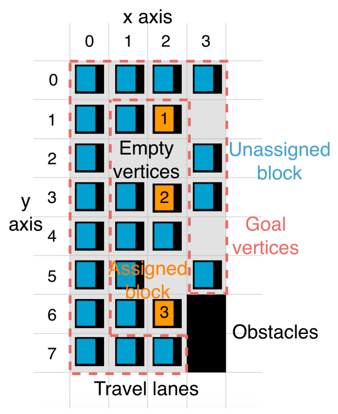
(b) Illustration diagram

Figure 1: Fulfillment center with blocks in storage grids separated by travel lanes. Assigned blocks marked in orange.

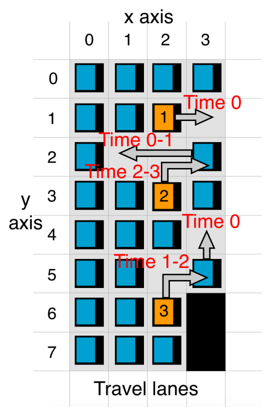
are within the grid, assigned blocks are positioned for potential future access. The plan consists of block moving actions with their associated timestamps. Cost metrics can include sum of action costs, makespan, or other user-defined objectives. An example plan for the problem shown in Figure 2a is illustrated in Figure 2b.

The Block Rearrangement Problem in dense warehouses shares fundamental structural similarities with various puzzles and real-world problems, including the sliding tile puzzle (Gozon and Yu 2024), Rush Hour puzzle (Cian et al. 2022), block relocation problem (Lu, Zeng, and Liu 2020), box world/Sokoban (Zawalski et al. 2024), parking lot rearrangement (Guo and Yu 2023), Multi-Agent Path Finding (MAPF) (Stern et al. 2019; Li et al. 2022; Shen et al. 2023; Okumura 2023b), and MAPF with Unassigned Agents (Felner and Stern 2026). These problems involve rearrangement in discrete space, aiming to achieve goal configurations un-

[*]These authors contributed equally.



(a) Definition



(b) Example plan

Figure 2: Block Rearrangement Problem definition.

der movement constraints. Due to the combinatorial increase in the number of possible configurations, these problems are generally NP-hard (Ratner and Warmuth 1986; Flake and Baum 2002; Yu and LaValle 2013a).

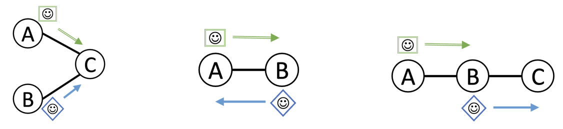
This paper makes several contributions: it formally defines the Block Rearrangement Problem, establishes similarities and differences between BRaP and existing puzzles and research problems, develops five distinct solution algorithms, and evaluates their optimality and computational efficiency. Through this comprehensive analysis, we establish baseline approaches for solving BRaPs and identify directions for future research.

2 Problem Description

A BRaP is defined on a graph $G = (V, E)$ with the set of vertices V and edges $E = \{(u, v) \mid u, v \in V\}$ connecting adjacent vertices. Let $\mathcal{I} = \{1, \cdots, n_{\mathcal{I}}\}$ denote the set of assigned blocks (marked in orange in Figure 2) and each assigned block $i \in \mathcal{I}$ starts at an initial vertex $s_i \in V$ and must be moved to a set of desired goal vertices $V_i \subseteq V$. Additionally, $\mathcal{J} = \{n_{\mathcal{I}}+1, \cdots, n_{\mathcal{I}}+n_{\mathcal{J}}\}$ denotes unassigned blocks without desired goals. Actions $A = \{move, wait, complete\}$ comprise a set of operations that modifies the time or location of the blocks. A path p_i for block $i \in \mathcal{I} \cup \mathcal{J}$ consists of an action sequence $(a_i^1, a_i^2, \cdots, a_i^{|p_i|})$, $a_i^k \in A$. Assigned block paths must move assigned blocks to goal vertices to be completed, while unassigned block paths facilitate assigned block movements. A solution $P = \{p_i \mid i \in \mathcal{I} \cup \mathcal{J}\}$ to a BRaP is a set of conflict-free paths, that transition all the assigned blocks to their goal vertices. The objective is to find a feasible solution minimizing either sum of action costs, makespan, or other user-defined cost metrics, as detailed in Sec. 2.3.

2.1 Actions

The action set A contains three types of actions that can be applied to the blocks, each takes one time step to execute and is associated with a corresponding cost. The action



(a) Vertex (b) Edge (c) Following

Figure 3: Three types of conflicts (Stern et al. 2019).

move(i, t, u, v) moves block $i \in \mathcal{I} \cup \mathcal{J}$ between two adjacent vertices $(u, v) \in E$ at time step t if u is a block and v is an empty vertex. *wait*(i, t) forces a block to wait in place at time t. The *complete*(i, t, v) action can only be applied when an assigned block $i \in \mathcal{I}$ is at one of its goal vertices $v \in V_i$. The complete action removes the block from the assigned block set: $\mathcal{I} \leftarrow \mathcal{I} \backslash i$.

It is important to note that, according to the above definition, the destination of a *move* action must be an empty vertex. We prohibit platooning, where one block closely follows another, to better align with warehouse operational constraints. Specifically, the movement must avoid three conflicts (Stern et al. 2019):

- *Vertex Conflict* (Figure 3a), where two blocks use the same vertex at the same timestep,
- *Edge Conflict* (Figure 3b), where two blocks use the same edge at the same timestep,
- *Following Conflict* (Figure 3c), where one block occupies a vertex that was occupied by another block in the previous time step.

2.2 Terminal State

The objective is to empty the assigned block set through completion actions. Three distinct completion actions are possible for assigned blocks. In the first case, an assigned block reaching the goal (e.g., the boundary) is considered rearranged and removed, converting its location to an empty vertex. In the second case, an assigned block becomes unassigned upon reaching its goal location. In the third case, an assigned block becomes an obstacle at its goal location. Each completion action corresponds to a specific practical scenario. This paper focuses on developing algorithms and conducting experiments using the third option.

2.3 Objectives and Metrics

Multiple metrics can be used to evaluate the quality of the BRaP solutions, including the number of moves of assigned blocks, the number of moves of unassigned blocks, sum of action costs, and the makespan of the whole plan. In Block Rearrangement, we care about all of these metrics.

Let $c(a)$ denote the cost of an action $a \in A$. $c(a)$ returns the cost according to the types of the blocks and the actions. An example action cost matrix is illustrated in Table 1.

