Dynamic Multi-Agent Pathfinding for Sorting and Delivery

Aneel Badesha, Anton Ilic, Duncan Kjellbotn
CMPT 417 Intelligent Systems

Abstract. Dynamic multi-agent pathfinding (MAPF) arises in automated sorting and delivery systems where multiple robots repeatedly transport items from a shared pickup location to multiple destination bins. We study a dynamic MAPF variant in which agents receive new delivery goals online and must replan in real time while avoiding collisions, similar to the reverse Amazon-warehouse scenario with robots dispersing from a central station. We explore techniques to reduce replanning time and maintain solution quality, including an implementation of large neighborhood search and optimizations to the A* algorithm.

1 Introduction

Automated sorting and delivery is increasingly more important within modern factory environments. In warehouses, this is done by fleets of autonomous robots which move items between pickup stations to multiple storage or pickup locations. As the number of robots grows, the need for collision free real time paths gives rise to the dynamic multi agent pathfinding (MAPF) problem.

In this project, we aim to optimize planning time, rather than guaranteeing strictly optimal solutions. We use Conflict-Based Search (CBS) as a high level search algorithm, but replace its low level search algorithm with variants of the A star algorithm, to obtain faster but optimal enough paths in real time.

We use Large Neighborhood Search (LNS) as an iterative repair mechanism that handles conflicts by selecting a subset of agents (neighborhood) involved in collisions and replanning their paths while treating other agents' paths as dynamic obstacles. The neighborhood is constructed by selecting agents involved in collisions and expanding to include agents they conflict with, up to a predefined size limit. This approach allows us to resolve conflicts locally without replanning all agents from scratch, significantly reducing computational overhead while still achieving collision-free solutions. By only replanning future movements, LNS adapts to sorting and delivery scenarios where multiple agents must coordinate their movements in real-time.

2 Implementation

2.1 Conflict Based Search [1]

Conflict Based Search (CBS) is a two-level algorithm that searches over a conflict tree. Each node in the CT represents a set of constraints and corresponding paths for all agents. At the low level, CBS uses A* to find optimal paths for individual agents. We run our experiments with a disjoint splitting version of CBS, as it will always have better performance over standard splitting.

2.1.1 Disjoint Splitting

Disjoint splitting resolves a conflict by creating two child nodes that add constraints only to a chosen primary agent and update its conflict set.One constraint is positive forcing that agent to be in that location at that timestep and the other is negative and enforces the opposite. This is much faster than standard splitting as the positive constraints are much stricter and quickly eliminate possible solutions. Our project uses exclusively disjoint splitting to accelerate the conflict resolution.

2.1.2 Windowed Conflict Based Search

We edited our CBS to have a dynamic horizon defined by the earliest finishing active agent. At each CT node, collisions beyond this timestamp are ignored because replanning occurs before they happen. This restricts the search to near-term conflicts and defers later ones.

```
function CREATE-NODE(map, starts, final_goals, heuristics,
                     constraints, parent_paths, non_goal, updated_agents)
    ...
    collision_list ← DETECT-COLLISIONS(new_node.paths)

    shortest_path ← FIND-SHORTEST-NON-FINAL-PATH(new_node.paths, final_goals)

    if shortest_path ≠ NIL then
        agent    ← shortest_path.agent
        end_time ← shortest_path.end_time
        new_node.shortest_non_path ← agent

        i ← 0
        while i < LENGTH(collision_list) do
            col ← collision_list[i]
            if col.timestep > end_time + 1 then
                REMOVE-AT(collision_list, i)
            else
                i ← i + 1
            end if
        end while
    end if

    new_node.collisions ← collision_list
```

[1] Implementation for the CBS and splitting can be found in the appendix.

```
    return new_node
```

2.2 A* algorithm improvements

Since CBS repeatedly calls the low-level A* planner whenever new constraints are added, even modest improvements or controlled suboptimality in A* can significantly reduce overall runtime, especially in our dynamic setting where agents replan frequently.

