# Software Systems Capstone Final Report

## Multi-Agent Path Finding (MAPF)

Tommy Oh & Sam Showkati

koa18@sfu.ca & ssa574@sfu.ca

CMPT 494, CMPT 495: Software Systems Program Capstone I & II

Supervised by Dr. Keval Vora

Table of Contents

# 1. Introduction

In modern automated systems such as warehouse logistics, autonomous vehicles and air traffic control, the efficient movement of single entities is no longer sufficient. The increasing density and complexity of operational environments necessitate sophisticated techniques for planning the simultaneous movement of numerous independent agents. The critical challenge forms the core of the Multi-Agent Path Finding (MAPF) problem.

## 1.1 Explaining the problem

The MAPF problem is formally defined as finding a set of collision-free paths for a collection of agents, where each agent must travel from its designated starting position to a unique goal position within a shared environment, such as a graph or a grid. In this project, we are using a shared graph that can be directed or undirected, and weighted or unweighted. We also have a set of constraints. A solution requires not only specifying the sequence of movements for each agent from source to sink, but also ensuring that constraints are met. The primary objective in this project is to find a collision-free path while optimizing metrics such as minimizing the total travel time for all agents or the sum of moves. While finding a path for a single agent is a well-studied problem, the introduction of multiple interacting agents increases the search space exponentially with the number of agents. This defines that solving MAPF optimally is NP-hard in general. The difficulty in MAPF lies in the agent's movement coordination: an agent may have to wait at the vertex or take a suboptimal detour to avoid colliding with another agent, making individual path planning insufficient.

## 1.2 What is Solution?

To tackle this complexity, MAPF solutions primarily focus on techniques that guarantee global optimality while mitigating the combinatorial explosion. Among the most successful methods, we have chosen Conflict-Based Search (CBS) and Increasing Cost Tree Search (ICTS). CBS operates on a robust, two-level framework: A low-level search finds the optimal path for each agent individually, ignoring conflicts, while a high-level search tree explores and resolves the conflicts. When a collision occurs, the constraint tree branches, adding constraints that force the agents to take different routes or wait. Similarly, ICTS utilizes a systematic framework, but primarily by searching over the total cost of the solution. It begins by finding the cheapest possible individual paths. Then, in an increasing sequence of total costs, ICTS searches for a combination of individual paths that satisfy all the non-collision constraints within that cost budget. These optimal approaches form the foundation for state-of-the-art MAPF solving.

## 1.3 Capstone Contribution

From Summer 2025 to Fall 2025, Tommy and Sam studied the requirements for MAPF problems, improved MAPF algorithms based on previous students' work, implemented complex constraints, enhanced the visualizer's features, implemented a complex graph generator, and created testing scripts for MAPF algorithm validation.

Tommy primarily focused on the Conflict-Based Search (CBS) algorithm, including improving its performance, implementing parallel CBS, enhancing its features, correcting bugs in the visualizer, developing a complex graph generator, and creating a Bash Script to validate MAPF algorithms.

Sam primarily worked on Increasing Cost Tree Search (ICTS), including improvements to its algorithm, parallel ICTS, implementation of the K-hop, Radius, and Complex constraints, improvements to the graph generator, creation of a Windows PowerShell script for parallel MAPF algorithm validation, and support for migrating the project to Python 3.14.

# 2. Overview of System

The system provides an end-to-end framework for solving Multi-Agent Path Finding (MAPF) problems. It reads configuration files, transforms them into internal data structures, solves the problem using the selected MAPF algorithm, and ultimately generates a solution that can be visualized.

## 2.1 Input Pipeline

The input pipeline is responsible for translating external inputs into well-structured internal objects. The pipeline begins by loading a TOML or JSON configuration file. This file specifies all major components of a MAPF instance, including the graph SNAP file, agent definitions, constraints, and the algorithm to run.

Once loaded, the data passes through several transformation stages:

