Technical Companion to

From Agent Centric to Obstacle Centric Planning: A Makespan-Optimal Algorithm for the Multi-Agent Warehouse Rearrangement Problem

A Self-Contained Guide for CS Undergraduates

Companion to the paper by Y. Sherma, E. Weiss, and O. Salzman
(SoCS 2025, Best Paper Award)

Abstract

This document explains every definition, algorithm, and theorem in the NAT-CBS paper in detail, assuming only undergraduate-level knowledge of algorithms and graph theory. No prior knowledge of Multi-Agent Path Finding (MAPF), warehouse rearrangement, network flow, or Conflict-Based Search is required—all prerequisites are developed from scratch. Each formal concept is accompanied by (1) a plain-English description, (2) a symbol-by-symbol breakdown, (3) an intuitive explanation of why the construction makes sense, and (4) concrete numerical examples on small grids.

Contents

1 Background: MAPF and Warehouse Rearrangement

1.1 What Is Multi-Agent Path Finding (MAPF)?

Multi-Agent Path Finding (MAPF) is the problem of computing collision-free paths for a group of agents on a shared graph. Each agent has a start vertex and a goal vertex, and time is discretized into integer timesteps $t = 0, 1, 2, \ldots$ At each timestep, every agent either *moves* along an edge to an adjacent vertex or *waits* at its current vertex.

Two types of **collisions** (called *conflicts*) can occur:

- **Vertex conflict:** two agents occupy the same vertex at the same timestep.
- **Edge conflict:** two agents traverse the same edge in opposite directions at the same timestep (they would "swap" positions and collide in transit).

The **makespan** of a solution is the time at which the *last* agent reaches its goal. A **makespan-optimal** solution is one with the smallest possible makespan.

Numerical Example

Consider a simple path graph with 4 vertices in a line: $A - B - C - D$.
Agent a_1 starts at A and wants to reach D. Agent a_2 starts at D and wants to reach A. Without conflicts, each agent needs 3 timesteps (three edge traversals). But they cannot simply walk toward each other—they would collide at timestep 1 or 2.
One valid solution (makespan = 5):

Time	a_1	a_2
0	A	D
1	B	C
2	B (wait)	B... *conflict!*

They cannot both be at B at time 2, so one must wait. A valid plan:

Time	a_1	a_2
0	A	D
1	A (wait)	C
2	A (wait)	B
3	A (wait)	A
4	B	A
5	C	A
6	D	A

Makespan = 6. Agent a_1 waits for a_2 to pass, then proceeds. This may not be optimal—perhaps the agents can coordinate better.

Key Idea

MAPF is fundamentally about **coordination**: even though each agent's individual shortest path is easy to compute, the agents must share the same graph, and their paths must not collide. This coordination is what makes the problem computationally hard (NP-hard in general).

1.2 What Is Multi-Agent Warehouse Rearrangement (MAWR)?

MAWR extends MAPF to a warehouse setting. Instead of agents simply traveling from start to goal, the agents must **rearrange movable obstacles** (think of boxes, pallets, or goods on

shelves) from an initial layout to a goal layout.

Key differences from MAPF:

1. **Agents move obstacles, not just themselves.** An agent can pick up an obstacle at its current location (instantaneously), carry it while moving, and set it down (also instantaneously). While carried, the obstacle moves with the agent.
2. **Tasks are not pre-assigned.** In MAPF, each agent has a specific goal. In MAWR, any agent can move any obstacle. A single obstacle might even be moved by multiple agents in sequence—one agent carries it partway, sets it down, and another agent picks it up later.
3. **The number of obstacles need not equal the number of agents.** There may be more obstacles than agents (or fewer), unlike standard MAPF where there is exactly one "task" per agent.
4. **Start and goal positions of obstacles can be arbitrary.** In MAPF variants like MAPD (Multi-Agent Pickup and Delivery), pickup and delivery locations are typically at fixed stations. In MAWR, obstacles can start and end anywhere.

Definition 1.1 (Layout)**.** Given a graph $G = (V, E)$, a **layout** $\mathcal{L} : O \to V$ is an injective function specifying the obstacles' locations in the environment. The layouts $\mathcal{L}_{\text{init}}$ and $\mathcal{L}_{\text{goal}}$ are called the *initial* and *goal* layouts, respectively.

Definition 1.2 (MAWR Instance)**.** A MAWR instance is a tuple $P = \langle G, A, O, s, \mathcal{L}_{\text{init}}, \mathcal{L}_{\text{goal}} \rangle$ where:

- $G = (V, E)$ is an undirected graph (the warehouse environment).
- $A = \{a_1, \ldots, a_n\}$ is a set of n agents.
- $O = \{o_1, \ldots, o_m\}$ is a set of m movable obstacles.
- $s : A \to V$ is an injective function giving the initial location of each agent.
- $\mathcal{L}_{\text{init}} : O \to V$ is an injective function specifying the initial location of each obstacle (the initial layout).
- $\mathcal{L}_{\text{goal}} : O \to V$ is an injective function specifying the goal location of each obstacle (the goal layout).

Symbol-by-symbol breakdown:

- $G = (V, E)$: the warehouse is modeled as a graph. Vertices V represent locations (grid cells, intersections). Edges E represent corridors where agents can walk between adjacent locations.
- A: the fleet of n robots/agents that will carry out the rearrangement.
- O: the set of m items (obstacles) that need to be relocated. The word "obstacle" is used because, from the agents' perspective, these items can block movement.
- $s : A \to V$: the function $s(a_i)$ tells you where agent a_i starts. It is *injective*, meaning no two agents start at the same vertex.
- $\mathcal{L} : O \to V$: a generic *layout*—an injective function mapping each obstacle to a location. This is the general concept; $\mathcal{L}_{\text{init}}$ and $\mathcal{L}_{\text{goal}}$ are specific instances of a layout.
- $\mathcal{L}_{\text{init}} : O \to V$: the initial layout. The function $\mathcal{L}_{\text{init}}(o_j)$ tells you where obstacle o_j is located initially. Injective—no two obstacles start at the same location.
- $\mathcal{L}_{\text{goal}} : O \to V$: the goal layout. The function $\mathcal{L}_{\text{goal}}(o_j)$ tells you where obstacle o_j must end up. The goal is to move every obstacle from $\mathcal{L}_{\text{init}}(o_j)$ to $\mathcal{L}_{\text{goal}}(o_j)$.

Agents and Obstacles Occupy Different "Levels"

The paper models two conceptual height levels: an "obstacle level" and an "agent level." An agent can share a vertex with an obstacle (it walks under/over the obstacle or picks it up). Conflicts between agents are checked at the agent level; conflicts between obstacles are checked at the obstacle level. An agent and an obstacle at the same vertex do not inherently conflict—this is how pickup works.

A Small MAWR Instance

Consider a 2×3 grid:

v_1	v_2	v_3
v_4	v_5	v_6

Agents: $A = \{a_1, a_2\}$ with $s(a_1) = v_4$, $s(a_2) = v_6$.
Obstacles: $O = \{o_{\text{red}}, o_{\text{blue}}\}$ with:

- $\mathcal{L}_{\text{init}}(o_{\text{red}}) = v_1$, $\mathcal{L}_{\text{goal}}(o_{\text{red}}) = v_3$ (red must move right).
- $\mathcal{L}_{\text{init}}(o_{\text{blue}}) = v_3$, $\mathcal{L}_{\text{goal}}(o_{\text{blue}}) = v_1$ (blue must move left).

The two obstacles need to swap positions. This requires coordination: since the grid is small, the agents must carefully sequence their pickups and moves to avoid collisions.

1.3 Paths, Plans, and Makespan

Definition 1.3 (Path)**.** A **path** $\pi = (v_0, v_1, \ldots, v_k)$ is a sequence of locations $v_i \in V$. The **length** of the path is $|\pi| := k$ (the number of timesteps, equivalently the number of edges traversed or wait actions taken).