2.2.1 Weighted A*

Weighted A* places a weight on the heuristic, thus, the formula is f(n) = g(n) + w * h(n) where w >= 1.

```
function W-ASTAR(map, start, goal, h, agent, constraints, w):
    constraint_table ← BUILD-CONSTRAINT-TABLE(constraints, agent)
    OPEN   ← priority queue ordered by f(n) = g(n) + w·h(n)
    CLOSED ← ∅; root ← (loc = start, g = 0, h = h(start), t = 0, parent = NIL)
    INSERT(OPEN, root)
    CLOSED[(start, 0)] ← root
    while OPEN ≠ ∅ do
        current ← POP-MIN(OPEN)
        if current.loc = goal and NO-FUTURE-CONSTRAINTS(current, constraint_table) then
            return EXTRACT-PATH(current)

        for each a ∈ {UP, RIGHT, DOWN, LEFT, WAIT} do
            child_loc ← MOVE(current.loc, a)
            t' ← current.t + 1
            if BLOCKED(child_loc, map) then continue
            if CONSTRAINED(current.loc, child_loc, t', constraint_table, agent) then
continue

            child ← (loc = child_loc,
                     g   = current.g + 1,
                     h   = h(child_loc),
                     t   = t',
                     parent = current)
            state ← (child_loc, t')

            if state ∉ CLOSED or f(child) < f(CLOSED[state]) then
                CLOSED[state] ← child
                INSERT(OPEN, child)

    return FAILURE
```

2.2.2 Focal Search

Focal search keeps both an OPEN list sorted by f(n) and also a focal list, containing nodes with f(n) <= w * f_min, where f_min is the min f-value in OPEN, and expands nodes from this list according to a heuristic.

```
function FocalSearch(start, goal, epsilon)
    OPEN = priority_queue()
    CLOSED = set()

    g(start) = 0
    h(start) = heuristic(start, goal)
    f(start) = g(start) + h(start)

    OPEN.push(start)

    while OPEN is not empty
        f_min = OPEN.get_min_f_value()

        FOCAL = { node in OPEN | f(node) <= (1 + epsilon) * f_min }

        current = node in FOCAL with minimum secondary_heuristic(node)

        if current == goal
            return reconstruct_path(current)

        OPEN.remove(current)
        CLOSED.add(current)

        for neighbor in get_neighbors(current)
            if neighbor in CLOSED
                continue

            tentative_g = g(current) + cost(current, neighbor)

            if neighbor not in OPEN
                g(neighbor) = tentative_g
                h(neighbor) = heuristic(neighbor, goal)
                f(neighbor) = g(neighbor) + h(neighbor)
                parent(neighbor) = current
                OPEN.push(neighbor)
            elif tentative_g < g(neighbor)
                g(neighbor) = tentative_g
                f(neighbor) = g(neighbor) + h(neighbor)
                parent(neighbor) = current
                OPEN.update(neighbor)

    return failure
```

To optimize the algorithm for MAPF, we defined states as (location, time) tuples rather than simple coordinates. However, we employed an explicit list management strategy where nodes were updated or removed directly from the OPEN list. While conceptually simple, this approach incurred significant overhead compared to 'lazy removal', likely contributing to the performance degradation observed in high-agent scenarios. We added a depth limit to prevent infinite search in unsolvable scenarios and selected the distance to the goal as a secondary heuristic.

2.3 Windowed Large Neighborhood Search[2]

Large Neighborhood Search (LNS) uses a complete solution, selects a subset of agents, deletes part of their paths, and then replans for that subset. In our implementation, we first detect collisions in the current set of paths. From these collisions, we construct a set of chosen agents involved in conflicts and use this for our neighborhood selection, while treating other agents as moving obstacles. For each chosen agent, we replan the path from the earliest collision timestep onward using our low-level search, under vertex and edge constraints given by previously replanned neighbors and the moving obstacles. We then apply atomicity: if all neighborhood agents have valid paths, we commit the changes; otherwise, we discard them.

