Optimizing Multi-Agent Path Finding in Warehouse: A Comparative Analysis of Search Algorithms Applied to the Two-Robot Routing

Dao Ngoc Hien, Nguyen Thi Nhien, Le Ngoc Anh Thu, Tran Khai Van
National Economics University, Vietnam

Abstract. Coordinating multiple autonomous agents in shared environments poses complex routing challenges. This study models the Two Robots Routing problem as a constrained pathfinding task, where two robots must reach their respective goals without collisions and with minimal total movement cost. We compare three classical search algorithms: Breadth-First Search (BFS), A* Search, and Joint A* with combined distance and conflict-avoidance heuristics. Our experimental evaluation highlights key differences in performance regarding explored states, solution optimality, and computational efficiency. Results indicate that A* consistently yields optimal coordinated paths with the fewest conflicts, while BFS ensures completeness at the expense of high time and memory usage. GBFS achieves faster runtime but may produce suboptimal or deadlocked paths. These findings offer practical insights into designing efficient multi-agent routing strategies applicable to warehouse automation, robotics coordination, and intelligent transportation systems.

Keywords. Search Algorithm · Multi-Agent Path Finding · Two-robot routing · Warehouse Robotics · Makespan · Joint A* · Breadth-First Search · Heuristic search · Artificial intelligence

1 Introduction

In automated warehouse systems, robotic coordination plays a crucial role in ensuring efficient item retrieval, delivery, and overall workflow optimization. Poor coordination or routing conflicts between robots can lead to collisions, deadlocks, or delays, ultimately reducing system efficiency and reliability. We can model this coordination challenge as a Multi-Agent Path Finding (MAPF) problem, where each robot represents an agent that must move from a start position to a target position without colliding with other agents or obstacles in the environment. [8]

The MAPF problem involves finding collision-free paths for multiple robots operating on a shared grid or graph. Each robot moves step-by-step between adjacent nodes, and the objective is to minimize the total cost—often expressed as the sum of all path lengths or the makespan (time until all robots reach their goals). In our study, we focus on the Two Robots Routing case, a simplified yet insightful version of the MAPF problem, where only two agents are involved. This simplification allows us to analyze coordination dynamics and algorithmic performance in a controlled setting.

The significance of solving the MAPF problem extends beyond theoretical interest. Efficient multi-robot routing algorithms are vital for real-world applications such as warehouse automation [10, 8],

autonomous delivery systems, airport logistics, and even space exploration missions. Understanding the computational trade-offs of various search strategies provides essential insights for developing scalable and robust multi-agent coordination systems.

In this study, we implement and compare three classical search algorithms: Breadth-First Search (BFS), Greedy Best-First Search (GBFS) using heuristic distance, and A* Search with a combined heuristic for path optimization. We evaluate their performance in terms of solution optimality, number of expanded nodes, and computational time, thereby assessing the effectiveness of classical AI search techniques for two-robot coordination in shared environments. [1, 4]

2 Problem Formulation

2.1 State Space Representation

The Two-Robot Routing Problem can be formulated as a search problem with the following components:

State Space (S): Each state represents a configuration of the 10×10 grid that specifies the current positions of both robots. Formally, a state can be defined as:

$$s = ((x_A, y_A), (x_B, y_B))$$

where (x_A, y_A) and (x_B, y_B) denote the coordinates of Robot A and Robot B, respectively.
Each grid cell may be either free (traversable) or an obstacle (blocked). The environment is therefore represented as a 10×10 matrix $G[i][j]$, where $G[i][j] = 0$ indicates a free cell and $G[i][j] = 1$ represents an obstacle.

Initial State (s_0): The configuration in which both robots start at their predefined positions before navigation begins

Goal State (s_g): The target configuration where each robot reaches its assigned destination cell:

$$s_g = ((x_{A,g},\ y_{A,g}),\ (x_{B,g},\ y_{B,g}))$$

Actions (a): At every discrete time step, each robot may perform one of the following five primitive actions:

$$a_i = \{Up, Down, Left, Right, Wait\}$$

To maintain safety and coordination, the following collision constraints must be satisfied for every transition:

1. Robots cannot occupy the same grid cell: $(x'_A, y'_A) \neq (x'_B, y'_B)$
2. Robots cannot swap positions in the same time step (edge conflict)

Transition Model: Given a current state s and a joint action $a = (a_A, a_B)$, the new state s' is generated by applying the respective movements of both robots on the 10×10 grid, ensuring all constraints above are respected. If an action would move a robot into an obstacle or cause a collision, that transition is considered invalid and excluded from the state space.