The cost of a path $p_i = (a_i^1, \cdots, a_i^{|p_i|})$ is defined as:

$$c(p_i) = \sum_{k=1}^{|p_i|} c(a_i^k) \tag{1}$$

Block type	*move*	*wait*	*complete*
Assigned	2	1	2
Unassigned	2	0	N/A

Table 1: Example action costs

We define the objectives: composite cost of a solution $P = \{p_i \mid i \in \mathcal{I} \cup \mathcal{J}\}$ and the makespan of the solution, according to (2)-(3), respectively.

$$\text{Composite cost: } c(P) = \sum_{i \in \mathcal{I} \cup \mathcal{J}} c(p_i) \tag{2}$$

$$\text{Makespan: } c_m(P) = \max_{i \in \mathcal{I} \cup \mathcal{J}} c(p_i) \tag{3}$$

2.4 Variants of the Problem

The Problem can be extended in several ways to accommodate different operational requirements:

- *Constraint on simultaneous actions:* Limiting the number of blocks that can move concurrently to reflect real-world constraints.
- *Shared or distinct goal sets for blocks:* Blocks might have individual specific goals or share a common set of acceptable end positions.
- *Priority assignments:* Additional priority for certain blocks, ensuring critical blocks are rearranged first.

3 Related Work

As mentioned in Sec. 1, BRaP is closely related to various puzzles and research problems. Solutions to these problems have employed diverse approaches including symbolic planning (Davesa Sureda et al. 2024), search and conflict resolution (Li et al. 2022; Sharon et al. 2015; Ma et al. 2019), mixed-integer programming (Guo and Yu 2023; Lu, Zeng, and Liu 2020), boolean satisfiability (Cian et al. 2022), reinforcement learning (Shoham and Elidan 2021; Damani et al. 2021; Wang et al. 2025), and supervised learning formulations (Li et al. 2020; Jiang et al. 2025) to find low-cost plans. Given their demonstrated optimality and efficiency, we leverage symbolic planning and MAPF algorithms to develop the planning methods in this paper.

3.1 Symbolic Planning and PDDL

Symbolic planning is a model-based approach in artificial intelligence that solves decision-making problems by explicitly representing states, actions, and their transitions using symbolic logic. Planning languages like Planning Domain Definition Language (PDDL) enable the formulation of complex planning problems as structured domains, encompassing crucial elements such as initial conditions, feasible actions, and desired goal states, allowing planners to search for action sequences that achieve specified goals. PDDL has been successfully applied to various domains: modeling Rush Hour and sliding puzzles (Davesa Sureda et al. 2024), formulating Rubik's cube solutions (Muppasani, Pallagani, and Srivastava 2024), solving logistics problems for transport routing (Helmert 2009), and finding robot paths in grid worlds (Estivill-Castro and Ferrer-Mestres 2013). Symbolic representations have also proven effective in task and motion planning for robotic manipulation (Garrett et al. 2021; Silver et al. 2021). Recent work has combined the symbolic representation with deep learning and large language models to create flexible long-horizon plans (Srivastava et al. 2022; Agia et al. 2023; Lin et al. 2023). Building on this foundation, we apply symbolic planning in configuration space and PDDL descriptions to develop planning algorithms for BRaPs.

3.2 Multi-Agent Path Finding

Multi-Agent Path Finding (Stern et al. 2019) is a problem of navigating a team of agents from their start locations to goal locations without collisions. The BRaP can be modeled as a complex MAPF variant that combines elements from the classic MAPF problem (Stern et al. 2019), where agents (blocks) must be moved to dedicated goal locations, the anonymous MAPF problem (Stern et al. 2019; Yu and LaValle 2013b), where goal vertices are unlabeled and are interchangeable for assigned blocks, and the Graph Motion Planning Problem (Papadimitriou et al. 1994), where unassigned blocks exists and must be moved to clear paths for assigned blocks but is limited to one motion per step. More recently, Felner and Stern (2026) introduced the Multi-Agent Path Finding with Unassigned Agents (MAPFUA), which formally defines this setting where assigned agents (our assigned blocks) have goals and others (our unassigned blocks) can move to clear the way. Our BRaP can be seen as a specific, challenging variant of MAPFUA, characterized by extremely dense environments, strict movement constraints, and task assignments. For a broader discussion of the MAPFUA problem and its potential variants, we direct readers to the MAPFUA paper (Felner and Stern 2026).

Different from the anonymous MAPF problem, assigned blocks may be assigned distinct goal vertex sets thus goal vertices are not fully interchangeable. This aspect is related to the Target Assignment and Path Finding (TAPF) problem (Ma and Koenig 2016), where agents in a team can be assigned to any of one of the team's dedicated goal vertices. However, the TAPF problem does not include unassigned agents or blocks.

The BRaP is made further challenging by the high-density environment, where unassigned blocks often occupy the vast majority of the workspace, and by its specific operational constraints prohibiting following. Although "following conflicts" are defined in the literature, most solvers ignore them and it is non-trivial to incorporate them into certain MAPF algorithms. These combined factors mean that even though powerful, state-of-the-art solvers like *LaCAM* (Okumura 2023b,a, 2024) are effective for high-density MAPF problems, they are not directly applicable to BRaPs. Therefore, building upon these advanced frameworks, we develop a MAPF-based solution tailored to the specific complexities of the BRaP.

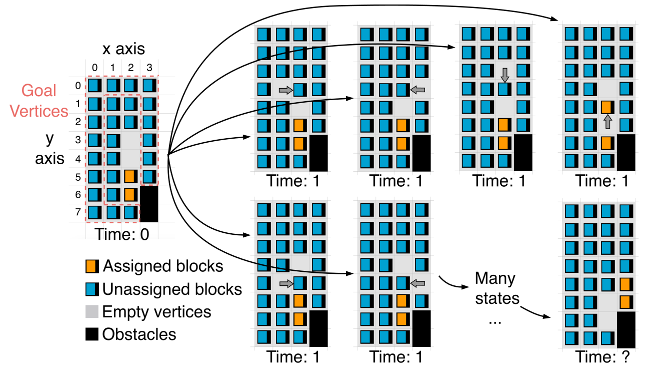


Figure 4: A graph of configuration states and actions.

4 Applicable Methodologies

4.1 Configuration Space Search

The BRaP can be formulated as a graph search in configuration space. Starting from an initial grid configuration, available actions transform the state until reaching a goal configuration that satisfies desired criteria. Actions at each time step represent edges that transition between neighboring configuration states. Each state comprises time, assigned block set (vertices containing assigned blocks), empty vertex set, and completed block set (vertices containing completed assigned blocks).

An example graph of configuration states and actions is illustrated in Figure 4. According to the above definition, the root state in the graph is: time: 0, assigned blocks: $\{$(2,5), (2,6)$\}$, empty vertices: $\{$(2,3), (2,4)$\}$, completed blocks: $\emptyset$. The assigned block set in the goal state should be empty.

The configuration space can be modified using the three actions described in Sec. 2.1. To ensure computational tractability and reduce the branching factor, we restrict the system to one action per time step. This restriction renders the *wait* action redundant and thus disallowed. Methods supporting multiple actions at each time step will be discussed in subsequent sections.