We extended traditional LNS with a temporal window that limits collision resolution to early timesteps, allowing agents with longer paths to focus on near-term conflicts while deferring distant collisions. This is particularly useful for dynamic MAPF where goals change over time. The window is defined by the shortest non-final path (agents marked as not reaching their goals): *window_limit = length_of_shortest_pending_path + 1*

Only collisions occurring before this timestep are resolved. This prevents over-planning for agents that will have their goals updated soon anyway. This window implements several significant decisions differing from standard LNS. First, it acts as a strict filter: both static and neighborhood constraints are bound by the limit, and collision resolution is restricted exclusively to conflicts occurring before this horizon. Second, we perform dynamic re-windowing, recalculating the limit at every iteration based on updated pending agents. This ensures the algorithm focuses entirely on near-term conflicts, preventing wasted computation on long-term plans that will inevitably be invalidated when agents receive new goals.

3 Methodology

We aim to reduce replanning time while maintaining solution quality. To this end, we systematically compare several CBS-based planners and low-level search variants, including baseline CBS with disjoint splitting, a windowed CBS variant, and low-level weighted and focal A* search. We also experiment with Large Neighborhood Search to further improve performance.

4 Experiment Setup

We implemented algorithms using Python 3.13.7, and ran our instances on a PC running Windows 10 with an Intel i7-1165G7 at 2.80GHz CPU, 16 Gb RAM at 4200 MHz, and an Intel Iris(R) Ze GPU. We used the code provided by Dr. Hang Ma for the MAPF individual project as a baseline. To validate our solution, we refactored the individual project's output module from a simple path visualizer into a continuous simulation environment. This update allows the system

[2] Implementation for Windowed LNS can be found in the appendix.

to track agent positions in real-time and automatically recalculate paths once an agent arrives at its destination

4.1 Experiment Instances

We evaluate on five sorting instances, with varying numbers of agents, goals, and room layouts. Each instance is specified in the format as specified below.

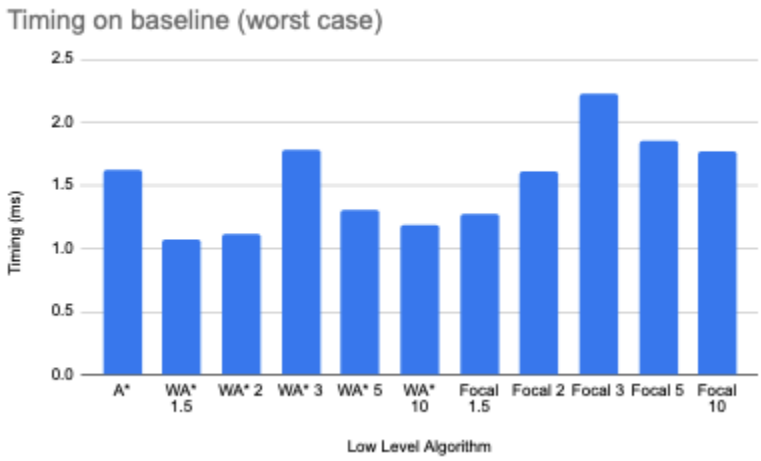
```
s                   # sorting instance tag
rows cols           # map dimensions
map                 # '@' = obstacle, '.' = free space (one line per row)
num_agents          # number of agents
sx sy               # start locations (one line per agent)
...
num_goals           # number of goals
gx gy               # goal / dropoff locations (one line per goal)
...
px py               # pickup location
d1 ... dN           # delivery sequence (indices into the goals)
```

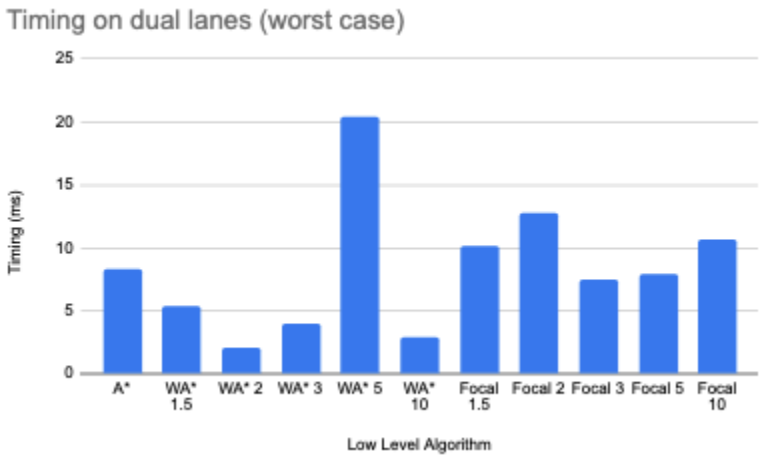
Instance	File name	Agents	Map Size	Description
Baseline	simpleSortInstance.txt	2	7x7	Simple baseline instance
Dual Lanes	dual_lanes.txt	4	7x11	Dual lane instance, with + shaped junction
8 agent warehouse	8_agent_sort.txt	8	19x21	Warehouse with obstacles.
12 agent warehouse	12_agent_sort.txt	12	19x21	
16 agent open room	large_16agents.txt	16	20x20	