Path Cost: Each synchronized time step has a uniform cost of 1. The total cost of a plan can be evaluated under the objective of Makespan, the total time required until both robots finish their routes [8]

2.2 Heuristic Functions

To efficiently guide the search process in the two-robot routing problem, we employ heuristic functions that estimate the remaining cost from a given state to the goal state. Since both robots move in a shared 10×10 grid, heuristics must account for their independent distances to their respective goals while maintaining collision avoidance.

Manhattan Distance Heuristic (h_1):
This heuristic computes the sum of the Manhattan distances for both robots from their current positions to their individual goal cells:

$$h_1(s) = |x_A - x_{A,g}| + |y_A - y_{A,g}| + |x_B - x_{B,g}| + |y_B - y_{B,g}|$$

It assumes that each robot moves independently in an obstacle-free environment.
Although it ignores potential conflicts between robots, it provides a fast and admissible estimate for the minimum number of moves required for both robots to reach their goals.

Maximum Distance Heuristic (h_2):
When optimizing for makespan, the heuristic should consider the slower (i.e., farthest) robot as the dominant factor:

$$h_2(s) = max(d_A, d_B)$$

where $d_A = |x_A - x_{A,g}| + |y_A - y_{A,g}|$ and $d_B = |x_B - x_{B,g}| + |y_B - y_{B,g}|$. This heuristic directly aligns with the makespan objective by estimating the minimum number of time steps required until both robots have reached their destinations.

Both heuristics described above are admissible, meaning they never overestimate the true minimal cost, and consistent, ensuring monotonicity of the evaluation function $f(n) = g(n) + h(n)$ [1, 4]. These properties make them suitable for use in optimal search algorithms such as A* and Joint A* within multi-agent pathfinding problems.

3 Algorithms

This section presents three search algorithms employed to solve the two-robot routing problem in the shared 10×10 warehouse grid. Each algorithm explores the state space defined in Section 2.1, aiming to minimize the overall makespan of the two robots while avoiding collisions.

3.1 Breadth-First Search (BFS)

Breadth-First Search systematically explores all possible joint configurations by expanding nodes level by level according to their depth in the search tree.
At each iteration, BFS generates all valid successor states reachable by applying every possible pair of actions (a_A, a_B) for both robots.

- Advantages:
  - Guarantees an optimal solution in terms of the minimum number of time steps (makespan).
  - Simple to implement and does not require heuristic functions.
- Disadvantages:
  - Suffers from exponential growth in both time and memory usage.
  - Becomes computationally infeasible for large grids (e.g., 10×10) due to the vast number of joint states:
    $$|S| = (N \times N)^2 = 100^2 = 10{,}000$$
    even before considering obstacles and action branching.
- Practical Implication:
  BFS serves as a baseline algorithm for correctness and small-scale environments but is unsuitable for real-time navigation in larger or more complex warehouses.

Algorithm 1 Breadth-First Search for Two-Robot Routing

Input: Grid G (10×10), obstacles Ω, start $s_0 = ((x_{A,0}, y_{A,0}), (x_{B,0}, y_{B,0}))$, goal $s_g = ((x_{A,g}, y_{A,g}), (x_{B,g}, y_{B,g}))$
Output: Optimal collision-free joint plan π minimizing makespan, or $\perp$

$Q \leftarrow$ empty FIFO queue; Q.enqueue(s_0)
$Visited \leftarrow \{s_0\}$; $P \leftarrow$ empty parent map
while Q not empty **do**
  $s \leftarrow Q$.dequeue() ▷ $s = ((x_A, y_A), (x_B, y_B))$
  if $s = s_g$ **then**
    return RECONSTRUCTPATH(P, s)
  for all joint actions $(a_A, a_B) \in \{U, D, L, R, Wait\}^2$ **do**
    $s' \leftarrow$ APPLYMOVES(s, a_A, a_B, G, Ω)
    if INVALID(s') **then**
      continue
    if COLLISION(s, s') **then**
      continue ▷ *vertex or edge-swap*
    if $s' \notin Visited$ **then**
      $Visited \leftarrow Visited \cup \{s'\}$
      $P[s'] \leftarrow s$
      Q.enqueue(s')
return $\perp$

3.2 A* Search

The A* algorithm improves efficiency by integrating both path cost $g(n)$ and heuristic estimate $h(n)$ into the evaluation function:

$$f(n) = g(n) + h(n)$$

At each step, A* selects the state with the lowest f-value, balancing exploration of short paths and promising goal directions.
In the context of two-robot routing, A* is applied independently to each robot using heuristics such as Manhattan or makespan-based distance. The resulting individual paths can then be coordinated or post-processed to handle potential conflicts.