A configuration graph can be constructed from the defined states and actions, as shown in Figure 4. With the cost of the actions defined, graph search techniques like Dijkstra and A* algorithms can be applied to find the minimum cost path from the initial state to the goal state (where there is no longer any assigned blocks).

A* with an admissible heuristic function is guaranteed to be complete (always find a solution if there is) and optimal. We formulate an admissible heuristic by assuming assigned blocks move along least-blocking vertices, with each blocking block requiring only one *move* action for removal. The detailed heuristic formulation is presented in Appendix.

4.2 PDDL-based Configuration Space Search

The configuration space search can also be described using PDDL. We define five predicates: `(emp ?u)`, `(asb ?u)`, `(blk ?u)`, `(cmp ?u)`, and `(goal ?u)`, indicating whether a vertex $u \in V$ contains an empty space, assigned block, unassigned block, completed block, or potential goal, respectively. The predicate `(adjacent ?u ?v)` denotes that vertices u and v are adjacent to each other, i.e., $(u, v) \in E$. Three actions are defined in the PDDL domain: `(move_blk ?u ?v)` and `(move_asb ?u ?v)` for moving unassigned and assigned blocks, respectively, and `(complete ?u)` for completion.

In the problem file, vertices serve as objects, denoted as `node-0-0`, `node-0-1`, etc., excluding vertices containing obstacles. The initial state specifies each node's type as either `(asb node-x-y)`, `(blk node-x-y)`, or `(emp node-x-y)`. All vertices except assigned blocks are marked with `(cmp node-x-y)`, and valid terminal locations are marked with `(goal node-x-y)`. The goal state requires all vertices to be marked as `(cmp node-x-y)`, with total action cost as the optimization metric.

While this formulation allows simultaneous moves, the `fast-downward` solver used in Sec. 5 generates totally-ordered plans with single actions per timestep. Formulations that support multi-agent temporal planning would introduce prohibitively high branching factors for PDDL solvers, motivating the restriction to single-action plans.

Algorithm 1: Priority-based search

```
1 Input: Grid $G(V, E)$, assigned block set $\mathcal{I}$, goal vertices
    $V_i (i \in \mathcal{I})$
2 $\mathcal{I}_{\text{sorted}} \leftarrow computePriority(\mathcal{I}, V_i, V, E)$
3 $P \leftarrow \emptyset$ //The initial plan
4 for $i \in \mathcal{I}_{sorted}$ do
5     $\xi \leftarrow computeConstraint(P)$
6     $p_i \leftarrow computeConstrainedPlan(i, V_i, V, E, P, \xi)$
7     $P \leftarrow P \cup p_i$
8 return $P$
```

4.3 Priority-based Configuration Space Search

In contrast to the single action per time step approach of Sec. 4.1, we now develop a formulation allowing one action per assigned block per time step, enabling parallel execution and reducing rearrangement makespan.

Algorithm 1 begins by assigning priorities based on block proximity to goals, as our experiments showed this produces better solutions. The priority for each assigned block $i \in \mathcal{I}$ is determined using the heuristic $h(s, i)$ described in Appendix, where smaller values indicate higher priorities.

The search iteratively creates rearrangement plans for individual assigned blocks while respecting plans of higher-priority blocks as constraints. For a prior plan P, the *computeConstraint*(P) function generates constraints ensuring the preconditions of higher-priority actions are satisfied. For example, if $move(i, t, u, v)$ appears in P, a constraint in ξ ensures block i occupies u at time t while v remains empty, and no other actions can modify these vertices at time t. The *computeConstrainedPlan*() function generates plans by pruning states violating ξ and treating prior plans as external forces, ensuring parallel executability of the resulting plans.

Algorithm 2: Heuristic approach

```
1  Input: Grid $G(V, E)$, assigned block set $\mathcal{I}$, goal vertices
       $V_i (i \in \mathcal{I})$
2  $\mathcal{I}_{\text{sorted}} \leftarrow$ computePriority($\mathcal{I}, V_i, V, E$)
3  $P \leftarrow \emptyset$ //The initial plan
4  for $i \in \mathcal{I}_{sorted}$ do
5      $\psi \leftarrow$ computeVertexBlockingTime($P$)
6      $U_i \leftarrow$ leastBlockingPath($i, V_i, V, E, P$)
7      $p_i \leftarrow \emptyset$
8      for $u \in U_i$ do
9          $p_i^u \leftarrow$ createEmptyVertexAt($u, V, E, P, \psi$)
10         $p_i \leftarrow p_i \cup p_i^u \cup$ move($i, t, u_{\text{previous}}, u$)
11     $P \leftarrow P \cup p_i \cup$ complete($i, t, u$)
12 return $P$
```

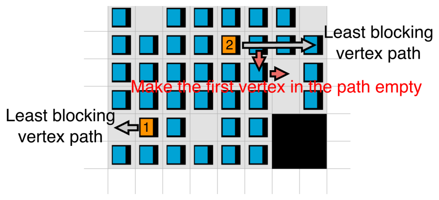


Figure 5: Move assigned blocks along the least blocking path.

4.4 Heuristic Approach: Moving Along the Least Blocking Path

Given the exponentially growing configuration search space with respect to the block number, algorithms in the previous three sections do not scale efficiently to large grids with numerous assigned blocks. In this section, we propose a pure heuristic-based approach that moves each assigned block along the least-blocking vertex path.

Algorithm 2 outlines the heuristic approach. The process begins by allocating priorities to assigned blocks using the method described in Sec. 4.3. As illustrated in Figure 5, for each assigned block, the planner first creates a least-blocking path, $U_i = \{u_i^1, \cdots, u_i^{|U_i|}\}$, to the goal vertices. Then, for each vertex along this path, $u_i^l \in U_i$, the planner create an empty vertex at u_i^l by iteratively moving blocks to the closest empty vertex and subsequently moves the assigned block to the emptied vertex u_i^l, thus generating a block movement plan. A least-blocking path is defined as the lowest-cost path where assigned blocks, unassigned blocks, and empty vertices are assigned different traversal costs.

During planning, the plan for the current assigned block i is created based on the map status where all previous assigned block plans have been executed. However, executing the plan for assigned block i does not require waiting for complete execution of all previous plans. Instead, it only requires waiting for the completion of plans affecting vertices that will be used by plan p_i. The function $t = \psi(u)$ takes a vertex $u \in V$ and returns this waiting time. By ensuring actions in p_i occur no earlier than times specified by ψ, the planner enables parallel movements of multiple assigned block.

4.5 LaCAM-based Multi-Agent Path Finding

By discretising time into timesteps and allowing multiple blocks to move to empty vertices simultaneously in each timestep, the problem is similar to a Multi-agent Path Finding problem. With the recent advances in solving large scale and high density MAPF problems, we introduce Block Rearrangement *LaCAM*, a *Lazy Constraints Addition Search* (*LaCAM*) (Okumura 2023b,a, 2024) based algorithm to solve the Block Rearrangement Problem in this section.