These instances were chosen to cover a spectrum of difficulty and interaction patterns, from a simple low-density baseline to tightly coupled, high-density scenarios. The cases with a larger number of agents highlight our performance improvements while the smaller cases (Baseline and Dual Lanes) test basic correctness.

5 Experiment Results

Our experiments proved that Weighted A* with weight = 1.5 offered the most consistent performance improvements, reducing worst case runtime in almost all cases over regular A*. In contrast, our implementation of Focal Search proved computationally expensive; rather than reducing search time, the overhead of managing the secondary heuristic caused it to perform worse than the baseline, eventually failing to find solutions within the time limit for the 16-agent test case at really high weights, such as weight == 5. This unexpected latency likely stems from the overhead by linearly scanning the OPEN list to build the FOCAL set during each expansion. Using two heaps, or doing lazy removal could potentially fix this issue, moving scan time from linear to logarithmic.






Timing on 8-agent instance (worst case)
100
75
50
25
0
Timing (ms)
A*
WA* 1.5
WA* 2
WA* 3
WA* 5
WA* 10
Focal 1.5
Focal 2
Focal 3
Focal 5
Focal 10
Low Level Algorithm


Timing on 12 agent instance (worst case)
1250
1000
750
500
250
0
Timing (ms)
A*
WA* 1.5
WA* 2
WA* 3
WA* 5
WA* 10
Focal 1.5
Focal 2
Focal 3
Focal 5
Focal 10
Low Level Algorithm


Timing on 16 agent instance (worst case)
2500
2000
1500
1000
500
0
Timing (ms)
A*
WA* 1.5
WA* 2
WA* 3
WA* 5
WA* 10
Focal 1.5
Focal 2
Focal 3
Focal 5
Focal 10
Low Level Algorithm

6 Conclusions

In this project, we investigated dynamic multi agent pathfinding problem, where agents receive goals and must replan in real time from a shared pickup location to multiple destinations. We found that modifying the low level planner to a nonoptimal but complete A* alternative did provide speedup benefits, with WA* with a weight of 1.5 significantly outperforming baseline A*.