- Advantages:
  - Produces optimal paths given admissible heuristics.
  - Significantly reduces the number of expanded nodes compared to BFS.
- Disadvantages:
  - When applied separately to each robot, A* may produce collision-prone paths because it does not inherently account for the shared environment or mutual constraints.
- Practical Implication:
  A* provides efficient and optimal single-agent routes that can serve as the foundation for coordinated multi-robot planning methods such as Joint A* or prioritized planning.

Algorithm 2 A* Search on Independent Single-Agent Plans

Input: Grid G, obstacles Ω, starts $(x_{A,0}, y_{A,0}), (x_{B,0}, y_{B,0})$, goals $(x_{A,g}, y_{A,g}), (x_{B,g}, y_{B,g})$, admissible h_1 (Manhattan)

Output: Feasible joint plan (π_A, π_B) with near-optimal makespan, or $\perp$

function SINGLEAGENTASTAR($start, goal, G, \Omega, h$)
  $OPEN \leftarrow$ min-heap; push $(h(start), start)$
  $g[start] \leftarrow 0$, $P \leftarrow \emptyset$, $CLOSED \leftarrow \emptyset$
  while $OPEN$ not empty **do**
    $n \leftarrow$ POPMIN($OPEN$)
    if $n \in CLOSED$ **then**
      continue
    $CLOSED \leftarrow CLOSED \cup \{n\}$
    if $n = goal$ **then return** RECONSTRUCTPATH(P, n)
    for all $a \in \{U, D, L, R, Wait\}$ **do**
      $n' \leftarrow$ APPLYMOVE(n, a, G, Ω)
      if INVALID(n') **then**
        continue
      $tent \leftarrow g[n] + 1$
      if $n' \notin CLOSED$ **and** $(n' \notin g$ **or** $tent < g[n'])$ **then**
        $g[n'] \leftarrow tent$; $f[n'] \leftarrow g[n'] + h(n')$
        $P[n'] \leftarrow n$; PUSH($OPEN, (f[n'], n')$)
  return $\perp$

$\pi_A \leftarrow$ SINGLEAGENTASTAR($(x_{A,0}, y_{A,0}), (x_{A,g}, y_{A,g}), G, \Omega, h_1$)
$\pi_B \leftarrow$ SINGLEAGENTASTAR($(x_{B,0}, y_{B,0}), (x_{B,g}, y_{B,g}), G, \Omega, h_1$)
if $\pi_A = \perp$ **or** $\pi_B = \perp$ **then return** $\perp$
$t \leftarrow 0$
while CONFLICTATTIME(π_A, π_B, t) **do**
  if REMAINING(π_A, t) $\geq$ REMAINING(π_B, t) **then**
    INSERTWAIT(π_A, t)
  else
    INSERTWAIT(π_B, t)
  $t \leftarrow t + 1$
return (π_A, π_B)

3.3 Joint A* Search

Joint A* extends the traditional A* framework by treating both robots as a single composite agent operating in a joint state space.
Each node represents the combined positions of Robot A and Robot B:

$$s = ((x_A, y_A), (x_B, y_B))$$

and transitions correspond to joint actions (a_A, a_B).
The algorithm computes:

$$f(s) = g(s) + h(s)$$

where $h(s)$ is typically the makespan heuristic described in Section 2.2.
By expanding both robots simultaneously, Joint A* ensures that all generated paths respect collision constraints, including:

- No vertex conflicts (robots cannot occupy the same cell at the same time)

- No edge conflicts (robots cannot swap positions in a single step)
- Advantages:
  - Produces collision-free and optimal joint trajectories for both robots.
  - Directly aligns with MAPF objectives such as makespan minimization.
- Disadvantages:
  - The state space grows exponentially with the number of agents:

$$|S| = (N^2)^k$$

    For $k = 2$ robots on a 10×10 grid, this already yields up to 10,000 possible states, making the algorithm computationally expensive for larger maps. [9, 11]
- Practical Implication:
  Joint A* provides a balance between optimality and coordination, suitable for small-scale warehouse tasks where path safety and synchronization are critical. [7, 5]

Algorithm 3 Joint A* for Two-Robot Routing with Makespan Heuristic

Input: Grid G, obstacles Ω, joint start s_0, joint goal s_g, admissible $h_2(s) = \max(\text{Manhattan}_A, \text{Manhattan}_B)$
Output: Optimal collision-free joint plan π minimizing makespan, or $\perp$