Lazy Constraints Addition Search *LaCAM* is a heuristic search algorithm that aims to rapidly find an initial solution and then continuously search for lower cost ones. The algorithm has three major components: (1) the high-level search tree, (2) the configuration generator, and (3) the low-level constraints. Similar to the Configuration Space Search presented in Sec. 4.1 and Figure 4, the high-level search component of *LaCAM* searches over the configuration space for solutions. A high-level search node n in *LaCAM* not only stores a configuration state $n.\pi$, but also low-level constraints $n.\xi$ which store spatial constraints that force the constrained blocks to perform the constrained actions. To rapidly find an initial solution, *LaCAM* searches in a way that combines *Depth-first Search* (DFS) and *Greedy Best-first Search* (GBFS).

However, unlike DFS that blindly expands from one node to the next or GBFS which applies a heuristic function to evaluate all possible successor states and expands the most promising one, *LaCAM* utilizes a configuration generator $f(n)$ to expand a given node n and generate a single successor node n' for exploration. $f(n)$ itself is a lightweight one step planner (e.g. PIBT (Okumura et al. 2022)) which suggests the most promising next state without enumerating all possible successor states.

If the new configuration, $n'.\pi$, has not yet been explored, the algorithm advances to n' and continues its deep dive in a DFS-style exploration, until a goal node is found. Conversely, if $n'.\pi$ has already been explored, the algorithm backtracks to the parent node n and grows its constraints, $n.\xi$. This addition forces the configuration generator, $f(n)$, to generate an alternative successor configuration state. $n.\xi$ grows in a brute-force manner to gradually take over the control of blocks from the configuration generator, and eventually force the exploration of all possible successor nodes of n if $f(n)$ keeps returning explored states.

***BR-PIBT* Configuration Generator** With the existence of following conflicts, PIBT (Okumura et al. 2022) is no longer applicable as a configuration generator for *LaCAM* in BRaP. This is because PIBT recursively decides the actions for chains of movable blocks, while, due to following conflicts, only the blocks next to empty vertices are movable. The additional complexity is the BRaP has multiple goal vertices for each assigned block, and the configuration generator needs to decide which goal vertex to move towards. In this section, we introduce *BR-PIBT* as the configuration generator for Block Rearrangement *LaCAM* (*BR-LaCAM*).

The core intuition of the *BR-PIBT* algorithm is that every block takes turns to call for empty vertices in a priority-based manner, ensuring that each block is processed exactly

Algorithm 3: *BR-PIBT*

```
1  Input: Grid $G(V, E)$, assigned block set $\mathcal{I}$, unassigned set
     $\mathcal{J}$, goal vertices $V_i$ $(i \in \mathcal{I})$, priorities $\omega$, current states $\pi$,
     next states $\pi'$, current temp goals $g$,
2  UpdatePriorities($\mathcal{I}, V_i, \omega, \pi$);
3  $g' \leftarrow \emptyset$; //Next temp goals
4  for each block $k \in \mathcal{I}$ in descending priority order do
5      if $\pi'_k = \perp$ then BR_PIBT(null, $k$) ;
6  return $\pi'$, $g'$
7  BR_PIBT(parent, block $i$): begin
8      $\pi'_i \leftarrow \pi_i$; //Default action wait
9      if $i \in \mathcal{I}$ then $g'_i \leftarrow$ if $g_i$ is null or occupied in $g'$ then
         closest goal $v \in V_i, v \notin g'$ else $g_i$ ;
10     if no unused empty vertices left then return true ;
11     $C \leftarrow$ neighbors($\pi_i$) $\cup \{\pi_i\}$; //Candidate locations

12     for each $u \in C$ do AssignPreferences($u, i, g'_i$) ;
13     Sort $C$ by preference in ascending order;
14     for each candidate $u \in C$ do
15         if $u$ is occupied in $\pi'$ then continue ;
16         if $parent \neq null$ and $u = \pi_i$ then continue;
17         $j \leftarrow$ the block occupies $u$ in $\pi$;
18         if $j \neq null$ and $j \neq i$ then
19             if $\pi'_j \neq \perp$ or not BR_PIBT($i, j$) then
                 continue;
20         if $j = null$ then $\pi'_i \leftarrow u$; //Move if empty
21         return true;
22     return false;
23 UpdatePriorities($\mathcal{I}, V_i, \omega, \pi$): begin
24     for each $i \in \mathcal{I}$ do
25         $\omega_i \leftarrow$ if $\pi_i \in V_i$ or $\omega_i = null$ then $rand(0, 1)$
             else $\omega_i + 1.0$
26 AssignPreferences($u, i, g'_i$): begin
27     $u.preference \leftarrow$ if $i \in \mathcal{I}$ then
         $(Distance(u, g'_i), DistanceToEmpty(u))$ else
         $(DistanceToEmpty(u), random())$
```

once per step to maintain algorithmic efficiency. This approach ensures that all blocks have the opportunity to make progress toward their goals by dynamically updating their priorities. Algorithm 3 describes the process of *BR-PIBT*.

The *BR-PIBT* algorithm operates as a single-step planner that generates the next configuration state by iteratively assigning actions and temporary goals to blocks based on dynamically managed priorities. The algorithm maintains three key components: *1) Priority Management, 2) Recursive Planning and 3) Temporary Goal Allocation*.

`1) Priority Management:` Similar to PIBT, each block i maintains a priority value ω_i that determines the order in which blocks are processed. Priorities are dynamically updated based on goal achievement and movement status. When a block i reaches any vertex in V_i, its priority is reset to a floating number in $(0, 1)$. When the block i is displaced from its goal vertices V_i, its priority is incremented by 1. This priority update process happens at the beginning of each configuration generation phase (Algorithm 3 line 2 and 23).

`2) Recursive Planning:` The algorithm processes assigned blocks according to their priority (Algorithm 3 lines 4 - 5). For each block, it attempts to move to the most preferred adjacent location. If the desired spot is occupied, the algorithm makes a recursive call on the blocking block, finding a chain of displacement requests that reaches an empty vertex (Algorithm 3, lines 7 - 22). During this process, the algorithm returns the default action *wait* if no empty vertices are left or the chain of displacement is not possible. It then identifies available neighboring vertices and sorts them based on action preference metrics (Algorithm 3 lines 11 - 12). The *wait* action is only considered as a preferred option if the block is not being actively pushed by another block to find the chain (Algorithm 3 line 16). The process ensures that a block remains stationary unless a move is strategically advantageous for itself or required to accommodate another block's movement.

`3) Temporary Goal Allocation:` For assigned blocks, *BR-PIBT* allocates temporary goals to guide them. When a block decides its action, a temporary goal is selected from the closest unallocated goal locations if the block does not have one or if its goal was taken away by a higher priority block (Algorithm 3 line 9).