A path is **valid** if for every consecutive pair either:

- $v_{i-1} = v_i$ (the entity waits), or
- $(v_{i-1}, v_i) \in E$ (the entity moves along an edge).

For an **agent path** $\pi_{a_i} = (v_0, v_1, \ldots, v_k)$, we additionally require $v_0 = s(a_i)$—the agent starts at its initial position. (Agents have no fixed goal; they just need to help move obstacles.)

For an **obstacle path** $\pi_{o_j} = (v_0, v_1, \ldots, v_k)$, we require $v_0 = \mathcal{L}_{\text{init}}(o_j)$ and $v_k = \mathcal{L}_{\text{goal}}(o_j)$—the obstacle starts at its initial location and ends at its goal location.

Definition 1.4 (Plans)**.** An **obstacles plan** is a vector $\Pi_{\text{obs}} = \langle \pi_{o_1}, \ldots, \pi_{o_m} \rangle$ of m obstacle paths.

An **agents plan** is a vector $\Pi_{\text{agents}} = \langle \pi_{a_1}, \ldots, \pi_{a_n} \rangle$ of n agent paths.

A **plan** is the pair $\Pi = \langle \Pi_{\text{obs}}, \Pi_{\text{agents}} \rangle$.

Definition 1.5 (Collision-Free)**.** Two paths (of agents or of obstacles) are **collision-free** if they never overlap at the same timestep. Formally, there are no vertex conflicts or edge conflicts between them (see Section 1.1 for definitions).

Π_{obs} is collision-free if all pairs of obstacle paths are collision-free. Π_{agents} is collision-free if all pairs of agent paths are collision-free.

Definition 1.6 (Realization)**.** Π_{agents} **realizes** Π_{obs} if for every movement of an obstacle along an edge, there is an agent that goes along the same edge at the same time (carrying the obstacle).

Formally: for every obstacle $o_j \in O$, for every timestep t where $\pi_{o_j}(t) \neq \pi_{o_j}(t+1)$ (the obstacle moves), there exists an agent $a_i \in A$ such that $\pi_{a_i}(t) = \pi_{o_j}(t)$ and $\pi_{a_i}(t+1) = \pi_{o_j}(t+1)$.

In other words, whenever an obstacle needs to move, some agent must be there to carry it.

Obstacles Cannot Move By Themselves

An obstacle can only change location if an agent is co-located with it and moves it. If no agent is at an obstacle's location, the obstacle stays put. This is the fundamental coupling between Π_{obs} and Π_{agents}.

Definition 1.7 (Valid Plan and Makespan)**.** A plan $\Pi = \langle \Pi_{\text{obs}}, \Pi_{\text{agents}} \rangle$ is **valid** if:

1. All obstacle and agent paths are valid (follow edges or wait).
2. Π_{obs} is collision-free (no two obstacles collide).
3. Π_{agents} is collision-free (no two agents collide).
4. Π_{agents} realizes Π_{obs} (agents execute the obstacle movements).

The **makespan** of a plan is the length of the longest path:

$$\text{MKSP}(\Pi) := \max_{\pi \in \Pi} |\pi|$$

A **solution** to a MAWR instance is a valid plan. An **optimal solution** is a valid plan with minimal makespan.

Makespan Calculation

Suppose we have 2 agents and 2 obstacles, with paths:

- π_{o_1}: length 4 (obstacle 1 reaches its goal at timestep 4).
- π_{o_2}: length 3 (obstacle 2 reaches its goal at timestep 3).
- π_{a_1}: length 5 (agent 1 finishes all its work at timestep 5).
- π_{a_2}: length 4.

Then $\text{MKSP}(\Pi) = \max(4, 3, 5, 4) = 5$.

Note: the makespan is determined by the *slowest* entity. Optimal planning requires minimizing this worst case.

1.4 Why MAWR Is Harder Than MAPF

MAWR generalizes MAPF: if we set $m = n$ (one obstacle per agent), fix each agent to carry exactly one specific obstacle, and set the agents' starting positions to the obstacles' starting positions, we recover a standard MAPF instance. Since MAPF is already NP-hard, MAWR is NP-hard as well (formally shown to be NP-complete by Bachor, Bergdoll, and Nebel, 2023).

Three additional difficulties arise in MAWR beyond MAPF:

1. **Task assignment is not given.** In MAPF, agent a_i must go to goal g_i. In MAWR, any agent can move any obstacle, and the algorithm must decide which agent moves which obstacle (and when).

2. **Tasks are non-atomic.** In MAPF, each agent handles one task from start to finish. In MAWR, a single obstacle might be moved by a relay of agents—a_1 carries it halfway, parks it, and a_2 picks it up later. This dramatically expands the solution space.

3. **Two levels of coordination.** The algorithm must simultaneously plan paths for obstacles (where should each obstacle go?) and paths for agents (which agents carry which obstacles, and when?), while ensuring both plans are collision-free and the agent plan realizes the obstacle plan.

Decoupled Approaches Cannot Be Optimal

A natural idea is to first plan obstacle paths, then assign agents to carry them. The paper by Li and Ma (2023) does exactly this with their MAPF-DECOMP algorithm. While fast, this decoupled approach can **never** guarantee optimality. Why? Because the obstacle paths are planned without considering agent availability. It is easy to construct examples where the optimal obstacle paths require agents that are not in the right place, and slightly suboptimal obstacle paths would allow a much better overall solution. The paper demonstrates instances where NAT-CBS finds solutions with half the makespan of MAPF-DECOMP.

2 Agent-Centric vs. Obstacle-Centric Planning

2.1 The Agent-Centric Paradigm

Most existing algorithms for MAPF and MAPD are **agent-centric**: they assign tasks to specific agents and plan each agent's path to complete its assigned tasks.

In agent-centric planning:

1. Each task (moving an obstacle) is assigned to one specific agent.
2. That agent carries the obstacle from start to goal along a single, uninterrupted path.
3. Tasks are **atomic**—once an agent picks up an obstacle, it will carry it all the way to the goal without handing it off.

Advantage: Simpler problem structure. Once tasks are assigned, the problem reduces to standard MAPF.

Disadvantage: Rigid. The assumption that each task must be completed by a single agent rules out potentially better solutions where obstacles are moved cooperatively. This rigidity means agent-centric approaches cannot guarantee optimality for MAWR.

Why Atomicity Hurts

Consider a narrow corridor: $v_1 - v_2 - v_3 - v_4 - v_5$.
Agent a_1 is at v_1, agent a_2 is at v_5. Obstacle o_1 starts at v_2 and must reach v_4.
Atomic solution: Agent a_1 picks up o_1 at v_2, carries it to v_4. But a_2 is at v_5 and might block the corridor. We need a_2 to move out of the way first. Total time might be 4–5 steps.
Non-atomic solution: Agent a_1 picks up o_1 at v_2, moves it to v_3, and sets it down. Meanwhile a_2 comes to v_3, picks up o_1, and carries it to v_4. Both agents move toward each other, meeting in the middle—potentially faster because neither agent needs to traverse the full corridor.

2.2 The Obstacle-Centric Paradigm (This Paper's Contribution)

The key insight of this paper is to **shift the perspective from agents to obstacles**.

In obstacle-centric planning:

1. First, plan optimal paths for the *obstacles*—decide where each obstacle should be at each timestep, without worrying about which agent carries it.
2. Then, figure out how to assign agents to *realize* the obstacle movements—which agent carries which obstacle segment.

Advantage: By planning obstacle paths first, the algorithm can find globally optimal rearrangement plans. Tasks naturally become **non-atomic**: an obstacle's path is a sequence of moves, and different agents can execute different segments.

Disadvantage: Harder to solve, because the algorithm must verify that the planned obstacle movements can actually be executed by agents (the "realization" problem).