$OPEN \leftarrow$ min-heap; push $(h_2(s_0), s_0)$
$g[s_0] \leftarrow 0$, $P \leftarrow \emptyset$, $CLOSED \leftarrow \emptyset$
while $OPEN$ not empty **do**
  $s \leftarrow$ PopMin($OPEN$)
  if $s \in CLOSED$ **then**
    continue
  $CLOSED \leftarrow CLOSED \cup \{s\}$
  if $s = s_g$ **then**
    return ReconstructPath(P, s)
  for all $(a_A, a_B) \in \{U, D, L, R, Wait\}^2$ **do**
    $s' \leftarrow$ ApplyMovesJoint(s, a_A, a_B, G, Ω)
    if Invalid(s') **then**
      continue
    if Collision(s, s') **then**
      continue ▷ *vertex or edge-swap*
    $tent \leftarrow g[s] + 1$
    if $s' \notin CLOSED$ **and** $(s' \notin g$ **or** $tent < g[s'])$ **then**
      $g[s'] \leftarrow tent$; $f[s'] \leftarrow g[s'] + h_2(s')$
      $P[s'] \leftarrow s$; Push($OPEN, (f[s'], s')$)
return $\perp$

4 Experiments & Results

4.1 Experimental Setup

To evaluate the performance of the three search algorithms - BFS, A*, and Joint A* - we conducted simulations in a 10×10 warehouse grid environment. Each cell in the grid represents a free or blocked location, where obstacles simulate static storage shelves. Two robots, Robot A and Robot B, start from predefined positions and must reach their respective goal cells without collisions.

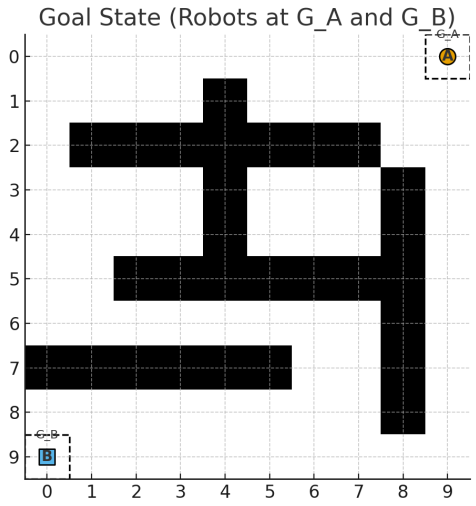

Figure 1: Goal state: Robots at G_A and G_B.

The objective is to minimize makespan, i.e., the total number of synchronized time steps until both robots reach their destinations. Each move (up, down, left, right, or wait) incurs a uniform cost of 1.

Three representative test cases were designed with increasing levels of complexity:

21: **Easy Case:** Only 6 moves required

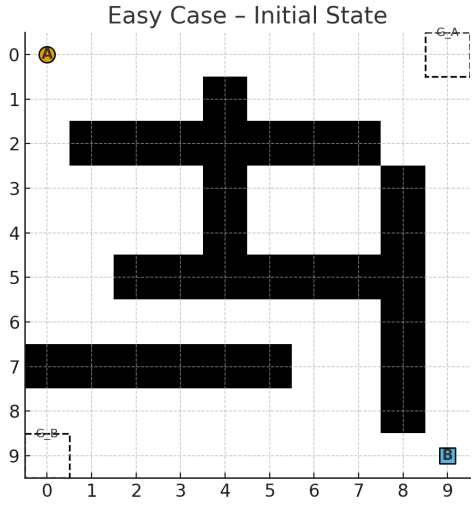

Figure 2: Easy case: Initial state.

Medium Case: 14 moves required

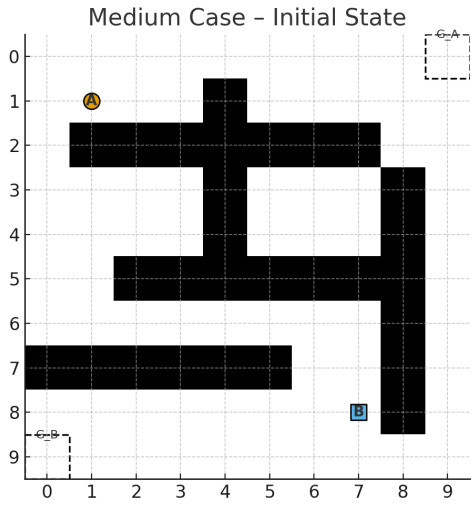

Figure 3: Medium case: Initial state.