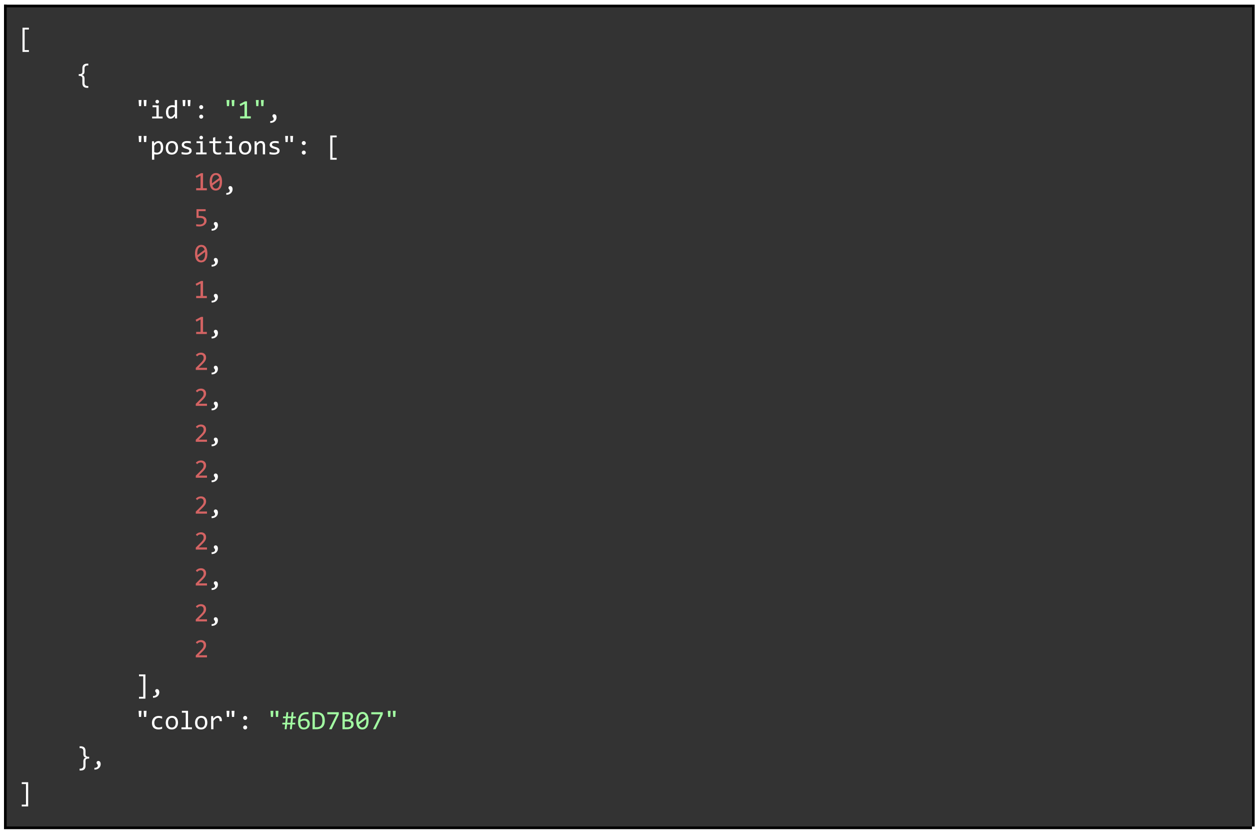
1. **Configuration Loading**
   The raw file is parsed into a Python dictionary. Both JSON and TOML are supported, with JSON preferred for complex, nested agent attributes.
2. **Algorithm Selection**
   The system extracts the name of the desired MAPF algorithm and maps it to the corresponding implementation (ICTS or CBS).
3. **Graph Construction**
   The graph file, which is encoded in SNAP format, is read and converted into a compressed sparse row (CSR) structure. This allows for efficient traversal over neighbouring nodes and reduces memory overhead.
4. **Agent Manager Construction**
   Each agent is instantiated with its start/goal nodes, waiting costs, and any constraint relationships. This stage also loads additional agent-specific metadata used during constraint checking.

## 2.2 Output Pipeline

The output pipeline prepares the final MAPF results for visualization. Instead of returning raw internal structures, the system converts and formats the solution into standardized JSON graph and solution files compatible with the visualizer.

The output pipeline will create the directory /out with the following files: solution.json, which defines how the agent should move the vertices; graph.json, which represents the nodes and contains the id and name (the label); and agent_map.json.

In solution.json, there will be three sections: id, positions, and color. The ‘id’ represents the agent’s id in the visualizer. The ‘positions’ represent how agents move between vertices, and the array will be the label of the vertices. The ‘color’ will be the colour in the visualizer.



```
[
    {
        "id": "1",
        "positions": [
            10,
            5,
            0,
            1,
            1,
            2,
            2,
            2,
            2,
            2,
            2,
            2,
            2,
            2
        ],
        "color": "#6D7B07"
    },
]
```

*Figure 1: Structure of solution.json*

In graph.json, there will be a nodes array, and inside the array, there are two sections: id and names. The id attribute represents the node's id, usually an integer. The name attribute represents the node label, which is displayed in the visualizer and is typically a string.

```
{
    "nodes": [
        {
            "id": 0,
            "name": "C^{0-0}_{0}"
        },
    ]
}
```

*Figure 2: Structure of graph.json*

In agent_map.json, the file shows the agent's id and the agent's label. It maps the map agent's id to the agent's label. However, this file is never used by the visualizer, which only accepts the graph.json and solution.json files.

```
{
    "0": "i",
    "1": "j",
    "2": "k",
    "3": "l",
    "4": "a",
    "5": "b",
    "6": "c",
    "7": "m",
    "8": "n"
}
```

*Figure 3: Structure of agent_map.json*

## 2.3 MAPF Algorithm

Two primary algorithms are supported: **Increasing Cost Tree Search (ICTS)** and **Conflict-Based Search (CBS)**. Both are optimal MAPF solvers that approach the problem differently.

### 2.3.1 ICTS

ICTS works by simultaneously exploring individual path costs for each agent and checking whether combinations of these paths can coexist without violating any constraints. It has a distinct two-tier structure:

- **High-level search** over cost vectors that represent the minimal allowed cost for each agent.
- **Low-level search** that generates all valid paths for a given cost and checks conflicts through MDD-based reasoning.

ICTS ensures optimality by expanding nodes in order of increasing total cost and returns the first conflict-free combination.

### 2.3.2 CBS

Conflict-Based Search (CBS) is an optimal MAPF algorithm that solves the multi-agent coordination problem using a hierarchical two-level framework. At its core, CBS separates pathfinding from conflict resolution by maintaining individual agent searches at the low level and resolving detected conflicts at the high level.

At the **high level**, CBS constructs a constraint tree (CT). Each node in this tree stores a set of constraints and a collection of individually optimal paths that satisfy them. When a conflict is detected between two agents, the node is expanded into two child nodes. Each child adds a constraint that prohibits one of the agents from performing the conflicting action. This branching process systematically explores the space of possible ways to resolve conflicts.