The Core Paradigm Shift

Agent-centric: "Which obstacles should each agent move, and what path should the agent take?"
Obstacle-centric: "What path should each obstacle follow, and which agents can carry it along the way?"
The obstacle-centric view decouples *what needs to happen* (obstacle paths) from *who does it* (agent assignment), enabling more flexible and potentially shorter plans.

3 Algorithmic Building Blocks

Before diving into the NAT-CBS algorithm, we need to understand two key techniques it builds upon: **Conflict-Based Search (CBS)** and **Min-Cost Max-Flow (MCMF)**.

3.1 Conflict-Based Search (CBS)

CBS is a well-known optimal algorithm for solving MAPF. It performs a **best-first search** on a *Constraint Tree (CT)*.

What is a Constraint Tree? The CT is a search tree where each node $\mathcal{N}$ stores:

1. A set of **constraints**—restrictions on where/when agents (or obstacles) can be.
2. A set of **paths**—one per agent (or obstacle), each of which is optimal given the constraints.
3. A **cost**—the makespan of the current set of paths.

How CBS works (high level):

1. **Root node:** No constraints. Each agent independently takes its shortest path. These paths may conflict.
2. **Check for conflicts:** If the current paths have no conflicts, we are done—return this solution.
3. **Split on a conflict:** If there is a conflict (say agents a_i and a_j both want to be at vertex v at time t), create two child nodes:
   - **Left child:** Add a *positive* constraint $\langle +, a_i, v, t\rangle$ (agent a_i must be at v at time t, so a_j must replan to avoid v at time t). Actually, in standard CBS, the left child adds a *negative* constraint on a_i (forbidding a_i from v at t) and the right child adds a negative constraint on a_j.
   - **Right child:** Add the complementary constraint.
4. **Replan:** In each child node, replan the constrained agent's path (respecting the new constraint) using a single-agent shortest-path algorithm like A*.
5. **Best-first search:** Insert children into a priority queue (OPEN), ordered by cost. Always expand the lowest-cost node next.
6. **Repeat** until a conflict-free node is found.

Why CBS Is Optimal

CBS always expands the node with the lowest cost first (best-first search). The first conflict-free node it finds is guaranteed to have the minimum possible cost, because all unexplored nodes have equal or higher cost. This is exactly how A* guarantees optimality—CBS is essentially A* on the space of constraint sets.

Types of constraints:

- **Positive vertex constraint** $\langle +, a, v, t\rangle$: agent a *must* be at vertex v at timestep t.
- **Negative vertex constraint** $\langle -, a, v, t\rangle$: agent a *must not* be at vertex v at timestep t.
- **Positive edge constraint** $\langle +, a, u, v, t\rangle$: agent a *must* traverse edge (u, v) at timestep t.
- **Negative edge constraint** $\langle -, a, u, v, t\rangle$: agent a *must not* traverse edge (u, v) at timestep t.

CBS on a Tiny Instance

Consider three vertices in a line: $A - B - C$.
Agent a_1: start at A, goal C. Agent a_2: start at C, goal A.
Root node (no constraints):

- $\pi_{a_1} = (A, B, C)$ — shortest path, length 2.
- $\pi_{a_2} = (C, B, A)$ — shortest path, length 2.
- Conflict! Both at B at $t = 1$ (vertex conflict).
- Cost $= \max(2, 2) = 2$.

Split: Create two children.

- Child 1: Add $\langle -, a_1, B, 1\rangle$ (a_1 cannot be at B at $t = 1$). Now a_1 must wait: $\pi_{a_1} = (A, A, B, C)$, length 3. Cost $= \max(3, 2) = 3$.
- Child 2: Add $\langle -, a_2, B, 1\rangle$. Now a_2 must wait: $\pi_{a_2} = (C, C, B, A)$, length 3. Cost $= \max(2, 3) = 3$.

Expand child 1 (cost 3): check for conflicts between (A, A, B, C) and (C, B, A, A). At $t = 1$: a_1 at A, a_2 at B—OK. At $t = 2$: a_1 at B, a_2 at A—OK. Also check edge conflicts: at $t = 1$, a_1 stays at A and a_2 moves $C \to B$—no edge conflict. At $t = 2$, a_1 moves $A \to B$ and a_2 moves $B \to A$—**edge conflict!** They swap on edge (A, B).
Must split again. Eventually CBS finds a conflict-free plan with makespan 4 (one agent waits two steps).

3.2 Min-Cost Max-Flow (MCMF)

Network flow is a classical optimization framework from combinatorial optimization.

Definition 3.1 (Min-Cost Max-Flow)**.** Given a directed graph with:

- A **source** w and a **sink** w'.
- Each edge has a **capacity** (maximum flow it can carry) and a **cost** (cost per unit of flow).

The goal is to find the maximum amount of flow from w to w', and among all maximum flows, find the one with minimum total cost.

Why is this relevant to MAWR?

The paper uses MCMF to solve the agent-assignment problem in Phase 2 (Ph2). Given a set of obstacle movements that need to happen, we need to route agents through the warehouse to execute these movements. This is naturally a flow problem:

- Each agent is one unit of flow from source to sink.
- The network encodes which locations each agent can visit at each timestep.
- Edge costs encourage agents to follow obstacle paths (realizing obstacle movements).
- Capacity constraints prevent two agents from being at the same place at the same time.

A variant with lower bounds: The paper uses a variant of MCMF where certain edges have *lower bounds* on flow, in addition to the standard upper bound (capacity). A lower bound of 1 on an edge means at least one unit of flow *must* pass through that edge. This is used to enforce that specific obstacle movements are realized by agents—if obstacle o_j moves from u to v at time t, the corresponding edge in the network must carry at least one unit of flow (at least one agent must be there).

3.3 Time-Expanded Graphs

A **time-expanded graph** is a standard technique for converting a temporal planning problem into a static graph problem.

Definition 3.2 (T-step Time-Expanded Graph)**.** Given an undirected graph $G = (V, E)$ and a time horizon T, the T-step time-expanded directed graph $G^T = (\mathcal{V}^T, \mathcal{E}^T)$ is constructed as follows.

Vertices:

$$\mathcal{V}^T = \{w, w'\} \cup \mathcal{V}^T_{\text{out}} \cup \mathcal{V}^T_{\text{in}} \cup \mathcal{V}^T_{\text{edges}}$$

where:

- w and w' are the source and sink.
- $\mathcal{V}^T_{\text{out}} = \{v_t^{\text{out}} \mid v \in V,\, 0 \le t \le T\}$ — an "out" copy of each vertex at each timestep.
- $\mathcal{V}^T_{\text{in}} = \{v_t^{\text{in}} \mid v \in V,\, 1 \le t \le T\}$ — an "in" copy of each vertex at each timestep (starting from $t = 1$).
- $\mathcal{V}^T_{\text{edges}} = \bigcup_{\substack{e \in E \\ 0 \le t \le T-1}} \{e_t^{\text{in}}, e_t^{\text{out}}\}$ — two vertices per edge per timestep.

Edges (for each undirected edge $e = (u, v) \in E$ and each timestep t):

$$\{(u_t^{\text{out}}, e_t^{\text{in}}),\ (v_t^{\text{out}}, e_t^{\text{in}}),\ (e_t^{\text{in}}, e_t^{\text{out}}),\ (e_t^{\text{out}}, u_{t+1}^{\text{in}}),\ (e_t^{\text{out}}, v_{t+1}^{\text{in}})\}$$

These encode "an agent at u or v at time t can traverse edge e and arrive at u or v at time $t+1$."

Wait edges (for each vertex v and each timestep t):

$$(v_t^{\text{out}}, v_{t+1}^{\text{in}}) \quad \text{and} \quad (v_t^{\text{in}}, v_t^{\text{out}})$$

These encode "an agent at v can wait at v" and the internal connection between in/out copies.

Source and sink edges:

- From source: (w, v_0^{out}) for every v that is a starting position of an agent.
- To sink: (v_T^{in}, w') for every $v \in V$ (agents can end anywhere at time T). (The paper actually uses (v_T^{out}, w').)