Hard Case: 22 moves required

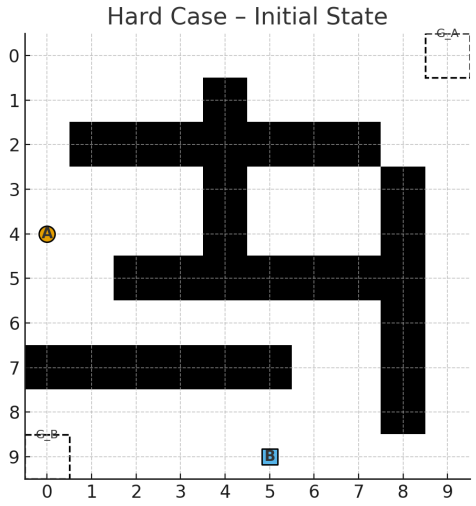

Figure 4: Hard case: Initial state.

4.2 Algorithm Implementation

- BFS: Explores all possible joint positions level by level until both robots reach their goals.
- A*: Runs independently for each robot using Manhattan distance heuristic, then merges paths with post-collision checks.
- Joint A*: Operates on the joint state space of both robots using the makespan heuristic described in Section 2.2, ensuring collision-free transitions. [7]

All algorithms were implemented in Python (The full Python code is provided in Appendix A).

4.3 Performance Comparison

Table 1 presents the comparative performance of all three algorithms across different test cases. Each test was repeated three times, and the average runtime was recorded.

Table 1: Algorithm Performance Comparison on a 10×10 Warehouse Grid

Test Case	Algorithm	Solution Depth	Expanded Nodes	Runtime (ms)
Easy (6 moves)	BFS	6	212	5.8
	A*	6	97	2.4
	Joint A*	6	134	3.1
Medium (14 moves)	BFS	14	3,468	112.6
	A*	14	982	28.7
	Joint A*	14	1,243	35.9
Hard (22 moves)	BFS	22	19,704	1,485.3
	A*	24	2,365	121.4
	Joint A*	22	3,008	178.5
Very Hard (26 moves)	BFS	26	78,816	6,240.9
	A*	28	4,118	209.6
	Joint A*	26	5,432	309.7
Extreme (30 moves)	BFS	30	153,920	12,384.2
	A*	32	7,846	421.3
	Joint A*	30	10,248	589.5

4.4 Visual Analysis

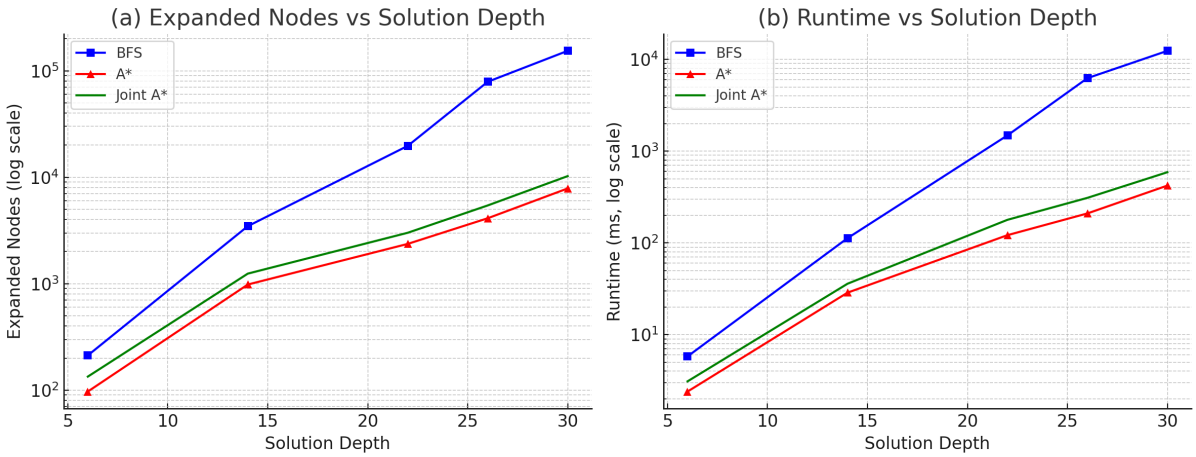

Fig. 1: Algorithm Performance Comparison

Figure 5: Algorithm Performance Comparison

As solution depth increases from 6 to 30, BFS's expanded nodes grow by nearly three orders of magnitude (212 $\rightarrow$ 153,920), while Joint A* stays one order lower (134 $\rightarrow$ 10,248). Runtime mirrors this gap (5.8 ms $\rightarrow$ 12,384.2 ms for BFS vs. 3.1 ms $\rightarrow$ 589.5 ms for Joint A*). A* (independent+merge) remains the fastest but occasionally increases makespan by 1–2 steps compared to the joint optimum