7 Appendix

Conflict Based Search Implementation

```
function FIND-SHORTEST-NON-FINAL-PATH(paths, final_goals):
    best_agent ← NIL
    best_len   ← ∞
    for each agent, path in paths do
        if final_goals[agent] = FALSE then
            if |path| < best_len then
                best_len   ← |path|
                best_agent ← agent
    if best_agent = NIL then
        return NIL
    end_time ← best_len − 1
    return (best_agent, end_time)


function STANDARD-SPLITTING(collision):
    if |collision.loc| = 1 then
        return [
            (agent = collision.a1, loc = collision.loc, timestep = collision.timestep),
            (agent = collision.a2, loc = collision.loc, timestep = collision.timestep)
        ]
    else
        reverse_loc ← REVERSE(collision.loc)
        return [
            (agent = collision.a1, loc = reverse_loc,        timestep = collision.timestep),
            (agent = collision.a2, loc = collision.loc,      timestep = collision.timestep)
        ]


function DISJOINT-SPLITTING(collision):
    r ← RANDOM-CHOICE({collision.a1, collision.a2})
    if |collision.loc| = 1 then
        return [
            (agent = r, loc = collision.loc, timestep = collision.timestep),
            (agent = r, loc = collision.loc, timestep = collision.timestep, positive = TRUE)
        ]
    else
```

```
        loc_r ← collision.loc
        if r = collision.a1 then
            loc_r ← REVERSE(collision.loc)
        return [
            (agent = r, loc = loc_r, timestep = collision.timestep),
            (agent = r, loc = loc_r, timestep = collision.timestep, positive = TRUE)
        ]


function GET-GOAL(agent, node, global_goals):
    if agent ∈ node.non_goal then
        return node.non_goal[agent]
    else
        return global_goals[agent]


function COUNT-OVERLAPPING-GOALS(node, global_goals, num_agents):
    goals_found ← ∅
    overlapping ← 0
    for agent from 0 to num_agents − 1 do
        g ← GET-GOAL(agent, node, global_goals)
        if g ∈ goals_found then
            overlapping ← overlapping + 1
        else
            goals_found ← goals_found ∪ {g}
    return overlapping


function FIND-SHARED-GOAL-PAIR(node, global_goals, num_agents):
    owner ← empty map
    for agent from 0 to num_agents − 1 do
        g ← GET-GOAL(agent, node, global_goals)
        if g ∈ owner then
            return (owner[g], agent)
        else
            owner[g] ← agent
    return NIL


function CREATE-NODE(map, starts, goals, final_goals, heuristics,
                     constraints, parent_paths, non_goal, updated_agents, num_agents):
    new_node ← {
        cost              = 0,
        constraints       = COPY(constraints),
        paths             = COPY(parent_paths),
        collisions        = ∅,
        non_goal          = COPY(non_goal),
        shortest_non_path = NIL
    }

    skipped ← FALSE
    for each a ∈ updated_agents do
```

```
        path ← A-STAR(map,
                      starts[a],
                      GET-GOAL(a, new_node, goals),
                      heuristics[a],
                      a,
                      new_node.constraints)
        if path = FAILURE then
            skipped ← TRUE
        else
            new_node.paths[a] ← path
    if skipped = TRUE then
        return NIL

    c1 ← GET-SUM-OF-COST(new_node.paths)
    c2 ← SHARED_COLLISION_MULT · COUNT-OVERLAPPING-GOALS(new_node, goals, num_agents)
    new_node.cost ← c1 + c2

    collision_list ← DETECT-COLLISIONS(new_node.paths)

    shortest_path ← FIND-SHORTEST-NON-FINAL-PATH(new_node.paths, final_goals)
    if shortest_path ≠ NIL then
        agent    ← shortest_path.agent
        end_time ← shortest_path.end_time
        new_node.shortest_non_path ← agent

        i ← 0
        while i < |collision_list| do
            col ← collision_list[i]
            if col.timestep > end_time + 1 then
                REMOVE-AT(collision_list, i)
            else