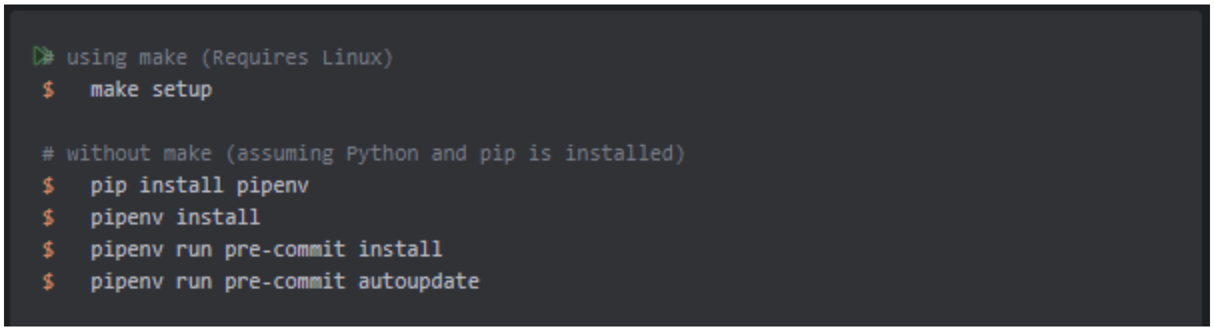
At the **low level**, each agent uses A* search to compute its optimal path under the constraints defined by the high-level node. Since constraints only apply to specific agents and timesteps, the low-level searches remain efficient and independent.

CBS ensures optimality by always selecting the lowest-cost node in the constraint tree for expansion. The algorithm terminates when a node is reached in which all agent paths are mutually conflict-free, guaranteeing that the returned solution is optimal with respect to the chosen cost metric.

## 2.4 Interface & Command line

The system can be executed directly through the command line once the appropriate environment has been configured. Before running the project, a Python virtual environment must be prepared. Python 3.14 and pip are required; this version of Python was selected because it provides full support for free-threaded parallelism (ensure that the *free-threading binaries* option is enabled during installation).

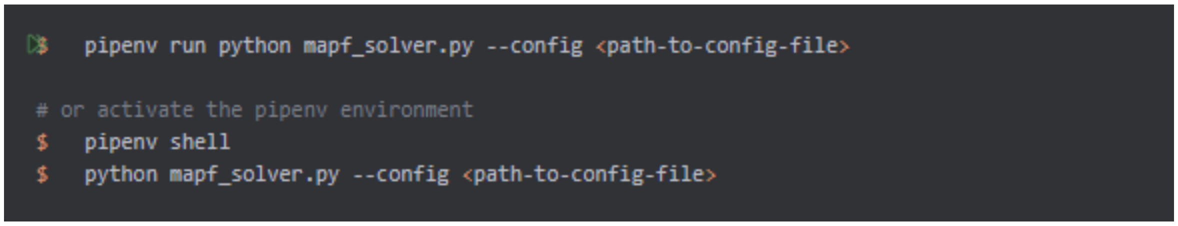
After installing Python, the project can be initialized using one of the setup methods illustrated below:



```
# using make (Requires Linux)
$   make setup

# without make (assuming Python and pip is installed)
$   pip install pipenv
$   pipenv install
$   pipenv run pre-commit install
$   pipenv run pre-commit autoupdate
```

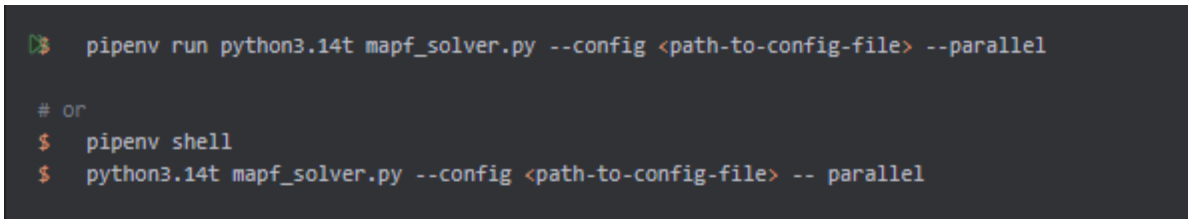
*Figure 4: Instructions for setting up the program (see the project README for additional details)*

Once the environment is configured, the MAPF solver can be launched using the following commands, each of which accepts a single input configuration file:configuration file:



```
$   pipenv run python mapf_solver.py --config <path-to-config-file>

# or activate the pipenv environment
$   pipenv shell
$   python mapf_solver.py --config <path-to-config-file>
```

*Figure 5: Running the program in sequential mode*



```
$   pipenv run python3.14t mapf_solver.py --config <path-to-config-file> --parallel

# or
$   pipenv shell
$   python3.14t mapf_solver.py --config <path-to-config-file> -- parallel
```

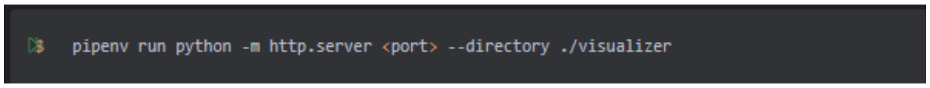
*Figure 6: Running the program in parallel mode*

The input configuration file defines the MAPF instance and includes the following fields:

- **graph_file**: Path to a SNAP-format graph file
- **vertex_label_file**: (Optional) Path to the vertex ID ↔ label mapping
- **mapf_algorithm**: Either “ICTS” or “CBS.”
- **constraints**: List of constraint class definitions
- **agents**: List describing all agents and their attributes

Concrete examples can be found in the /example directory.