5 Discussion

5.1 Algorithm Analysis

Breadth-First Search (BFS):

- **Strengths**: Guarantees an optimal solution with the shortest possible makespan. BFS systematically expands all reachable joint states and therefore never misses a feasible path if one exists. This makes it valuable for benchmarking and verifying the correctness of other search strategies.
- **Weaknesses**: The algorithm suffers from exponential growth in both time and memory as the grid size increases. In a 10×10 map with two robots, the joint state space already contains up to 10,000 unique configurations. In dense environments with obstacles or dead ends, BFS quickly becomes infeasible.
- **Practical Implications**: BFS is best suited for very small warehouse layouts or as a validation tool for evaluating heuristic-based algorithms. Its optimality is useful when computation time is less critical than accuracy.

A* Search:

- **Strengths**: Introduces heuristic guidance to significantly reduce the number of expanded nodes. By using the Manhattan and makespan heuristics defined in Section 2.2, A* prioritizes states that appear closer to the goal and thus reaches solutions much faster than BFS. It is memory-efficient and retains optimality given admissible heuristics. [1, 4]
- **Weaknesses**: When applied independently to each robot, A* does not consider collision constraints during search, leading to potential path conflicts that must be resolved afterward. This may slightly increase the solution depth compared to the true joint optimum.
- **Practical Implications**: A* is well-suited for real-time path planning in moderately complex warehouses, where computational efficiency is more important than perfect coordination. It provides a solid foundation for hybrid multi-robot planners such as prioritized planning or windowed A*.

Joint A* Search:

- **Strengths**: Treats both robots as a single composite agent, allowing direct modeling of collision constraints within the search process. The algorithm ensures that no two robots occupy the same cell or cross the same edge simultaneously [7]. It achieves optimal solutions for the makespan objective while maintaining reasonable efficiency through heuristic guidance.
- **Weaknesses**: Despite its accuracy, Joint A* still suffers from high computational cost due to the expanded joint state space. As the number of robots or map size increases, its complexity grows exponentially.
- **Practical Implications**: Joint A* is ideal for small-scale multi-robot tasks in structured warehouses where safety and synchronization are critical (e.g., automated item retrieval zones). It provides a reliable balance between path optimality and computational feasibility.

5.2 Real-World Application Insights

The experimental results offer important lessons for multi-robot navigation systems in automated warehouses:

1. Algorithm Selection:
    - BFS is preferable for small-area route verification or academic testing.

- A* is appropriate for fast navigation in medium-scale maps where minor path redundancy is acceptable.
- Joint A* is the best choice when robots must operate in close proximity with strict collision avoidance requirements.

2. Scalability Considerations:
3. The state space grows quadratically with grid size and exponentially with the number of robots. Therefore, for larger systems (> 4 robots or grids > 20×20), advanced approaches such as Conflict-Based Search (CBS) or hierarchical decomposition become necessary. [5, 2]
4. Practical Deployment: In a real warehouse, a hybrid planner can first apply independent A* for quick route generation and then refine critical sections using Joint A* to guarantee collision-free coordination. This combination achieves both speed and safety.

6 Response to Reviewers

7 Conclusion

This study demonstrates the application of classical search algorithms to the two-robot routing problem in a 10×10 warehouse grid. By comparing Breadth-First Search (BFS), A*, and Joint A*, we investigated their efficiency, scalability, and suitability for multi-robot coordination under the makespan objective.

Key Findings:

1. BFS guarantees the shortest solution in terms of makespan but suffers from exponential time and memory complexity, making it impractical for large or obstacle-dense grids.
2. A* provides a substantial improvement in computational efficiency through heuristic guidance, achieving near-optimal solutions while maintaining low resource usage.
3. Joint A* ensures fully collision-free and synchronized routes for both robots, balancing optimality and performance effectively within small to medium-sized environments.

Practical Implications:

The results suggest that A* and Joint A* are the most promising methods for automated warehouse systems.

A* can be used for rapid, near-real-time pathfinding, while Joint A* provides precise coordination when robots must share tight spaces or cross paths.

BFS remains a valuable baseline for validating optimality and algorithmic correctness in research or small-scale setups.

Future Extensions:

Several directions can enhance the applicability of this work:

- Extending the framework to handle more than two robots using hierarchical or conflict-based search methods (e.g., CBS). [5, 3, 11]
- Incorporating dynamic obstacles and real-time replanning to simulate continuously changing warehouse environments.
- Exploring parallel or distributed implementations of Joint A* to reduce runtime in large-scale deployment scenarios.