The action preference differentiates between assigned blocks and unassigned blocks (Algorithm 3 line 26). Assigned blocks prioritize moves that minimize the distance to their goals, while also considering proximity to empty vertices for tie breaking. Unassigned blocks focus primarily on maintaining access to empty vertices to facilitate overall system mobility. The algorithm's efficiency stems from its guarantee that each block is processed at most once per timestep. When a block is called recursively to facilitate movements, it is immediately decided an action, thus is marked as processed.

Integration with LaCAM The *BR-PIBT* configuration generator integrates seamlessly with the *LaCAM* framework. For each high-level node expansion, *LaCAM* uses *BR-PIBT* to generate a single successor configuration state. If explored state are generated and low-level constraints are imposed, *LaCAM* forces *BR-PIBT* to satisfy the low-level constraints by first setting the next states in π' as the constrained locations of constrained blocks, then calling *BR-PIBT* for decisions on unconstrained blocks. The combination of *LaCAM* and *BR-PIBT*, called *BR-LaCAM*, enables the rapid exploration of the configuration space while still maintaining *LaCAM*'s completeness guarantee.

Claim 1: *BR-LaCAM* is complete, which returns a solution for solvable Block Rearrangement Problems.

Proof. Following (Okumura 2023b), the search space is finite and the number of configuration states is $|V|^{|\mathcal{I} \cup \mathcal{J}|}$. Since the low-level constraints enumerate all possible combinations of movements for all blocks, it generates all possible configuration states that connects to a high-level node's configuration state. Consequently, all reachable configuration states from the start configuration state are explored. □

Parameter	Setup
Grid size	4x10, 6x10, 8x10, 10x10, 20x20, 40x40, 80x80
Assigned blocks number	Minimum: 1 Maximum: 12.5% of grid vertices
Empty vertex number	Minimum: 1 Maximum: 25% of grid vertices
Goal type	Goal B: All boundary vertices Goal R1: Random goals, $1\times$ # of assigned blocks Goal R2: Random goals, $2\times$ # of assigned blocks
Random	10 cases for each parameter combination

Table 2: Experimental setup parameters. For Goal B, the maximum assigned block number is limited to either 12.5% of $|V|$ or twice the grid height, whichever is smaller.

Anytime Improvement The other advantage of *LaCAM* is its anytime search ability. After an initial solution is found, it uses this initial solution cost as an upper bound and continues to explore the search space for a lower cost solution.

5 Results and Discussion

In this section, we evaluate the solution quality, computational time, and scalability of the algorithms proposed in Sec. 4 using an extensive set of 13,860 test cases. We systematically vary grid size, number of assigned blocks, number of empty vertices, and goal location selection methods. Table 2 lists the parameters used for generating these test cases, with representative examples visualized in Figure 6. Goal type B represents boundary vertex goals, while types R1 and R2 denote randomly selected goals, with type R1 being more restrictive in selection. Obstacles are modeled as a square region occupying 1/5 of the grid length, positioned in the bottom right corner of the grid. The PDDL formulation is solved using `fast-downward`[1]. All algorithms are evaluated on a single machine with AMD EPYC 7R13 Processor. The time limit for one test case is 10 seconds, in line with the typical convergence time of the algorithms.

For each test case, success indicates finding a feasible plan within the time limit. To compare solution quality across algorithms, we establish a baseline for each test instance using the algorithm that finds the lowest final composite cost. Each algorithm's final composite cost is divided by the baseline to calculate cost ratios (resulting in ratios always greater than 1). Makespan ratios are calculated in a similar way.

Table 3 summarizes the success rate, composite cost ratio, and makespan ratio of the proposed algorithms across different grid configurations (13,860 test cases). The *BR-LaCAM* achieves the highest success rate, followed by the heuristic, priority, config, and PDDL algorithms. Only *BR-LaCAM* and heuristic algorithms successfully generate solutions for large instances. Overall, *BR-LaCAM* and heuristic algorithms achieve the lowest composite cost and makespan ratios, followed by priority, config, and PDDL approaches. Note that, the *BR-LaCAM* algorithm exhibits a higher standard deviation in composite cost on grid sizes 20×20 to 80×80 and for goal type R1. This variability indicates that for instances with goal vertices sparsely distributed and on large maps, *BR-PIBT*'s myopic behavior may leads to substantially higher solution costs than its average performance.

As shown in Table 3, algorithm success rates decrease with increasing grid size. Only the *BR-LaCAM* and heuristic algorithms successfully solve large instances (grid size $\geq 20\times20$). It is important to note that the maximum number of assigned blocks increases proportionally with grid size. Notably, *BR-LaCAM* failures occur only when the grid size or assigned block number exceeds computational capacity within the 10-second time limit, confirming its completeness as an algorithm - it fails only due to time constraints rather than algorithmic limitations. Not reflected in the table, increasing the number of empty vertices reduces search depth, empirically improving success rates and reducing both composite cost and makespan. Table 3 also demonstrates that algorithms achieve higher success rates with goal types B and R2. Goal type R1, due to its more restrictive selection criteria, results in relatively lower success rates. Figure 7 visualizes the composite cost ratios across all 13,860 test instances, with each instance represented by a single dot. Consistent with the results in Table 3, failure rates increase with grid size and number of assigned blocks. The Cost Ratios among algorithms typically range between 1 and 10 relative to the best performer.

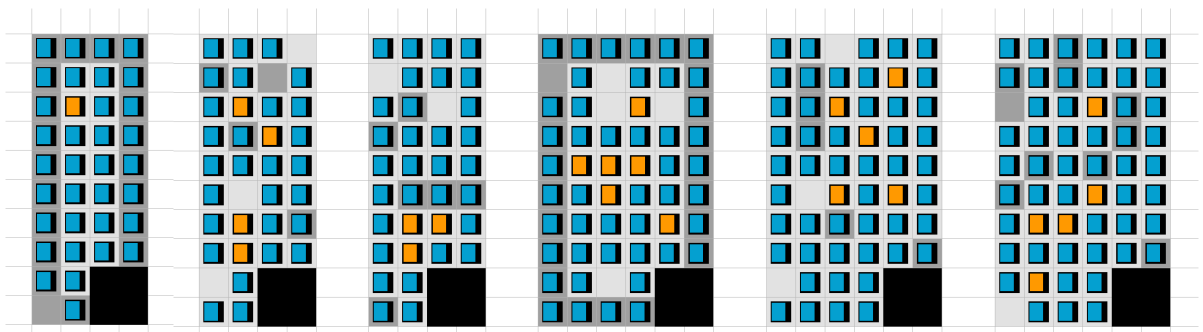
Figure 8 illustrates the time required for each algorithm to report its first solution. The *BR-LaCAM* and heuristic-based algorithms find solutions within 1 millisecond for over 50% of the test cases. *BR-LaCAM*'s initial solution time exceeds 1 second in fewer than 10% of cases.