                i ← i + 1

    new_node.collisions ← collision_list
    return new_node


function PUSH-NODE(OPEN, node):
    key ← (|node.collisions|, node.cost)
    HEAP-PUSH(OPEN, (key, node))


function POP-NODE(OPEN):
    (key, node) ← HEAP-POP(OPEN)
    return node


function GENERATE-NODES(collision, parent_node, disjoint,
                        map, starts, goals, final_goals, heuristics,
                        num_agents, OPEN):
    if disjoint = TRUE then
        split ← DISJOINT-SPLITTING(collision)
```

```
    else
        split ← STANDARD-SPLITTING(collision)

    for each constr in split do
        new_constraints ← COPY(parent_node.constraints)
        APPEND(new_constraints, constr)

        updated_agents ← {constr.agent}
        if "positive" ∈ constr and constr.positive = TRUE then
            updated_agents ← updated_agents ∪ PATHS-VIOLATE-CONSTRAINT(constr,
parent_node.paths)

        new_node ← CREATE-NODE(map, starts, goals, final_goals, heuristics,
                               new_constraints,
                               parent_node.paths,
                               parent_node.non_goal,
                               updated_agents,
                               num_agents)
        if new_node ≠ NIL then
            PUSH-NODE(OPEN, new_node)


function HANDLE-SHARED-GOAL-COLLISION(agents, node,
                                      map, starts, goals, final_goals, heuristics,
                                      num_agents, OPEN):
    a1 ← agents[0]
    a2 ← agents[1]
    loc ← GET-GOAL(a1, node, goals)

    t1 ← GET-TIMESTEP-FOR-LOCATION(node.paths[a1], loc)
    t2 ← GET-TIMESTEP-FOR-LOCATION(node.paths[a2], loc)
    if t1 > t2 then
        delayed  ← a1
        priority ← a2
    else
        delayed  ← a2
        priority ← a1

    for dir ∈ {0, 1, 2, 3} do
        new_non_goal ← COPY(node.non_goal)
        new_loc ← MOVE-GOAL(GET-GOAL(delayed, node, goals), dir)
        if IS-BLOCKED(new_loc, map) then
            continue
        new_non_goal[delayed] ← new_loc

        new_node ← CREATE-NODE(map, starts, goals, final_goals, heuristics,
                               node.constraints,
                               node.paths,
                               new_non_goal,
                               [delayed],
                               num_agents)
        if new_node ≠ NIL then
```

```
            PUSH-NODE(OPEN, new_node)


function CBS-RESOLVE-COLLISIONS(map, starts, goals, final_goals, heuristics,
                                paths, num_agents, timestep, disjoint = TRUE):
    OPEN ← ∅
    old_starts ← COPY(starts)

    relevant_paths ← empty list
    new_starts ← empty list
    for i from 0 to num_agents − 1 do
        (start_i, suffix_i) ← SPLIT-PATH(paths[i], timestep)
        new_starts[i] ← start_i
        relevant_paths[i] ← suffix_i

    starts ← new_starts

    root ← CREATE-NODE(map, starts, goals, final_goals, heuristics,
                       [], relevant_paths, {}, [], num_agents)
    PUSH-NODE(OPEN, root)

    best_node ← NIL
    while OPEN ≠ ∅ do
        node ← POP-NODE(OPEN)

        if node.collisions = ∅ then
            best_node ← node
            break

        shared_agents ← FIND-SHARED-GOAL-PAIR(node, goals, num_agents)
        if shared_agents ≠ NIL then
            HANDLE-SHARED-GOAL-COLLISION(shared_agents, node,
                                         map, starts, goals, final_goals, heuristics,
                                         num_agents, OPEN)
            continue

        collision ← node.collisions[0]
        GENERATE-NODES(collision, node, disjoint,
                       map, starts, goals, final_goals, heuristics,
                       num_agents, OPEN)

    for i from 0 to num_agents − 1 do
        paths[i] ← EXTEND-PATH(paths[i], timestep, best_node.paths[i])

    pending_agents ← KEYS(best_node.non_goal)
    starts ← old_starts
    return (paths, pending_agents)
```