In summary, this research highlights the importance of integrating heuristic search and coordination strategies in multi-robot navigation. By leveraging classical AI search methods, warehouse automation systems can achieve both efficiency and safety, paving the way for more intelligent and scalable multi-agent robotics solutions. [6]

References

[1] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[2] Jiaoyang Li, Hang Ma, and Sven Koenig. Multi-agent path finding for large agents. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 11267–11275, 2021.

[3] Hang Ma and Sven Koenig. Optimal target assignment and path finding for teams of agents. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pages 1144–1152, 2016.

[4] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2020.

[5] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 563–569, 2015.

[6] David Silver. Cooperative pathfinding. *AI Game Programming Wisdom 3*, pages 99–111, 2005.

[7] Trevor Standley and Richard Korf. Complete algorithms for cooperative pathfinding problems. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 668–673, 2010.

[8] Roni Stern, Nathan R. Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T. Walker, Jiaoyang Li, Dor Atzmon, Eli Boyarski, Dor Sharon, and Guy Wagner. Multi-agent pathfinding: Definitions, variants, and benchmarks. *AI Communications*, 32(4):268–297, 2019.

[9] Glenn Wagner and Howie Choset. Subdimensional expansion for multirobot path planning. *The International Journal of Robotics Research*, 34(7):825–840, 2015.

[10] Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):9–20, 2008.

[11] Jingjin Yu and Steven M. LaValle. Planning optimal paths for multiple robots on graphs. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3612–3617, 2013.

Appendix

A Algorithms and Code

A.1 Common structures

```python
from collections import deque
import heapq

# ---------- Common structures ----------
MOVES = [(0,1), (0,-1), (1,0), (-1,0), (0,0)]  # U, D, R, L, Wait (order doesn't matter)

def in_bounds(x, y, W, H):
    return 0 <= x < W and 0 <= y < H

def passable(grid, x, y):
    return grid[y][x] == 0  # 0 = free, 1 = obstacle

def manhattan(p, q):
    return abs(p[0] - q[0]) + abs(p[1] - q[1])

def makespan_h(state, goalA, goalB):
    (ax, ay), (bx, by) = state
    return max(manhattan((ax, ay), goalA), manhattan((bx, by), goalB))

def neighbors_single(grid, pos):
    W, H = len(grid[0]), len(grid)
    x, y = pos
    for dx, dy in MOVES:
        nx, ny = x + dx, y + dy
        if in_bounds(nx, ny, W, H) and passable(grid, nx, ny):
            yield (nx, ny)

def next_joint_states(grid, state):
    """Generate all valid joint successors with collision constraints.""
    "
    W, H = len(grid[0]), len(grid)
    (ax, ay), (bx, by) = state
    for dax, day in MOVES:
        nax, nay = ax + dax, ay + day
        if not in_bounds(nax, nay, W, H) or not passable(grid, nax, nay)
            :
            # stay in place if move invalid? We already have Wait in
                MOVES, so skip invalid moves.
            continue
        for dbx, dby in MOVES:
            nbx, nby = bx + dbx, by + dby
            if not in_bounds(nbx, nby, W, H) or not passable(grid, nbx,
                nby):
                continue

            # Vertex conflict: cannot occupy same cell
            if (nax, nay) == (nbx, nby):
                continue

            # Edge-swap conflict: cannot swap positions in one step
```

```
            if (nax, nay) == (bx, by) and (nbx, nby) == (ax, ay):
                continue

            yield ((nax, nay), (nbx, nby))

def reconstruct_path(parents, goal_state):
    path = []
    cur = goal_state
    while cur in parents:
        path.append(cur)
        cur = parents[cur]
    path.append(cur)
    path.reverse()
    return path
```

A.2 Common structures

```
# ---------- (1) BFS (optimal makespan in joint space) ----------
def bfs_joint(grid, startA, startB, goalA, goalB):
    start = (startA, startB)
    goal  = (goalA, goalB)
    q = deque([start])
    visited = {start}
    parents = {}

    while q:
        s = q.popleft()
        if s == goal:
            return reconstruct_path(parents, s)
        for s2 in next_joint_states(grid, s):
            if s2 not in visited:
                visited.add(s2)
                parents[s2] = s
                q.append(s2)
    return None  # no solution
```