6 Conclusion

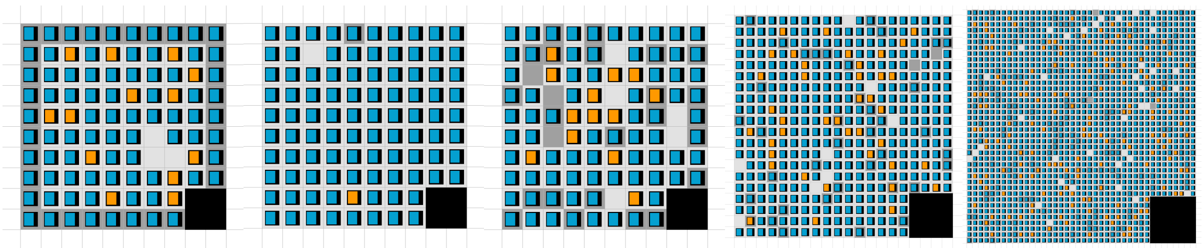
This paper formally defines the Block Rearrangement Problem and establishes its relationship to sliding puzzles. We develop five solution algorithms: configuration space search, PDDL-based planning, priority-based search, *BR-LaCAM*, and heuristic-based methods. Through extensive evaluation across 13,860 test cases with varying grid sizes, assigned block numbers, and goal configurations, we evaluate the effectiveness of the proposed approaches and provide insights into their relative strengths. While most algorithms show degraded success rates with increasing grid size and more restrictive goal choices, the *BR-LaCAM* and heuristic-based approaches successfully solve large instances, achieving 99% and 93% success rates respectively with dense grids from 4×10 to 80×80. Among the proposed methods, the *BR-LaCAM* and heuristic approaches demonstrate superior performance in both solution quality (composite cost and makespan) and computational efficiency, finding solutions within 1 millisecond for over 50% of test cases and within 1 second for over 90% of cases. These results establish a comprehensive baseline for solving Block Rearrangement Problems.

Based on our comprehensive analysis, future research directions include exploring various problem variants in-

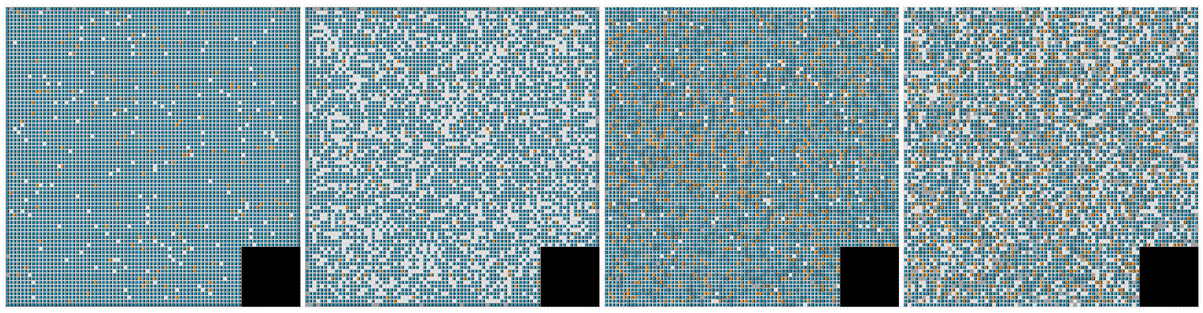
[1]Fast-downward: https://fast-downward.org



(a) 4×10 (b) 4×10 (c) 4×10 (d) 6×10, 6 assigned (e) 6×10, 5 assigned (f) 6×10, 5 assigned



(g) 10×10, 13 assigned, 3 empties (h) 10×10, 1 assigned, 1 empty vertex (i) 10×10, 13 assigned, 8 empties (j) 20×20, 40 assigned, 10 empties (k) 40×40, 160 assigned, 40 empties



(l) 80×80, 128 assigned, 160 empties (m) 80×80, 128 assigned, 1600 empties (n) 80×80, 800 assigned, 160 empties (o) 80×80, 800 assigned, 1600 empties

Figure 6: Example test cases. Dark gray vertices indicate the goals for the assigned blocks. Black vertices indicate obstacles.

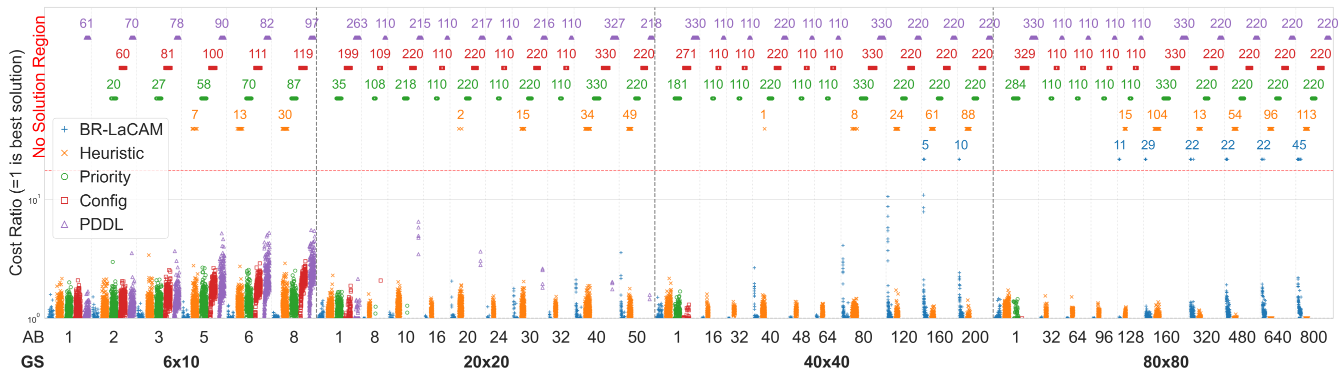


Figure 7: Composite Cost Ratio for each instance, grouped by number of assigned blocks (AB) and grid size (GS). The number and dots in the no solution region shows how many instances are unsolved for each algorithm in each group.