Why this construction?

Each unit of flow from w to w' in G^T corresponds to one agent's path through the warehouse over T timesteps. The capacity constraints (all edges have capacity 1) ensure no two agents can be at the same place at the same time (collision-free). The cost structure can be designed to encourage agents to realize specific obstacle movements.

Time-Expanded Graph for a Triangle

Consider graph G with vertices $\{a, b, c\}$ and edges $\{(a, b), (b, c)\}$ (a path $a - b - c$), with $T = 2$ timesteps.
The time-expanded graph has layers $t = 0, 1, 2$. At each layer, there are "out" and "in" copies of each vertex.
An agent starting at a can:

- Move $a \to b$ at $t = 0$: path goes $w \to a_0^{\text{out}} \to e_{(a,b),0}^{\text{in}} \to e_{(a,b),0}^{\text{out}} \to b_1^{\text{in}} \to b_1^{\text{out}} \to \dots$
- Wait at a at $t = 0$: path goes $w \to a_0^{\text{out}} \to a_1^{\text{in}} \to a_1^{\text{out}} \to \dots$

Each path through the time-expanded graph is a valid trajectory for one agent.

4 The NAT-CBS Algorithm

NAT-CBS (Non-Atomic Task Conflict-Based Search) is the paper's main contribution. It adapts CBS to solve MAWR optimally by iterating between two phases:

1. **Phase Ph1: Obstacle-Path Computation.** Use a CBS-like search to find optimal, conflict-free paths for the obstacles.
2. **Phase Ph2: Obstacle-Path Realization.** Given the obstacle paths, use a network-flow algorithm to find collision-free agent paths that realize the obstacle movements.

If Ph2 succeeds (agents can realize the obstacle paths), we have a valid plan. If Ph2 fails (some obstacle movement cannot be executed by any agent), a new type of conflict—a **realization conflict**—is identified and fed back to Ph1, which adjusts the obstacle paths accordingly.

4.1 Phase Ph1: Planning Obstacle Paths

Phase Ph1 uses CBS to plan paths for the *obstacles* (not agents). The CT here is searched over *obstacle constraints*.

Each CT node $\mathcal{N}$ in Ph1 contains:

1. A set of **obstacle constraints**—restrictions arising from either obstacle-obstacle conflicts or realization conflicts.
2. A set of **obstacle paths** $\mathcal{N}.\Pi_{\text{obs}}$—one path per obstacle, each optimal given the constraints.
3. The **cost** $\text{MKSP}(\mathcal{N}.\Pi_{\text{obs}})$—the makespan of the obstacle paths.

Types of constraints in Ph1:

1. **Obstacle conflicts** ($\mathcal{C}_{\text{obs}}$): These are the standard CBS conflicts, but between obstacle paths instead of agent paths. If two obstacle paths collide (vertex or edge conflict), the CT splits just like standard CBS.
2. **Realization conflicts** ($\mathcal{C}_{\text{real}}$): This is the novel conflict type introduced by the paper. A realization conflict $\langle o_i, u, v, t\rangle$ arises when Phase Ph2 determines that obstacle o_i's movement from u to v at timestep t cannot be realized by any agent. The conflict is fed back to Ph1, which splits the CT node:
   - **Positive child:** Add a positive edge constraint forcing an agent to be at (u, v) at time t to realize o_i's movement. (This ensures future agent plans will accommodate this move.)
   - **Negative child:** Add a negative constraint on o_i forbidding it from making the move (u, v) at time t. (This forces o_i to find an alternative path.)

Realization Conflicts: The Novel Ingredient

Standard CBS only handles conflicts *between* paths at the same level (agent-agent or obstacle-obstacle). Realization conflicts are *cross-level*: they arise when the obstacle plan and the agent plan are incompatible. This is the key mechanism that couples the two phases and enables NAT-CBS to find optimal solutions.
Think of it this way: Ph1 says "the obstacle should move here," Ph2 says "no agent can do that." The realization conflict forces Ph1 to either commit to the move (and find agents who can do it) or change the obstacle's path.

4.2 Phase Ph2: Realizing Obstacle Paths via Network Flow

Once Ph1 produces a conflict-free obstacles plan Π_{obs}, Phase Ph2 computes an agents plan Π_{agents} that *realizes* Π_{obs}.

The agents plan must satisfy two requirements:

- **R1:** Π_{agents} is a valid, collision-free plan satisfying all positive edge constraints of the CT node $\mathcal{N}$.
- **R2:** Π_{agents} realizes all obstacle move actions in Π_{obs}.

How Ph2 works:

Ph2 constructs a T-step time-expanded graph G^T (where T is the makespan of Π_{obs}) and formulates a Min-Cost Max-Flow with Lower Bounds problem on it.

Setting up the flow problem:

1. **Capacities:** All edges have capacity (upper bound) 1. This ensures at most one agent can use each edge at each timestep—preventing collisions.
2. **Lower bounds:** Initially all zero. For each positive edge constraint $\langle o_i, u, v, t\rangle$ in $\mathcal{N}$'s constraints, set the lower bound of the corresponding edges $(u_t^{\text{out}}, e_t^{\text{in}})$, $(e_t^{\text{in}}, e_t^{\text{out}})$, $(e_t^{\text{out}}, v_{t+1}^{\text{in}})$ to 1. This forces at least one unit of flow (one agent) through this edge at this time, ensuring the obstacle movement is realized.
3. **Costs:** Initially, set the cost of the *wait* edges $(v_t^{\text{out}}, v_{t+1}^{\text{in}})$ and the *in-to-out* edges $(v_t^{\text{in}}, v_t^{\text{out}})$ to 1, and the cost of all other edges (the edge-traversal edges) to 0. Then, for each obstacle move action in Π_{obs} (obstacle o_j moves from u to v at time t along edge $e = (u, v)$), update the cost of edges $(e_t^{\text{in}}, e_t^{\text{out}})$, $(u_t^{\text{out}}, e_t^{\text{in}})$, and $(e_t^{\text{out}}, v_{t+1}^{\text{in}})$ to 0. The effect is that traversing an edge *in the same direction* as an obstacle move is free (cost 0), waiting or staying in place costs 1, and traversing an edge *in the opposite direction* of an obstacle move costs 2 (the agent must enter and exit the edge vertices, each costing 1). This incentivizes agents to follow obstacle paths and penalizes moves that reverse obstacle movements.
4. **Supply/demand:** Set a supply of n at the source w and demand of n at the sink w'. This means n units of flow (one per agent) must travel from source to sink.

Interpreting the solution: A feasible flow of n units can be decomposed into n source-to-sink paths, each representing one agent's trajectory through the warehouse over T timesteps.

Network Flow for Agent Assignment

Consider a MAWR instance with 2 agents on a path graph $a - b - c$ with $T = 2$.
Π_{obs} says obstacle o_1 moves from b to c at $t = 0$.

In the time-expanded graph:

- The edge corresponding to "traverse (b, c) at $t = 0$" has cost 0 (free to realize this obstacle move) and lower bound 1 (some agent *must* do this).
- All other edges have cost 1 and lower bound 0.

The MCMF solver finds the cheapest way to route 2 agent-flows from source to sink while satisfying the lower bounds. One agent is forced through the $b \to c$ edge at $t = 0$ (realizing the obstacle move); the other agent takes whatever cheapest path remains.

What if the MCMF has no feasible solution?

If no feasible flow exists (the lower bounds cannot all be satisfied simultaneously while respecting capacity constraints), then the obstacles plan cannot be realized. Ph2 returns a **realization conflict**—the specific obstacle movement that could not be realized—and the algorithm returns to Ph1 to adjust the obstacle paths.

Reduced Time-Expanded Graph

In practice, Ph2 does not build the full time-expanded graph. Vertices unreachable from the source w are omitted (they represent locations at timesteps where no agent can possibly be, and no obstacle action needs to happen). This significantly reduces the graph size and improves runtime.