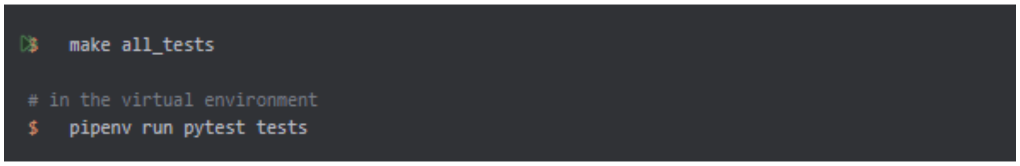
To visualize the resulting solution, the visualizer can be launched using the command shown below:



```
pipenv run python -m http.server <port> --directory ./visualizer
```

*Figure 7: Instructions for setting up and running the visualizer*

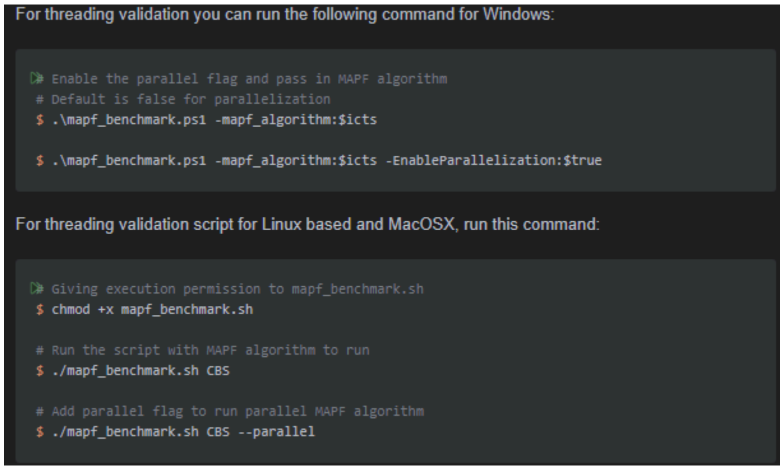
Testing support is provided through the pytest framework. All unit tests can be executed with the following command:



```
make all_tests

# in the virtual environment
$   pipenv run pytest tests
```

*Figure 8: Instructions for running pytest*

In addition to unit tests, the project includes Bash and PowerShell scripts designed to evaluate the correctness of the parallel execution model. These scripts perform repeated randomized validation runs. Parameters such as the number of iterations, graph size, and agent count range are configurable directly within the scripts:



For threading validation you can run the following command for Windows:

```
# Enable the parallel flag and pass in MAPF algorithm
# Default is false for parallelization
$ .\mapf_benchmark.ps1 -mapf_algorithm:$icts

$ .\mapf_benchmark.ps1 -mapf_algorithm:$icts -EnableParallelization:$true
```

For threading validation script for Linux based and MacOSX, run this command:

```
# Giving execution permission to mapf_benchmark.sh
$ chmod +x mapf_benchmark.sh

# Run the script with MAPF algorithm to run
$ ./mapf_benchmark.sh CBS

# Add parallel flag to run parallel MAPF algorithm
$ ./mapf_benchmark.sh CBS --parallel
```

*Figure 9: Instructions for running the parallel-validation shell scripts*

# 3. Improvement of MAPF Algorithms

Development efforts have been focused on enhancing the system's two MAPF solvers. While the foundational behaviour of ICTS and CBS remains consistent with their classical formulations, several improvements were made to ensure correctness and improve performance.

## 3.1 Increasing Cost Tree Search (ICTS) Algorithm improvement

When comparing the performance of ICTS to CBS, it was apparent that the performance of ICTS was significantly worse than that of CBS. This led to an investigation and an effort to improve the performance of sequential ICTS.

Performance analysis revealed that deep copying was responsible for a significant portion of the program's total runtime. Further examination showed that the object being duplicated was a list of CostToVertex instances. Since these instances are immutable, replacing the deep copy with a shallow copy of the top-level list preserves correctness while providing a substantial performance improvement.

Another improvement made was caching some values locally rather than recalling functions to get their values, which also led to another significant performance improvement. These values included the outgoing neighbouring nodes from a given node.

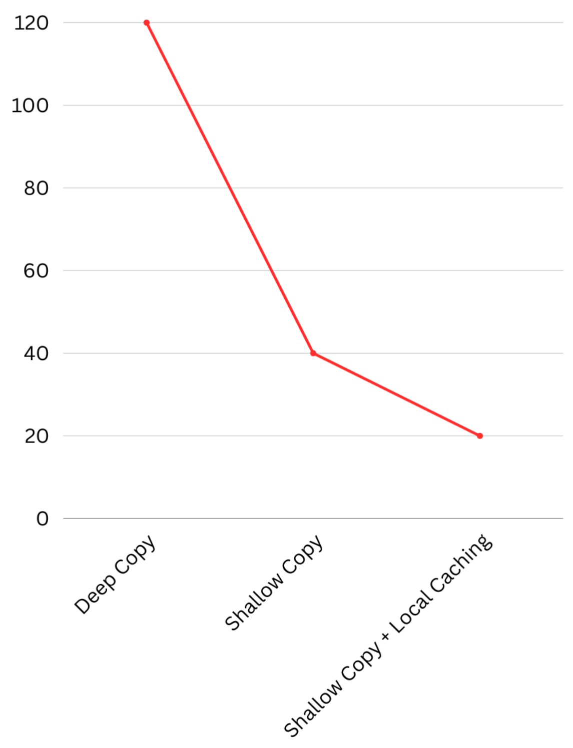


*Figure 10: Graph of sequential ICTS performance (in seconds) using large_input_file.toml*