Groups	*BR-LaCAM*			Heuristic			Priority			Config			PDDL		
	Succ. Rate	Comp. Cost	Make-span	Succ. Rate	Comp. Cost	Make-span	Succ. Rate	Comp. Cost	Make-span	Succ. Rate	Comp. Cost	Make-span	Succ. Rate	Comp. Cost	Make-span
4×10	**100**	**1.00**$_{0.03}$	**1.01**$_{0.04}$	93	1.22$_{0.31}$	1.34$_{0.59}$	91	1.27$_{0.31}$	1.53$_{0.61}$	87	1.46$_{0.38}$	1.77$_{0.76}$	89	1.39$_{0.46}$	1.72$_{0.85}$
6×10	**100**	**1.01**$_{0.04}$	**1.00**$_{0.03}$	97	1.22$_{0.26}$	1.40$_{0.54}$	87	1.27$_{0.26}$	1.53$_{0.55}$	76	1.48$_{0.41}$	1.82$_{0.87}$	76	1.55$_{0.73}$	1.96$_{1.33}$
8×10	**100**	**1.01**$_{0.03}$	**1.00**$_{0.03}$	97	1.23$_{0.24}$	1.45$_{0.57}$	75	1.27$_{0.23}$	1.54$_{0.55}$	60	1.53$_{0.48}$	1.90$_{1.08}$	54	1.63$_{0.93}$	2.01$_{1.43}$
10×10	**100**	**1.00**$_{0.03}$	**1.00**$_{0.02}$	96	1.22$_{0.23}$	1.50$_{0.63}$	60	1.26$_{0.21}$	1.56$_{0.57}$	42	1.56$_{0.60}$	1.96$_{1.38}$	34	1.70$_{1.27}$	2.04$_{1.67}$
20×20	**100**	**1.01**$_{0.10}$	**1.00**$_{0.02}$	95	1.19$_{0.17}$	1.67$_{0.69}$	15	N/A	N/A	7	N/A	N/A	4	N/A	N/A
40×40	**99**	**1.04**$_{0.51}$	**1.01**$_{0.15}$	91	1.11$_{0.12}$	1.56$_{0.55}$	8	N/A	N/A	3	N/A	N/A	0	N/A	N/A
80×80	**92**	**1.04**$_{0.12}$	**1.01**$_{0.04}$	80	1.06$_{0.09}$	1.34$_{0.38}$	2	N/A	N/A	0	N/A	N/A	0	N/A	N/A
Goal B	**100**	**1.01**$_{0.03}$	**1.00**$_{0.03}$	97	1.13$_{0.14}$	1.37$_{0.44}$	56	1.20$_{0.21}$	1.45$_{0.46}$	49	1.49$_{0.47}$	1.94$_{1.05}$	45	1.51$_{0.88}$	1.89$_{1.28}$
Goal R1	**98**	**1.02**$_{0.34}$	**1.01**$_{0.10}$	82	1.27$_{0.29}$	1.63$_{0.74}$	33	1.29$_{0.28}$	1.51$_{0.64}$	22	1.35$_{0.33}$	1.46$_{0.62}$	22	1.45$_{0.82}$	1.71$_{1.16}$
Goal R2	99	**1.02**$_{0.10}$	**1.01**$_{0.04}$	**100**	1.16$_{0.22}$	1.42$_{0.52}$	56	1.29$_{0.27}$	1.56$_{0.61}$	47	1.53$_{0.48}$	1.85$_{1.02}$	44	1.57$_{0.76}$	1.96$_{1.35}$
All	**99**	**1.02**$_{0.21}$	**1.01**$_{0.06}$	93	1.18$_{0.23}$	1.46$_{0.59}$	48	1.26$_{0.26}$	1.51$_{0.57}$	39	1.48$_{0.46}$	1.81$_{1.00}$	37	1.52$_{0.82}$	1.88$_{1.29}$

Table 3: Algorithm performance across different configurations grouped by grid sizes and goal types, including success rate, mean composite cost ratio, and mean makespan ratio. The subscript values are standard deviations. Smaller ratios indicate better performance. Bold values indicate the best performance across all algorithms. Ratio metrics are omitted for success rates below 20%. The algorithm abbreviations are as follows. *BR-LaCAM*: *LaCAM* with *BR-PIBT*, Heuristic: heuristic approach, Priority: priority-based configuration space search, Config: configuration space search, PDDL: PDDL-based configuration space search.

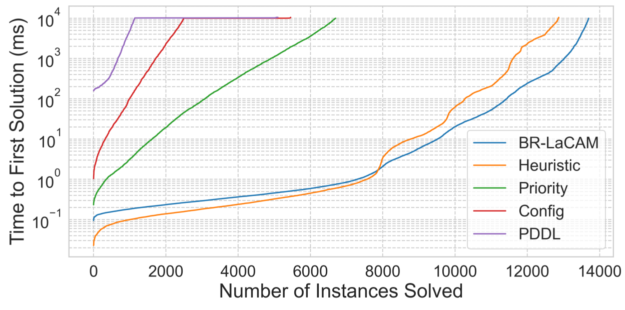


Figure 8: Time to find the first solution.

cluding different terminal states (where assigned blocks become empty vertices, obstacles, or unassigned blocks), additional objective functions, constraints on simultaneous actions, and practical priority requirements. Given *BR-LaCAM* approach's superior performance in this work, improving its solution quality and scalability through enhanced goal selection processes and informative action preference heuristics emerges as a particularly promising direction.

Appendix: Admissible Heuristic for Configuration Space Search

An admissible heuristic is formulated by assuming assigned blocks move along least-blocking vertices and each blocking block can be cleared using only one *move* action. Using the example state s in Figure 9, we demonstrate the heuristic value calculation for two assigned blocks. We calculate a lower bound cost for moving each assigned block to its goal set. Block 1 requires: an initial *move*, clearing the block at (3, 4) (minimum one *move*), a final *move*, and a *complete* action. Therefore, the lower bound cost for block 1 is $h(s,1) = c_{move} + c_{move} + c_{move} + c_{complete} = 3c_{move} + c_{complete}$. Similarly, the movement of block 2 to the goals requires two *move* actions and one *complete* action, yielding a lower bound cost of $h(s,2) = c_{move} + c_{move} + c_{complete} = 2c_{move} + c_{complete}$. Since actions moving one assigned block can benefit the movement of other assigned block, we estimate a joint cost lower bound by averaging individual costs. Thus, for this example: $h(s) = (h(s,1) + h(s,2))/2 = [(3c_{move} + c_{complete}) + (2c_{move} + c_{complete})]/2 = 2.5c_{move} + c_{complete}$.

Let the lowest action cost to move the assigned blocks to the goals be $h^*(s)$.

Claim 2: $h(s)$ is admissible: $h(s) \leq h^*(s)$.

Proof. Let the lowest cost to move each assigned block $i \in \mathcal{I}$ to their goal vertices be $h^*(s,i)$. By definition, $h(s,i)$ underestimates the cost to move block i. Therefore, $h(s,i) \leq h^*(s,i) \leq h^*(s)$. Therefore, $h(s) = \frac{1}{|\mathcal{I}|}\sum_{i \in \mathcal{I}} h(s,i) \leq \frac{1}{|\mathcal{I}|}|\mathcal{I}|h^*(s) = h^*(s)$. □

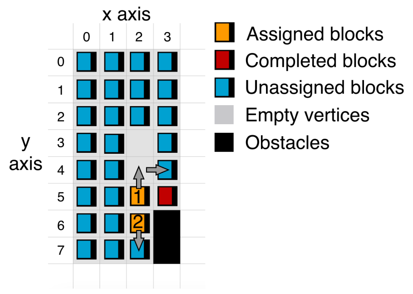


Figure 9: Illustration of the formulated heuristic. The goal of the assigned blocks is the boundary.