4.3 The Complete NAT-CBS Algorithm

We now present the complete NAT-CBS algorithm in pseudocode and walk through every line.

Algorithm 1 High-level NAT-CBS

Require: MAWR instance P
Ensure: Makespan-optimal plan Π for P

$\Pi_{\text{obs}} \leftarrow$ Optimal obs. paths ▷ no obs. constraints
$\mathcal{R} \leftarrow \langle \emptyset, \Pi_{\text{obs}}, \text{MKSP}(\Pi_{\text{obs}}) \rangle$ ▷ root CT node, no constraints
OPEN $\leftarrow \{\mathcal{R}\}$
while OPEN is not empty **do**
  $\mathcal{N} \leftarrow$ OPEN.pop() ▷ CT node with min cost
  if $\mathcal{N}.\Pi_{\text{obs}}$ has obstacle conflict $\mathcal{C}_{\text{obs}}$ **then**
    $\mathcal{N}_{\text{children}} \leftarrow \text{split}(\mathcal{N}, \mathcal{C}_{\text{obs}})$
    for each $\mathcal{N}' \in \mathcal{N}_{\text{children}}$ **do** OPEN.insert($\mathcal{N}'$)
    continue
  end if
  $\Pi_{\text{agents}} \leftarrow$ realize_obs_paths($\mathcal{N}$) ▷ Phase Ph2
  if $\Pi_{\text{agents}} = \emptyset$ **then**
    continue ▷ can't satisfy $\mathcal{N}$'s constraints
  end if
  if $(\mathcal{N}.\Pi_{\text{obs}}, \Pi_{\text{agents}})$ has realization conflict $\mathcal{C}_{\text{real}}$ **then**
    $\mathcal{N}_{\text{children}} \leftarrow \text{split}(\mathcal{N}, \mathcal{C}_{\text{real}})$
    for each $\mathcal{N}' \in \mathcal{N}_{\text{children}}$ **do** OPEN.insert($\mathcal{N}'$)
  else
    return $\Pi = \langle \mathcal{N}.\Pi_{\text{obs}}, \Pi_{\text{agents}} \rangle$
  end if
end while
return *not found*

Line-by-line walkthrough:

Lines 1–3 (Initialization):

- **Line 1:** Compute optimal paths for all obstacles, ignoring all constraints. Each obstacle independently takes its shortest path from $\mathcal{L}_{\text{init}}(o_j)$ to $\mathcal{L}_{\text{goal}}(o_j)$. These paths may conflict with each other.
- **Line 2:** Create the root CT node $\mathcal{R}$ with no constraints, the initial obstacle paths, and cost equal to the makespan of these paths.
- **Line 3:** Initialize the priority queue OPEN with just the root node.

Lines 4–5 (Main loop):

- **Line 4:** Continue as long as there are nodes to explore.
- **Line 5:** Pop the CT node with the *lowest* cost from OPEN. Since OPEN is a min-priority queue ordered by cost, this is a best-first search.

Lines 6–9 (Check for obstacle conflicts):

- **Line 6:** Check if any two obstacle paths in $\mathcal{N}.\Pi_{\text{obs}}$ conflict (vertex or edge conflict).
- **Lines 7–9:** If there is a conflict, split $\mathcal{N}$ into two children (one constraining each conflicting obstacle), insert them into OPEN, and continue to the next iteration. This is standard CBS splitting.

Lines 10–12 (Phase Ph2 — realize obstacle paths):

- **Line 10:** Invoke Phase Ph2 to compute an agents plan that realizes $\mathcal{N}.\Pi_{\text{obs}}$.
- **Lines 11–12:** If Ph2 returns no agent paths (the MCMF problem has no feasible solution, meaning the positive edge constraints of $\mathcal{N}$ cannot be satisfied), discard this node and continue. This happens when the constraints are contradictory—e.g., requiring an agent to be in two places at once.

Lines 13–17 (Check for realization conflicts):

- **Line 13:** Check if the agents plan Π_{agents} actually realizes all obstacle movements. Even though the MCMF succeeded, it may not have routed an agent through every obstacle movement (some movements had cost 0 but no lower bound forcing realization).
- **Lines 14–15:** If there is a realization conflict (an obstacle movement not realized by any agent), split $\mathcal{N}$ into two children using the realization conflict and insert them into OPEN.
- **Lines 16–17:** If no realization conflicts remain, we have a valid plan! Return it.

NAT-CBS Trace on the Paper's Toy Example

The paper provides a detailed trace using two obstacles (o_{red}, o_{blue}) and two agents (a_0, a_1) on a 5-vertex graph.
Root node $\mathcal{R}$: No constraints. Obstacle paths:

- $\pi_{o_{\text{red}}} = [b, c, e]$ (red goes $b \to c \to e$).
- $\pi_{o_{\text{blue}}} = [c]$ (blue is already at its goal, stays put).

Obstacle conflict detected: o_{red} and o_{blue} both want to be at vertex c at timestep 1 — vertex conflict $\langle o_{\text{red}}, o_{\text{blue}}, c, 1\rangle$.
$\mathcal{R}$ is split into $\mathcal{N}_1$ (positive constraint on o_{red} at c at $t = 1$) and $\mathcal{N}_3$ (negative constraint on o_{red} at c at $t = 1$).
Node $\mathcal{N}_1$: o_{red} gets positive constraint at c, $t = 1$. After replanning, o_{blue} must detour.

No obstacle conflicts remain. Ph2 is invoked. But Ph2 finds a realization conflict: o_{red}'s move from b to c at $t = 0$ cannot be realized. This generates $\mathcal{C}_{\text{real}} = \langle o_{\text{red}}, b, c, 0 \rangle$.
$\mathcal{N}_1$ splits into $\mathcal{N}_2$ (positive: force an agent to carry o_{red} at $b \to c$ at $t = 0$) and $\mathcal{N}'$ (negative: o_{red} cannot move $b \to c$ at $t = 0$). Node $\mathcal{N}'$ is infeasible (no valid path for o_{red}).
Node $\mathcal{N}_2$: Ph2 fails because the positive edge constraint creates a contradictory flow problem.
Back to $\mathcal{N}_3$: o_{red} is forbidden from c at $t = 1$, so it takes a longer path: $[b, b, c, e]$ (wait, then move). This creates another obstacle conflict with o_{blue} at c at $t = 0$. Splitting yields $\mathcal{N}_4$ and $\mathcal{N}''$.
Node $\mathcal{N}_4$: After resolving all obstacle conflicts, Ph2 finds valid agent paths. No realization conflicts. Solution found with makespan 3.

5 Makespan Optimality Proof

The paper proves that NAT-CBS returns a makespan-optimal solution for any solvable MAWR instance. The proof relies on three lemmas building up to the main theorem.

5.1 Lemma 1: The Cost Is Determined by Obstacle Paths

Lemma 5.1 (Lemma 1 in the paper)**.** *Given a realizable obstacles plan Π_{obs} with makespan T, there exists an agents plan Π_{agents} that realizes Π_{obs} with the same makespan T.*

What it says in plain English: If the obstacle paths can be realized at all, they can be realized without increasing the makespan. The agents do not need extra time beyond what the obstacles need.

Why this is true (proof sketch): By definition, if Π_{obs} is realizable, there exists some Π_{agents} that realizes it. The makespan of any realizing Π_{agents} must be $\geq T$ because every obstacle movement requires an agent, and the obstacle movements span T timesteps. If a realizing Π_{agents} had makespan $> T$, we could simply truncate the agent paths to T timesteps—the agent movements after time T are irrelevant because all obstacle movements are done by time T.

Why this matters: This lemma establishes that the cost (makespan) of a MAWR solution is *entirely determined* by the obstacles plan. The agents plan does not add to the cost. Therefore, to find an optimal solution, it suffices to find the optimal obstacles plan—the agent plan just needs to exist.