## 3.2 Conflict-Based Search (CBS) Algorithm

Previous students implemented the Conflict-Based Search (CBS) algorithm; however, the code contained critical issues, particularly in the conflict-handling logic.

```
while open_list:
     current_node = heapq.heappop(open_list)
     self._normalize_paths(current_node.paths)
     constraint_result =
self.agent_manager.evaluate_agent_constraints(current_node.paths)

      if constraint_result.is_satisfied:
         return current_node.paths

      conflict_target = constraint_result.target
      conflict_competitor = constraint_result.competitor
      conflict_time = constraint_result.conflict_index

      blocked_node_target = conflict_target.path[conflict_time]
      blocked_node_competitor = conflict_competitor.path[conflict_time]
      agent1_id = conflict_target.agent.agent_id
      agent2_id = conflict_competitor.agent.agent_id

      log.info(f"Conflict detected: Agent {agent1_id} and Agent {agent2_id} at time
step {conflict_time}")
      base_constraints = current_node.constraints
      new_constraints_branch1 = {**base_constraints,
                                    agent1_id: base_constraints[agent1_id] +
[(conflict_time, blocked_node_target)]
                                    }
      new_constraints_branch2 = {**base_constraints,
                                    agent2_id: base_constraints[agent2_id] +
[(conflict_time, blocked_node_competitor)]

       for agent_id, branch_constraints in [(agent1_id, new_constraints_branch1),
(agent2_id, new_constraints_branch2)]:
           agent = self.agent_manager.agents[agent_id]
           agent_at_goal = current_node.paths[agent_id][-1] ==
agent.agent_endpoints.goal
           if not agent_at_goal or branch_constraints[agent_id]:
                search_method = AStar(self.graph, heuristics[agent.agent_id])
                path = search_method.search(agent.agent_id,
agent.agent_endpoints.start, agent.agent_endpoints.goal,
branch_constraints[agent_id])
            else:
                path = current_node.paths[agent_id]

            if path:
               new_paths = deepcopy(current_node.paths)
               new_paths[agent_id] = path
               cost = sum(len(p) for p in new_paths.values())
               child_node = CBSNode(cost=cost, constraints=branch_constraints,
paths=new_paths)
               heapq.heappush(open_list, child_node)
               log.info(f"Added child node for Agent {agent_id} with cost {cost}
using A*")
            else:
                log.info(f"No valid path found for Agent {agent_id} with current
constraints")

raise Exception("No conflict-free paths found for all agents.")
```

*Figure 11: Code Block that handles the conflict inside CBS (worked by previous students)*

The CBS logic is that the A* search algorithm will compute the initial path and, if a conflict is detected, expand the CBS Node until it finds a conflict-free path. One issue in the code is that, when a conflict is detected, CBS must recalculate the collision-free agent paths. However, the code does not save previous paths to avoid recomputing identical ones, keeps expanding the node, and ultimately falls into an infinite loop. To address this problem, the algorithm must track previously visited high-level solutions to avoid re-expansion and revisitation. This issue has been addressed by initializing `set()` before the `open_list` loop starts and checking whether the path is a duplicate. If the path is identical, the algorithm will ignore and avoid recalculation and an infinite loop. Using the set() function maintains performance because lookups for duplicate paths are O(1).

```
visited_signatures = set()
while open_list:
      current_node = heapq.heappop(open_list)
      initial_path_lengths = {aid: len(p) for aid, p in current_node.paths.items()}
      path_for_eval = deepcopy(current_node.paths)
      self._normalize_paths(path_for_eval)
      agent_path_list = [
          AgentPath(self.agents.get(agent_id), path)
          for agent_id, path in path_for_eval.items()
      ]
      constraint_result = self.agent_manager.evaluator.evaluate_constraints(
          self.graph,
          agent_path_list
      )
      if constraint_result.is_satisfied:
          solution_paths = deepcopy(current_node.paths)
          self._normalize_paths(solution_paths)
          return solution_paths
# Build a high-level state signature (tuple of per-agent paths) AFTER
normalization, so identical
       state_signature = tuple(sorted((aid, tuple(path_for_eval[aid])) for aid in
path_for_eval))
       if state_signature in visited_signatures:
          log.debug("Skipping since it is a duplicate agent path.")
          continue
       visited_signatures.add(state_signature)
       conflict_target = constraint_result.target
       conflict_competitor = constraint_result.competitor
       conflict_time = constraint_result.conflict_index
# Build branching data for the two agents involved in the conflict.
       target_path = conflict_target.path
       competitor_path = conflict_competitor.path
       conflicting_agents = [
           (
                conflict_target.agent.agent_id,
                target_path[conflict_time],
                conflict_time,
           ),
           (
                conflict_competitor.agent.agent_id,
                competitor_path[conflict_time],
                conflict_time,
           ),
       ]
      # Iterate over the two branches: one per conflicting agent.
      conflict_time_counter = time.perf_counter()
      for agent_id, conflicted_node, proposed_time in conflicting_agents:
          branch_constraints = deepcopy(current_node.constraints)
          agent = self.agents[agent_id]
          clipped_time = proposed_time
          raw_len = initial_path_lengths.get(agent_id, proposed_time)

          if proposed_time >= raw_len:
             clipped_time = raw_len - 1 if raw_len > 0 else 0
             agent_start = agent.agent_endpoints.start
             if clipped_time == 0 and conflicted_node == agent_start:
                clipped_time = 1
             new_constraint = (clipped_time, conflicted_node)
             branch_constraints[agent_id].append(new_constraint)
             search_method = AStar(self.graph, heuristics[agent.agent_id])
             path = search_method.search(agent.agent_id,
agent.agent_endpoints.start, agent.agent_endpoints.goal,
branch_constraints[agent_id])

             if path:
                new_paths = current_node.paths.copy()
                new_paths[agent_id] = path
                self._normalize_paths(new_paths)
                cost = sum(len(p) for p in new_paths.values())
                child_node = CBSNode(cost=cost, constraints=branch_constraints,
                                     paths=new_paths)
                heapq.heappush(open_list, child_node)
             else:
                log.info(f"No valid path found for Agent {agent_id} with current
                           constraints")

raise Exception("No conflict-free paths found for all agents.")
```

*Figure 12: Code Block with improvement of conflict handling*

Another bug in CBS is related to conflict detection. For instance, the evaluator in CBS returns no conflict detected for the path `{'i': [0, 0, 5], 'j': [2, 7, 8, 8], 'k': [5, 0, 1]}`. With these agent paths, there is vertex conflict in the time step 1, 'agent i' stays in vertex 0, and 'agent k' moves from 5 to 0. The bug was caused by the constraint implementation being valid only for ICTS, not CBS. Both vertex and swapping conflicts had this issue. This bug has been fixed, and the detailed explanation will be provided in the 4.1 Simple Conflicts section.