References

Agia, C.; Migimatsu, T.; Wu, J.; and Bohg, J. 2023. Stap: Sequencing task-agnostic policies. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 7951–7958. IEEE.

Cian, L.; Dreossi, T.; Dovier, A.; et al. 2022. Modeling and solving the rush hour puzzle. In *CEUR Workshop Proceedings*, volume 3204, 294–306. CEUR-WS.

Damani, M.; Luo, Z.; Wenzel, E.; and Sartoretti, G. 2021. PRIMAL _2: Pathfinding via reinforcement and imitation multi-agent learning-lifelong. *IEEE Robotics and Automation Letters*, 6(2): 2666–2673.

Davesa Sureda, C.; Espasa Arxer, J.; Miguel, I.; and Villaret Auselle, M. 2024. Towards High-Level Modelling in Automated Planning. *arXiv e-prints*, arXiv–2412.

Estivill-Castro, V.; and Ferrer-Mestres, J. 2013. Pathfinding in dynamic environments with PDDL-planners. In *2013 16th International Conference on Advanced Robotics (ICAR)*, 1–7. IEEE.

Felner, A.; and Stern, R. 2026. Multi-Agent Path Finding with Unassigned Agents (MAPFUA). In *Proceedings of the AAAI Conference on Artificial Intelligence*.

Flake, G. W.; and Baum, E. B. 2002. Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1-2): 895–911.

Garrett, C. R.; Chitnis, R.; Holladay, R.; Kim, B.; Silver, T.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Integrated task and motion planning. *Annual review of control, robotics, and autonomous systems*, 4(1): 265–293.

Gozon, M.; and Yu, J. 2024. On computing makespan-optimal solutions for generalized sliding-tile puzzles. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 10288–10296.

Guo, T.; and Yu, J. 2023. Toward efficient physical and algorithmic design of automated garages. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 1364–1370. IEEE.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173(5-6): 503–535.

Jiang, H.; Wang, Y.; Veerapaneni, R.; Duhan, T.; Sartoretti, G.; and Li, J. 2025. Deploying Ten Thousand Robots: Scalable Imitation Learning for Lifelong Multi-Agent Path Finding. In *2025 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2022. MAPF-LNS2: Fast repairing for multi-agent path finding via large neighborhood search. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, 10256–10265.

Li, Q.; Gama, F.; Ribeiro, A.; and Prorok, A. 2020. Graph neural networks for decentralized multi-robot path planning. In *2020 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 11785–11792. IEEE.

Lin, K.; Agia, C.; Migimatsu, T.; Pavone, M.; and Bohg, J. 2023. Text2motion: From natural language instructions to feasible plans. *Autonomous Robots*, 47(8): 1345–1365.

Lu, C.; Zeng, B.; and Liu, S. 2020. A study on the block relocation problem: Lower bound derivations and strong formulations. *IEEE Transactions on Automation Science and Engineering*, 17(4): 1829–1853.

Ma, H.; Harabor, D.; Stuckey, P. J.; Li, J.; and Koenig, S. 2019. Searching with consistent prioritization for multi-agent path finding. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, 7643–7650.

Ma, H.; and Koenig, S. 2016. Optimal Target Assignment and Path Finding for Teams of Agents. In Jonker, C. M.; Marsella, S.; Thangarajah, J.; and Tuyls, K., eds., *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems, Singapore, May 9-13, 2016*, 1144–1152. ACM.

Muppasani, B.; Pallagani, V.; and Srivastava, B. 2024. Solving the Rubik's Cube with a PDDL Planner. *ICAPS 2024 System's Demonstration track*.

Okumura, K. 2023a. Improving LaCAM for scalable eventually optimal multi-agent pathfinding. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence*, 243–251.

Okumura, K. 2023b. Lacam: Search-based algorithm for quick multi-agent pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 11655–11662.

Okumura, K. 2024. Engineering LaCAM*: Towards Real-time, Large-scale, and Near-optimal Multi-agent Pathfinding. In *Proceedings of the 23rd International Conference on Autonomous Agents and Multiagent Systems*, 1501–1509.

Okumura, K.; Machida, M.; Défago, X.; and Tamura, Y. 2022. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence*, 310: 103752.

Papadimitriou, C. H.; Raghavan, P.; Sudan, M.; and Tamaki, H. 1994. Motion planning on a graph. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, 511–520. IEEE.

Ratner, D.; and Warmuth, M. 1986. Finding a shortest solution for the N x N extension of the 15-puzzle is intractable. In *Proceedings of the Fifth AAAI National Conference on Artificial Intelligence*, 168–172.

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

Shen, B.; Chen, Z.; Li, J.; Cheema, M. A.; Harabor, D. D.; and Stuckey, P. J. 2023. Beyond Pairwise Reasoning in Multi-Agent Path Finding. In *International Conference on Automated Planning and Scheduling*.

Shoham, Y.; and Elidan, G. 2021. Solving sokoban with forward-backward reinforcement learning. In *Proceedings of the International Symposium on Combinatorial Search*, volume 12, 191–193.

Silver, T.; Chitnis, R.; Tenenbaum, J.; Kaelbling, L. P.; and Lozano-Pérez, T. 2021. Learning symbolic operators for task and motion planning. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3182–3189. IEEE.

Srivastava, S.; Li, C.; Lingelbach, M.; Martín-Martín, R.; Xia, F.; Vainio, K. E.; Lian, Z.; Gokmen, C.; Buch, S.; Liu, K.; et al. 2022. Behavior: Benchmark for everyday household activities in virtual, interactive, and ecological environments. In *Conference on robot learning*, 477–490. PMLR.

Stern, R.; Sturtevant, N. R.; Felner, A.; Koenig, S.; Ma, H.; Walker, T. T.; Li, J.; Atzmon, D.; Cohen, L.; Kumar, T. K. S.; Boyarski, E.; and Barták, R. 2019. Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *ArXiv*, abs/1906.08291.

Wang, Y.; Duhan, T.; Li, J.; and Sartoretti, G. 2025. LNS2+RL: Combining multi-agent reinforcement learning with large neighborhood search in multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, 23343–23350.

Yu, J.; and LaValle, S. 2013a. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 27, 1443–1449.

Yu, J.; and LaValle, S. M. 2013b. Multi-agent path planning and network flow. In *Algorithmic Foundations of Robotics X: Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics*, 157–173. Springer.

Zawalski, M.; Góral, G.; Tyrolski, M.; Wiśnios, E.; Budrowski, F.; Cygan, M.; Kuciński, Ł.; and Miłoś, P. 2024. What Matters in Hierarchical Search for Combinatorial Reasoning Problems? *arXiv preprint arXiv:2406.03361*.