Obstacle Paths Determine the Cost

Lemma 1 means that NAT-CBS's strategy of optimizing obstacle paths in Ph1 and then finding realizing agent paths in Ph2 does not lose anything. If the optimal obstacle paths can be realized, the resulting plan has optimal makespan.

Numerical Example

Suppose Π_{obs} has makespan $T = 5$ (the longest obstacle path has 5 timesteps). If Π_{obs} is realizable, then there is an Π_{agents} with makespan ≤ 5. Agents might need to move around for timesteps 0–5 to carry obstacles, but they do not need timestep 6 or beyond. If an agent finishes carrying its last obstacle at timestep 3, it simply waits at its current location from timestep 3 to 5. Its path length is 5 (padded with waits), and the makespan is still $\max(\ldots) = 5$.

5.2 Lemma 2: Bijection Between Flows and Agent Plans

Lemma 5.2 (Lemma 2 in the paper)**.** *Let $\mathcal{N}$ be a CT node with corresponding obstacles plan Π_{obs}, and consider the MCMF problem of node $\mathcal{N}$. There is a bijection (one-to-one correspondence) from each feasible flow to a valid agents plan Π_{agents} that is consistent with the set of positive edge constraints in $\mathcal{N}$.*

What it says in plain English: Every feasible flow in the network corresponds to exactly one valid agents plan, and vice versa. The network-flow formulation perfectly captures the space of valid agent plans.

Why this is true (proof sketch): A feasible flow of n units from source to sink can be decomposed into n edge-disjoint source-to-sink paths (by the flow decomposition theorem). Each path corresponds to one agent's trajectory through the time-expanded graph. The key properties:

1. Each path is a valid agent path on G (by construction of the time-expanded graph—flow conservation ensures the path follows edges of G).
2. Each agent starts at a unique starting location (source edges go to different starting vertices).
3. The paths are collision-free (edge-disjoint paths in the time-expanded graph means no two agents share the same location at the same time, since all capacities are 1).
4. The flow satisfies lower bounds, so the paths pass through the required edges (realizing positive edge constraints).

The reverse direction is similar: given n collision-free agent paths, they induce n edge-disjoint flows in the time-expanded graph, which combine into a feasible flow of n units.

Why this matters: This lemma justifies the use of MCMF in Phase Ph2. Solving the MCMF problem is equivalent to finding the best agents plan, and the cost structure ensures agents prefer to realize obstacle movements.

5.3 Lemma 3: Flow Cost Characterizes Realization

Lemma 5.3 (Lemma 3 in the paper)**.** *Given a candidate goal node $\mathcal{N}$ with cost T, there exists a valid agents plan Π_{agents} that realizes $\mathcal{N}.\Pi_{obs}$ if and only if there exists a feasible minimum-cost flow with total cost $(n \cdot T - R)$, where R is the total number of obstacle move actions in $\mathcal{N}.\Pi_{obs}$.*

What it says in plain English: We can determine whether the obstacle paths are realizable by checking the cost of the minimum-cost flow. If the minimum-cost flow has cost exactly $n \cdot T - R$, the obstacle paths are realizable. If it costs more, they are not.

Symbol breakdown:

- n: the number of agents.
- T: the makespan (length of the longest path).
- R: the total number of obstacle move actions across all obstacle paths. For example, if obstacle o_1 makes 3 moves and obstacle o_2 makes 2 moves, then $R = 5$.
- $n \cdot T$: the total number of agent "action slots." Each of n agents has T timesteps, so there are $n \cdot T$ total agent actions.
- $n \cdot T - R$: the number of agent actions that are *not* realizing obstacle moves. These are "wasted" actions (agents moving without carrying obstacles, or waiting).

Why this formula works: Recall from the cost model (Section 4.2) that each agent action in the flow incurs a cost that depends on its relationship to obstacle movements:

- **Realizing** an obstacle move (agent traverses the edge in the same direction as the obstacle): cost **0**.
- **Reversing** an obstacle move (agent traverses the edge in the *opposite* direction of an obstacle move): cost **2**.
- **All other actions** (waiting, moving on an edge with no obstacle movement): cost **1**.

Each of the n agents performs exactly T actions (one per timestep), so there are $n \cdot T$ total agent actions. Let S_1 denote the number of realized obstacle move actions and S_2 denote the number of obstacle move actions that agents *reverse* in the plan. The total flow cost is:

$$\text{cost} = 0 \times S_1 + 2 \times S_2 + 1 \times (n \cdot T - S_1 - S_2) = n \cdot T - S_1 + S_2$$

A lower bound on the cost is achieved when $S_1 = R$ (all obstacle moves are realized) and $S_2 = 0$ (no agent reverses an obstacle move). In that case:

$$\text{cost} = n \cdot T - R + 0 = n \cdot T - R$$

Conversely, if the minimum-cost flow has cost exactly $n \cdot T - R$, it must be the case that all R obstacle moves are realized (since any unrealized move would increase the cost above this lower bound). Thus, cost $= n \cdot T - R$ if and only if all obstacle moves are realized.

Numerical Example

Suppose $n = 2$ agents, $T = 4$ timesteps, and the obstacle paths contain $R = 3$ total move actions.
Total action slots: $2 \times 4 = 8$. Target cost: $n \cdot T - R = 8 - 3 = 5$.
If all 3 obstacle moves are realized ($S_1 = 3$) and none reversed ($S_2 = 0$), the remaining $8 - 3 = 5$ agent actions cost 1 each. Total flow cost $= 8 - 3 + 0 = 5$.
If only 2 of the 3 obstacle moves are realized ($S_1 = 2$, $S_2 = 0$), the cost is $8 - 2 + 0 = 6 > 5$.
If all 3 are realized but one agent also reverses an obstacle move ($S_1 = 3$, $S_2 = 1$), the cost is $8 - 3 + 1 = 6 > 5$.
So minimum cost $= 5$ means all obstacle moves are realized with no reversals. Minimum cost > 5 means at least one obstacle move is not realized or some agent reverses an obstacle move.

5.4 Theorem 1: Optimality of NAT-CBS

Theorem 5.4 (Theorem 1 in the paper)**.** *For any MAWR instance P for which a solution exists, NAT-CBS returns a makespan-optimal solution for P.*

What it says in plain English: If the MAWR instance is solvable, NAT-CBS will find a solution, and that solution will have the smallest possible makespan.

Proof sketch (following the paper): The proof has three parts:

Part 1: NAT-CBS preserves CBS's optimality for obstacle paths.

Phase Ph1 is a CBS search over obstacle paths. CBS guarantees that it explores nodes in order of increasing cost (best-first search). Therefore, the first obstacle plan Π_{obs} found to be realizable has the minimum possible cost among all realizable obstacle plans.

Part 2: A solution exists implies a realizable obstacle plan exists.

If the MAWR instance has a solution, that solution contains both valid obstacle paths and valid agent paths. The obstacle paths form a realizable obstacles plan. Therefore, CBS in Phase Ph1

will eventually reach a node with a realizable obstacles plan (by the completeness property of CBS—it explores all possible constraint combinations).

Part 3: The first realizable node is optimal.

By Lemma 1, the cost of a MAWR solution is determined by its obstacle paths. Since NAT-CBS explores obstacle plans in non-decreasing order of cost (best-first search), and the first realizable plan it finds corresponds to a valid solution (by Lemmas 2 and 3), this solution has minimum cost.

Why the Two-Phase Approach Is Optimal

The optimality proof hinges on three facts:

1. The makespan depends only on obstacle paths (Lemma 1).
2. CBS explores obstacle plans in cost order.
3. The network flow correctly determines realizability (Lemmas 2 and 3).

Together, these guarantee that the *first* realizable obstacle plan CBS finds is optimal, and the corresponding agent plan does not increase the cost.