## 4. Constraints

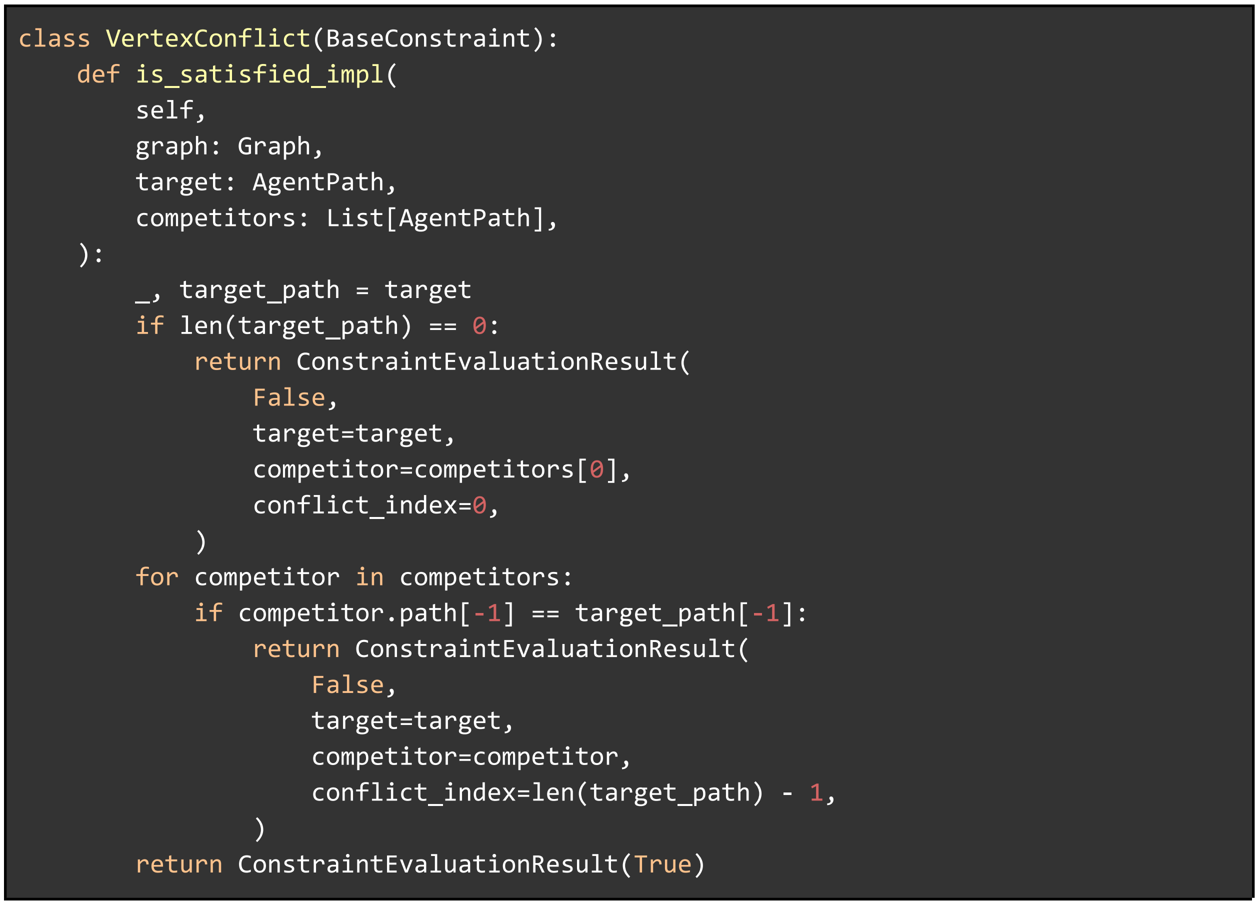
A Multi-agent path-finding solution is only valid if every agent's path is entirely collision-free. Constraints create rules that agents must adhere to in-order to have a complete, valid solution. The agent_manager class detects constraints; therefore, each search method does not need to know the type of conflict that occurred; it only knows that conflicts occurred.

### 4.1 Simple Conflicts

Previous students mainly implemented simple constraints; however, there was an issue with the CBS algorithm.

### 4.1.1 Vertex Conflict

A **Vertex Conflict** occurs when two agents attempt to occupy the same vertex at the same timestep. During constraint evaluation, the system compares the positions of all agents at each time step; if any pair shares a vertex, the conflict is reported immediately.



```
class VertexConflict(BaseConstraint):
    def is_satisfied_impl(
        self,
        graph: Graph,
        target: AgentPath,
        competitors: List[AgentPath],
    ):
        _, target_path = target
        if len(target_path) == 0:
            return ConstraintEvaluationResult(
                False,
                target=target,
                competitor=competitors[0],
                conflict_index=0,
            )
        for competitor in competitors:
            if competitor.path[-1] == target_path[-1]:
                return ConstraintEvaluationResult(
                    False,
                    target=target,
                    competitor=competitor,
                    conflict_index=len(target_path) - 1,
                )
        return ConstraintEvaluationResult(True)
```