Windowed Large Neighborhood Search

```
function FIND-NEIGHBOURHOOD(collisions, num_agents, k):
    if num_agents ≤ k then
        return [0, 1, ..., num_agents − 1]
    A ← ∅
    for each col in collisions do
        A ← A ∪ {col.a1, col.a2}
    neighbourhood ← [RANDOM-CHOICE(A)]
    checked ← ∅
    while |neighbourhood| < k do
        path_agent ← any agent ∈ neighbourhood \ checked
        if path_agent = NIL then
            remaining ← {0, ..., num_agents − 1} \ neighbourhood
            path_agent ← RANDOM-CHOICE(remaining)
            neighbourhood.append(path_agent)
        for each col in collisions do
            if |neighbourhood| ≥ k then break
            if col.a1 = path_agent and col.a2 ∉ neighbourhood then
                neighbourhood.append(col.a2)
            else if col.a2 = path_agent and col.a1 ∉ neighbourhood then
                neighbourhood.append(col.a1)
        checked ← checked ∪ {path_agent}
    SHUFFLE(neighbourhood)
    return neighbourhood
end function

function PATH-CONSTRAINTS(path, agent_id):
    C ← ∅
    for t from 0 to |path| − 1 do
        v ← path[t]
        C ← C ∪ { (agent = agent_id, loc = [v], t = t, goal = (t = |path| − 1)) }
        if t > 0 then
            u ← path[t − 1]
            C ← C ∪ { (agent = agent_id, loc = [v, u], t = t) }
    return C
end function

function REPLAN-NEIGHBOURHOOD(map, paths, goals, heuristics,
                              neighbourhood, timestep, nonarriveDist):
    relevantPaths ← empty map
    new_starts ← empty map
    for each a in neighbourhood do
        suffix ← SPLIT-PATH(paths[a], timestep)
        relevantPaths[a] ← suffix
        new_starts[a] ← suffix[0]
    remaining ← {0, ..., num_agents − 1} \ neighbourhood
    static_paths ← [SPLIT-PATH(paths[b], timestep) for b in remaining]
```

```
    // Calculate window limit for temporal constraints
    window_timestep ← FIND-SHORTEST-NON-FINAL-PATH()
    window_limit ← NIL
    if window_timestep ≠ NIL then
        window_limit ← window_timestep + 1 − timestep  // Adjust for split offset

    old_paths ← copy of paths restricted to neighbourhood
    planned ← ∅
    success ← TRUE

    for each a in neighbourhood do
        constraints ← ∅

        // Collect constraints from static (non-neighbourhood) agents
        for each sp in static_paths do
            C ← PATH-CONSTRAINTS(sp, a)
            if window_limit ≠ NIL then
                C ← {c ∈ C : c.timestep < window_limit}  // Filter by window
            constraints ← constraints ∪ C

        // Collect constraints from previously replanned neighbourhood agents
        for each b in planned do
            C ← PATH-CONSTRAINTS(relevantPaths[b], a)
            if window_limit ≠ NIL then
                C ← {c ∈ C : c.timestep < window_limit}  // Filter by window
            constraints ← constraints ∪ C

        if IS-MARKED-FOR-UPDATES(a) then
            new_path ← SEARCH-CLOSEST(map, new_starts[a], goals[a],
                                      heuristics[a], a, constraints, nonarriveDist)
        else
            new_path ← LOW-LEVEL-PLAN(map, new_starts[a], goals[a],
                                      heuristics[a], a, constraints)
        if new_path = FAILURE then
            success ← FALSE
            break
        relevantPaths[a] ← new_path
        planned ← planned ∪ {a}

    if success then
        for each a in neighbourhood do
            paths[a] ← EXTEND-PATH(paths[a], timestep, relevantPaths[a])
    else
        for each a in neighbourhood do
            paths[a] ← old_paths[a]
    return success
end function

function WINDOWED-LNS-RESOLVE-COLLISIONS(map, paths, goals, heuristics,
                                         neighbourhood_size, timestep, nonarriveDist):
    HANDLE-SHARED-GOALS(paths, goals)
    collisions ← DETECT-COLLISIONS(paths)
```

```
    // Apply windowing: filter collisions beyond shortest non-final path
    window_timestep ← FIND-SHORTEST-NON-FINAL-PATH()
    if window_timestep ≠ NIL then
        window_limit ← window_timestep + 1
        collisions ← {col ∈ collisions : col.timestep < window_limit}

    while collisions ≠ ∅ do
        N ← FIND-NEIGHBOURHOOD(collisions, num_agents, neighbourhood_size)
        REPLAN-NEIGHBOURHOOD(map, paths, goals, heuristics,
                             N, timestep, nonarriveDist)
        collisions ← DETECT-COLLISIONS(paths)

        // Re-apply windowing after replanning
        window_timestep ← FIND-SHORTEST-NON-FINAL-PATH()
        if window_timestep ≠ NIL then
            window_limit ← window_timestep + 1
            collisions ← {col ∈ collisions : col.timestep < window_limit}

    return paths
end function

function FIND-SHORTEST-NON-FINAL-PATH():
    """Returns length of shortest path among pending (non-goal) agents"""
    if pending_agents = ∅ then
        return NIL
    return min{|paths[a]| : a ∈ pending_agents}
end function
```

References

1. LaFayette Engineering. (n.d.). *Transforming warehouse dynamics: The 5 powers of automated sortation systems*. https://www.lafayette-engineering.com/transforming-warehouse-dynamics-the-5-powers-of-automated-sortation-systems/