Completeness Caveat

NAT-CBS is complete only in the sense that CBS is complete: if a solution exists, it will be found—eventually. In practice, CBS (and by extension NAT-CBS) can be very slow because the constraint tree can grow exponentially. The paper's experiments confirm this: NAT-CBS is approximately four orders of magnitude slower than the non-optimal MAPF-DECOMP algorithm.

6 NAT-CBS vs. MAPF-DECOMP: Compared in Detail

6.1 MAPF-DECOMP (The Baseline)

MAPF-DECOMP (Li and Ma, 2023) is a decoupled approach:

1. Use a standard MAPF solver to compute collision-free obstacle paths.
2. Build a dependency graph of obstacle trajectories.
3. Partition each trajectory into segments that can be executed by individual agents.
4. Solve a MAPD-like (Multi-Agent Pickup and Delivery) problem: assign trajectory segments to agents and compute collision-free agent paths using a sub-optimal algorithm (EECBS).

Key limitation: Steps 1 and 4 are independent. The obstacle paths are planned without considering agent availability, and the agent assignment uses a sub-optimal solver. This means MAPF-DECOMP provides **no optimality guarantees**.

6.2 Solution Quality Comparison

The paper's experiments show:

- NAT-CBS always produces solutions with makespan $\leq$ MAPF-DECOMP's makespan (since NAT-CBS is optimal).
- On some instances, NAT-CBS's makespan is **less than half** of MAPF-DECOMP's.
- The solution cost ratio μ_{MKSP} (MAPF-DECOMP cost / NAT-CBS cost) typically increases as the number of tasks t increases. With more tasks, the coordination advantage of NAT-CBS's coupled approach becomes more pronounced.

6.3 A Concrete Comparison

The paper provides a side-by-side comparison (Figure 3) on a grid instance with 2 agents and 2 obstacles:

- NAT-CBS solution: makespan **8**. The blue obstacle undergoes non-atomic movement—agent a_1 moves it at timestep 1, then agent a_0 moves it at timesteps 3–4.
- MAPF-DECOMP solution: makespan **11**. In this solution, the red obstacle (which happens to start at the same location as its goal) must temporarily move out of the way to let another obstacle pass, then move back. The atomic task assumption forces suboptimal coordination.

Non-Atomicity Enables Shorter Plans

The key advantage of NAT-CBS's obstacle-centric approach is **non-atomic** task execution. When an obstacle can be carried by different agents at different times, the algorithm has more flexibility to avoid congestion and reduce the makespan. MAPF-DECOMP's atomic tasks (one agent per obstacle, start to finish) cannot exploit this flexibility.

6.4 Runtime Comparison

There is a significant trade-off:

- NAT-CBS is approximately 10,000× slower than MAPF-DECOMP.
- The success rate of NAT-CBS (fraction of instances solved within a 5-minute time limit) decreases as the number of tasks increases.
- The drop is especially sharp when the number of tasks t exceeds the number of agents n, because this forces the algorithm to introduce multiple realization conflicts (an obstacle plan that moves more obstacles than there are agents *must* have at least some realization conflicts).

Scalability Is the Main Limitation

While NAT-CBS guarantees optimal solutions, it is currently practical only for small instances (few agents and obstacles on small maps). The paper suggests that future work could use bounded-suboptimal variants (e.g., replacing the optimal obstacle path planner with EECBS) to trade off a small amount of optimality for dramatically improved scalability.

7 Key Definitions and Concepts Reference

This section collects the important definitions and concepts from the paper for easy reference.

Definition 7.1 (Atomic vs. Non-Atomic Tasks)**.** A task is **atomic** if it must be completed by a single agent from start to finish. A task is **non-atomic** if it can be carried out by multiple agents in sequence—one agent picks up the obstacle, carries it partway, sets it down, and another agent picks it up later.

NAT-CBS naturally supports non-atomic tasks because it plans obstacle paths independently of agent assignments. MAPF-DECOMP uses atomic tasks.

Definition 7.2 (Obstacle Conflict)**.** An **obstacle conflict** $\mathcal{C}_{\text{obs}}$ is a vertex or edge conflict between two obstacle paths. This is identical to a standard CBS conflict but occurs between

obstacles rather than agents.

Example: $\mathcal{C}_{\text{obs}} = \langle o_{\text{red}}, o_{\text{blue}}, c, 1 \rangle$ means obstacles o_{red} and o_{blue} both occupy vertex c at timestep 1.

Definition 7.3 (Realization Conflict)**.** A **realization conflict** $\mathcal{C}_{\text{real}} = \langle o_i, u, v, t \rangle$ represents an obstacle $o_i \in O$ moving along edge $(u, v) \in E$ from u to v at timestep t, without any agent being there to carry it.

This is the novel conflict type introduced by the paper. It arises when Ph2 (the network-flow phase) determines that an obstacle movement planned in Ph1 cannot be realized by any agent.

Definition 7.4 (Candidate Goal Node)**.** A **candidate goal node** is a CT node that contains a valid (collision-free) obstacles plan. It is a "candidate" because we still need to verify that the obstacle paths can be realized by agents.

Definition 7.5 (Goal Node)**.** A **goal node** is a CT node that contains a realizable and valid obstacles plan—i.e., a candidate goal node for which Ph2 successfully found a valid agents plan.

8 Summary of Notation

For quick reference, here is a table of all major symbols used in the paper:

Symbol	Type / Dimensions	Meaning
$G = (V, E)$	Graph	Undirected warehouse environment graph
V	Set of vertices	Locations in the warehouse
E	Set of edges	Corridors connecting adjacent locations
n	Scalar (integer)	Number of agents
m	Scalar (integer)	Number of movable obstacles
t	Scalar (integer)	Number of tasks (obstacles that change location)
$A = \{a_1, \ldots, a_n\}$	Set	Fleet of agents
$O = \{o_1, \ldots, o_m\}$	Set	Set of movable obstacles
$s : A \to V$	Function	Initial locations of agents (injective)
$\mathcal{L} : O \to V$	Function	A layout (injective map from obstacles to locations)
$\mathcal{L}_{\text{init}} : O \to V$	Function	Initial layout (positions of obstacles)
$\mathcal{L}_{\text{goal}} : O \to V$	Function	Goal layout (desired positions of obstacles)
$\pi = (v_0, \ldots, v_k)$	Sequence	A path (sequence of locations over time)
$\lvert\pi\rvert$	Scalar (integer)	Length of path π (number of timesteps = k)
π_{a_i}	Path	Path of agent a_i
π_{o_j}	Path	Path of obstacle o_j
Π_{obs}	Vector of paths	Obstacles plan $\langle \pi_{o_1}, \ldots, \pi_{o_m} \rangle$
Π_{agents}	Vector of paths	Agents plan $\langle \pi_{a_1}, \ldots, \pi_{a_n} \rangle$
$\Pi = \langle \Pi_{\text{obs}}, \Pi_{\text{agents}} \rangle$	Plan	Complete plan (obstacles + agents)
$\text{MKSP}(\Pi)$	Scalar (integer)	Makespan: $\max_{\pi \in \Pi} \lvert\pi\rvert$
T	Scalar (integer)	Makespan of current obstacles plan
P	Tuple	MAWR instance $\langle G, A, O, s, \mathcal{L}_{\text{init}}, \mathcal{L}_{\text{goal}} \rangle$
	Conflict-Based Search	
$\mathcal{N}$	CT node	A node in the Constraint Tree
$\mathcal{R}$	CT node	Root node of the CT
OPEN	Priority queue	Queue of CT nodes ordered by cost
$\mathcal{C}_{\text{obs}}$	Conflict	Obstacle conflict (vertex or edge)
$\mathcal{C}_{\text{real}}$	Conflict	Realization conflict $\langle o_i, u, v, t \rangle$
$\langle +, a, v, t \rangle$	Constraint	Positive vertex constraint
$\langle -, a, v, t \rangle$	Constraint	Negative vertex constraint
$\langle +, a, u, v, t \rangle$	Constraint	Positive edge constraint
$\langle -, a, u, v, t \rangle$	Constraint	Negative edge constraint
	Time-Expanded Graph and Network Flow	
$G^T = (\mathcal{V}^T, \mathcal{E}^T)$	Directed graph	T-step time-expanded graph
w	Vertex	Source in G^T
w'	Vertex	Sink in G^T
v_t^{out}	Vertex	"Out" copy of vertex v at time t
v_t^{in}	Vertex	"In" copy of vertex v at time t
$e_t^{\text{in}}, e_t^{\text{out}}$	Vertices	Edge vertices for edge e at time t
R	Scalar (integer)	Total number of obstacle move actions
S_1	Scalar (integer)	Number of realized obstacle move actions
S_2	Scalar (integer)	Number of reversed obstacle move actions
$n \cdot T - S_1 + S_2$	Scalar	Flow cost formula (general)
$n \cdot T - R$	Scalar	Flow cost lower bound (when $S_1 = R$, $S_2 = 0$)

9 Putting It All Together: A Complete Worked Example

Let us trace through a complete (small) MAWR instance from start to finish, applying every concept introduced in this companion.