*Figure 13: Previous Implementation of Vertex Conflict*

This implementation applies only to ICTS, which iteratively updates the agent path and checks for conflicts. However, unlike ICTS, CBS returns the full path from the initial point to the end point and must check for conflicts; therefore, the conflict implementation must traverse the complete list of agent paths. In the previous implementation, the CBS algorithm returns True, indicating that there is no conflict with the current agent paths. The previous implementation detects vertex conflict only at the end because it checks the last element of the path. Because of [-1], it reports no vertex conflict if no conflict occurs at the end. To resolve this issue, the conflict needs to be checked against the complete list to determine whether the agent paths contain vertex conflicts.

```python
class VertexConflict(BaseConstraint):
    def is_satisfied_impl(
        self,
        graph: Graph,
        target: AgentPath,
        competitors: List[AgentPath],
    ):
        _, target_path = target
        if len(target_path) == 0:
            return ConstraintEvaluationResult(
                False,
                target=target,
                competitor=competitors[0],
                conflict_index=0,
            )

        for competitor in competitors:
            competitor_path = competitor.path
            for t in range(len(target_path)):
                if competitor_path[t] == target_path[t]:
                    return ConstraintEvaluationResult(
                        False,
                        target=target,
                        competitor=competitor,
                        conflict_index=t
                    )
        return ConstraintEvaluationResult(True)
```

*Figure 14: Current Implementation of Vertex Conflict*

### 4.1.2 Swapping Conflict

A **Swapping Conflict** arises when two agents attempt to traverse the same edge in opposite directions at the same timestep. For example, one agent moves from vertex A to vertex B while another moves from B to A simultaneously. Although neither agent occupies the same vertex, this head-on traversal is considered an invalid interaction under this constraint. Similar to the vertex constraint, there was a logical issue with the swapping conflict.

```
class SwappingConflict(BaseConstraint):
    def is_satisfied_impl(
        self,
        graph: Graph,
        target: AgentPath,
        competitors: List[AgentPath],
    ) -> ConstraintEvaluationResult:
        target_path = target.path
        if len(target_path) < 2:
            return ConstraintEvaluationResult(
                False,
                target=target,
                competitor=competitors[0],
                conflict_index=0,
            )
        for competitor in competitors:
            competitor_path = competitor.path
            for t in range(min(len(target_path), len(competitor_path)) - 1):
                if target_path[t] == competitor_path[t+1] and target_path[t+1] ==
competitor_path[t]:
                    return ConstraintEvaluationResult(
                        is_satisfied=False,
                        target=target,
                        competitor=competitor,
                        conflict_index=t,
                    )
        return ConstraintEvaluationResult(True)
```

*Figure 15: Previous Implementation of Swapping Conflict*

If the agent is forbidden from moving to another vertex at time t, it means the agent is now allowed to wait and is forced to move to another vertex at time t. However, the swapping conflict occurs at t + 1; therefore, to detect it, the conflict index must be at t + 1. If the conflict_index is at t, the agent is forced to move to other vertices, which may require an impossible replan; therefore, no alternative paths exist, and the search algorithm will return as no solution is visible.

```
class SwappingConflict(BaseConstraint):
    def is_satisfied_impl(
        self,
        graph: Graph,
        target: AgentPath,
        competitors: List[AgentPath],
    ) -> ConstraintEvaluationResult:
        target_path = target.path
        if len(target_path) < 2:
            return ConstraintEvaluationResult(
                False,
                target=target,
                competitor=competitors[0],
                conflict_index=0,
            )
        for competitor in competitors:
            competitor_path = competitor.path
            for t in range(min(len(target_path), len(competitor_path)) - 1):
                if (
                    target_path[t] == competitor_path[t + 1] and
                    target_path[t + 1] == competitor_path[t]
                ):
                    return ConstraintEvaluationResult(
                        is_satisfied=False,
                        target=target,
                        competitor=competitor,
                        conflict_index=t + 1,
                    )
        return ConstraintEvaluationResult(True)
```

*Figure 16: Current Implementation of Swapping Conflict*

## 4.2 K-Hop Conflict

The **K-Hop Conflict** constraint generalizes the classic vertex conflict by preventing agents from approaching each other within a specified number of graph hops. Instead of only disallowing collisions at the same vertex, the constraint requires competitors to remain separated by at least *k* unweighted edges at every timestep.

For each time index, the constraint inspects the target agent's current position and compares it against the positions of all competing agents at the same timestep. For each competitor, a breadth-first search (BFS) is performed from that competitor's location, exploring outward up to *k* hops. All vertices within this unweighted distance form a forbidden region/area.

If the target agent occupies any vertex in this region, the constraint is considered violated. The constraint is satisfied only if all agents avoid entering each other's *k*-hop neighbourhoods throughout the entire duration of their paths.

To create a K-Hop conflict config file, you need to add an additional 'parameter' field within the constraints section (i.e. `parameters = { k = 1 }`). This value is then taken through the input pipeline and used to initialize the K-Hop conflict object.

## 4.3 Radius Conflict

The **Radius Conflict** constraint extends the idea of distance-based safety from unweighted hop metrics to weighted graph distances. Similar in spirit to the Height Conflict constraint, which enforces local spatial exclusion, the Radius Conflict computes a weighted radius around each competitor's current position and disallows the target agent from entering that region.

To determine the forbidden area, the constraint runs Dijkstra's algorithm from the competitor's vertex, accumulating edge weights until the total cost exceeds the specified radius $r$. All reachable vertices with cost $\leq r$ constitute restricted positions. At each timestep, the target agent's location is compared to the forbidden set generated for every competitor. If the target lies within a competitor's radius, a violation is reported, and the timestep is recorded as the conflict index.

Where the K-Hop Conflict interprets distance strictly in terms of hop count, the Radius Conflict allows for fine-grained control on weighted graphs and aligns closely with scenarios requiring weight-based safety zones.