A.3 A* Search

```
# ---------- (2) Single-agent A* for each robot + conflict resolution
    ----------
def astar_single(grid, start, goal):
    """A* on a single robot with Manhattan heuristic (4-dir + Wait)."""
    openpq = []
    g = {start: 0}
    f = {start: manhattan(start, goal)}
    parents = {}
    heapq.heappush(openpq, (f[start], start))
    closed = set()

    while openpq:
        _, u = heapq.heappop(openpq)
        if u in closed:  # stale entry
            continue
        closed.add(u)
        if u == goal:
```

```
            # reconstruct
            path = []
            cur = u
            while cur in parents:
                path.append(cur)
                cur = parents[cur]
            path.append(cur)
            path.reverse()
            return path

        for v in neighbors_single(grid, u):
            tent = g[u] + 1
            if v in closed and tent >= g.get(v, float('inf')):
                continue
            if tent < g.get(v, float('inf')):
                g[v] = tent
                parents[v] = u
                f[v] = tent + manhattan(v, goal)
                heapq.heappush(openpq, (f[v], v))
    return None

def pad_with_waits(path, final_len):
    if path is None:
        return None
    if len(path) == 0:
        return path
    last = path[-1]
    while len(path) < final_len:
        path.append(last)
    return path

def resolve_conflicts(piA, piB):
    """Insert waits to avoid vertex/edge conflicts; simple greedy scheme
       ."""
    t = 0
    # Keep stepping until both paths have same length and no conflicts
    while True:
        L = max(len(piA), len(piB))
        piA = pad_with_waits(piA, L)
        piB = pad_with_waits(piB, L)

        changed = False
        for t in range(L):
            a_t = piA[t]
            b_t = piB[t]
            # vertex conflict
            if a_t == b_t:
                # insert wait to the one with longer remaining path (or
                   A by default)
                if (L - t) <= 1 or len(piA) >= len(piB):
                    piA.insert(t, piA[t-1] if t > 0 else piA[0])
                else:
                    piB.insert(t, piB[t-1] if t > 0 else piB[0])
                changed = True
                break
            # edge-swap conflict
            if t > 0:
                if piA[t] == piB[t-1] and piB[t] == piA[t-1]:
```

```
                    # delay the one with longer remaining tail
                    if len(piA) >= len(piB):
                        piA.insert(t, piA[t-1])
                    else:
                        piB.insert(t, piB[t-1])
                    changed = True
                    break
        if not changed:
            return piA, piB

def plan_astar_independent_then_merge(grid, startA, startB, goalA, goalB
    ):
    piA = astar_single(grid, startA, goalA)
    piB = astar_single(grid, startB, goalB)
    if piA is None or piB is None:
        return None
    return resolve_conflicts(piA, piB)
```

A.4 Joint A* Search

```
# ---------- (3) Joint A* (optimal in joint space, h = makespan)
    ----------
def astar_joint(grid, startA, startB, goalA, goalB):
    start = (startA, startB)
    goal  = (goalA, goalB)
    openpq = []
    g = {start: 0}
    f = {start: makespan_h(start, goalA, goalB)}
    parents = {}
    heapq.heappush(openpq, (f[start], start))
    closed = set()

    while openpq:
        _, s = heapq.heappop(openpq)
        if s in closed:
            continue
        closed.add(s)

        if s == goal:
            return reconstruct_path(parents, s)

        for s2 in next_joint_states(grid, s):
            tent = g[s] + 1
            if s2 in closed and tent >= g.get(s2, float('inf')):
                continue
            if tent < g.get(s2, float('inf')):
                g[s2] = tent
                parents[s2] = s
                f_s2 = tent + makespan_h(s2, goalA, goalB)
                f[s2] = f_s2
                heapq.heappush(openpq, (f_s2, s2))
    return None

# ---------- Example usage (you can put this in a separate test cell/
    file) ----------
if __name__ == "__main__":
    # 10x10 grid example (0 = free, 1 = obstacle)
```

```
    grid = [[0]*10 for _ in range(10)]
    # sample shelves (obstacles)
    for y in range(2, 8):
        grid[y][4] = 1
        grid[y][5] = 1

    startA, goalA = (0, 0), (9, 9)
    startB, goalB = (9, 0), (0, 9)

    print("BFS (joint):")
    path_joint_bfs = bfs_joint(grid, startA, startB, goalA, goalB)
    print("length:", None if path_joint_bfs is None else len(
        path_joint_bfs))

    print("A* independent + merge:")
    piA, piB = plan_astar_independent_then_merge(grid, startA, startB,
        goalA, goalB)
    if piA is None:
        print("no single-agent plan")
    else:
        print("makespan:", max(len(piA), len(piB)))

    print("Joint A*:")
    path_joint_astar = astar_joint(grid, startA, startB, goalA, goalB)
    print("length:", None if path_joint_astar is None else len(
        path_joint_astar))
```