9.1 The Instance

Consider a 2×2 grid with a side extension:

```
v1 -- v2
|      |
v3 -- v4 -- v5
```

$V = \{v_1, v_2, v_3, v_4, v_5\}$, $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4), (v_4, v_5)\}$.

Agents: $A = \{a_1, a_2\}$ with $s(a_1) = v_5$, $s(a_2) = v_3$.

Obstacles: $O = \{o_A\}$ with $\mathcal{L}_{\text{init}}(o_A) = v_1$, $\mathcal{L}_{\text{goal}}(o_A) = v_5$.

So we have 2 agents and 1 obstacle. The obstacle must move from v_1 to v_5.

9.2 Step 1: Initialize (Ph1)

Compute the shortest path for o_A from v_1 to v_5:

$$\pi_{o_A} = (v_1, v_2, v_4, v_5), \quad |\pi_{o_A}| = 3$$

(Path: $v_1 \to v_2 \to v_4 \to v_5$, length 3.)

Root CT node $\mathcal{R}$: constraints $= \emptyset$, $\Pi_{\text{obs}} = \langle (v_1, v_2, v_4, v_5) \rangle$, cost $= 3$.

There is only one obstacle, so there are no obstacle conflicts. We proceed directly to Ph2.

9.3 Step 2: Ph2 — Realize the Obstacle Path

The obstacle makes 3 moves:

- $t = 0$: $v_1 \to v_2$ (move action 1)
- $t = 1$: $v_2 \to v_4$ (move action 2)
- $t = 2$: $v_4 \to v_5$ (move action 3)

So $R = 3$ (three obstacle move actions).

We build the 3-step time-expanded graph G^3 and formulate MCMF with:

- Supply of $n = 2$ at source w, demand of 2 at sink w'.
- Source edges: $(w, v_5^{\text{out},0})$ and $(w, v_3^{\text{out},0})$ (agents start at v_5 and v_3).
- Wait edges and in-to-out vertex edges have cost 1; edge-traversal edges have cost 0 (with obstacle move edges also at cost 0).
- No lower bounds yet (no positive edge constraints in the root node).

Expected cost if all moves realized: $n \cdot T - R = 2 \times 3 - 3 = 3$.

The MCMF solver finds the minimum-cost flow. Suppose the minimum cost is 3. Then all 3 obstacle moves are realized, and we have found agent paths.

Can any agent realize the obstacle's first move? Agent a_2 starts at v_3. To carry o_A at $t = 0$ (move $v_1 \to v_2$), a_2 would need to be at v_1 at $t = 0$. But a_2 starts at v_3, not v_1. So a_2 cannot realize this move at $t = 0$.

Agent a_1 starts at v_5. To be at v_1 at $t = 0$, a_1 would need to already be at v_1—but a_1 starts at v_5. So neither agent can realize the $v_1 \to v_2$ move at $t = 0$.

This means we have a **realization conflict**: $\mathcal{C}_{\text{real}} = \langle o_A, v_1, v_2, 0 \rangle$.

9.4 Step 3: Handle the Realization Conflict

The CT splits $\mathcal{R}$ into two children:

Child $\mathcal{N}_1$ (positive constraint): Force an agent to traverse $v_1 \to v_2$ at $t = 0$. But no agent starts at v_1, so this is infeasible (R1 fails). Discard.

Child $\mathcal{N}_2$ (negative constraint): Forbid o_A from moving $v_1 \to v_2$ at $t = 0$. This forces o_A to wait at v_1 at $t = 0$, then find another path. Replanning:

$$\pi_{o_A} = (v_1, v_1, v_2, v_4, v_5), \quad |\pi_{o_A}| = 4$$

Cost = 4.

Now obstacle moves are:

- $t = 0$: wait at v_1
- $t = 1$: $v_1 \to v_2$ (move action 1)
- $t = 2$: $v_2 \to v_4$ (move action 2)
- $t = 3$: $v_4 \to v_5$ (move action 3)

$R = 3$, $T = 4$. Expected cost: $2 \times 4 - 3 = 5$.

Now Ph2 is invoked again. Agent a_2 starts at v_3. Can it reach v_1 by $t = 1$? Yes: $v_3 \to v_1$ is one step. Then a_2 can be at v_1 at $t = 1$ and carry o_A from $v_1 \to v_2$.

Let us attempt to construct an agent plan (first attempt—may contain conflicts):

Time	a_2 (starts v_3)	a_1 (starts v_5)
0	v_3	v_5
1	v_1 (arrives at o_A)	v_4
2	v_2 (carries o_A: $v_1 \to v_2$... wait, o_A moves $v_1 \to v_2$ at $t = 1$)	v_2

The obstacle moves at $t = 1$: $v_1 \to v_2$. Agent a_2 is at v_1 at $t = 1$ and moves to v_2 at $t = 2$... but the obstacle moves at $t = 1$ (from the obstacle path, the obstacle is at v_1 at $t = 1$ and at v_2 at $t = 2$, i.e., the move is during timestep 1→2). So a_2 must be at v_1 at $t = 1$ and at v_2 at $t = 2$—this matches!

Continuing:

Time	o_A	a_2 (starts v_3)	a_1 (starts v_5)
0	v_1	v_3	v_5
1	v_1 (wait)	v_1	v_4
2	v_2	v_2 (carries o_A)	v_4 (wait)
3	v_4	v_4 ...conflict with a_1!	v_4

Vertex conflict at v_4, $t = 3$. So a_1 must move away. Revised:

Time	o_A	a_2 (starts v_3)	a_1 (starts v_5)
0	v_1	v_3	v_5
1	v_1	v_1	v_4
2	v_2	v_2 (carries o_A)	v_2... conflict!

There are coordination issues. The MCMF solver handles all of this automatically—it finds collision-free agent paths by construction (capacity-1 edges prevent collisions). The feasible flow might route a_1 through $v_5 \to v_4 \to v_3 \to v_3 \to v_3$ (wait at v_3) while a_2 goes $v_3 \to v_1 \to v_2 \to v_4 \to v_5$ (carrying o_A the entire way).

In this case: a_2 single-handedly carries o_A from v_1 to v_5, while a_1 moves out of the way. The makespan is 4.

The MCMF returns cost $= 2 \times 4 - 3 = 5$ (all 3 obstacle moves realized). No realization conflicts. NAT-CBS returns this as the solution.

The Example Illustrates All Key Concepts

This small example demonstrates:

1. Initial obstacle paths may not be realizable (no agent at v_1 at $t = 0$).
2. Realization conflicts force the obstacle to adjust its path (wait one step).
3. The adjusted path has higher makespan (4 vs. 3) but is realizable.
4. The MCMF solver automatically handles agent collision avoidance.
5. NAT-CBS guarantees this makespan-4 solution is optimal.