Similar to K-Hop Conflict, Radius Conflict's config file also has an extra 'parameter' field (i.e. `parameters = { r = 1.0 }`).

## 4.4 Complex Constraint

The **Complex Constraint** module introduces a high-fidelity physical interaction model in which agents navigate a structured environment composed of sectors, transition vertices, and mechanical gears. Unlike previous constraints, this mechanism enforces *multi-step, geometry-dependent* behaviours that simulate gear rotation, directional pushing, and chain-reaction propagation across multiple agents.

### 4.4.1 Structural Model

The environment is partitioned into **sectors**, each containing six labelled vertices arranged in a circular pattern. Among these six vertices, three correspond to **gear-contact positions** (subscripts 0, 2, and 4), and the other three correspond to **transition positions** (subscripts 1, 3, and 5).

Agents can occupy any of the six vertices of a sector and will contact three gears simultaneously, depending on their orientation (head/tail, left, right). These contacts determine how their movement affects mechanical rotation.

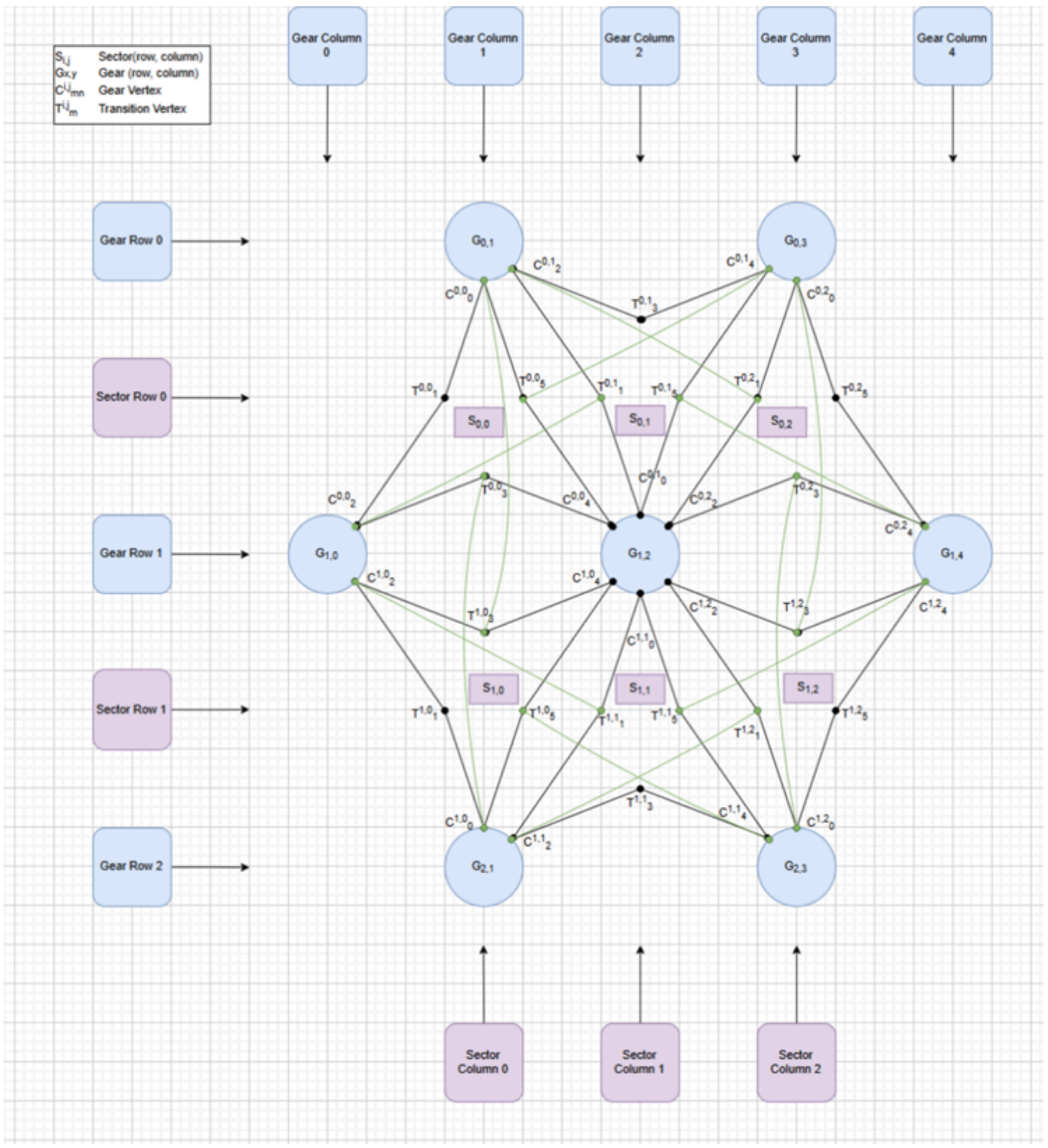


*Figure 17: Example 2x3 sector graph for complex constraint*

## 4.4.2 Mapping and Initialization

Before evaluating any movement, the constraint constructs several internal maps:

- **VERTEX_MAP** links graph node IDs to complex constraint style nodes with their own (row, col, subscript).
- **GEAR_VERTEX_MAP** assigns each vertex touching a gear to that specific gear.
- **GEAR_MAP** stores Gear objects with rotation state and static agent occupancy.
- **SECTOR_GEAR_MAP** associates each sector with its three connected gears.

These mappings are created dynamically from labelled graph inputs, ensuring compatibility across arbitrary grid sizes. During initialization, each agent is inspected at a consistent path index to determine which gears it is currently touching and whether it does so via head, tail, left, or right. This produces the initial **static_agents** occupancy lists for all gears.

### 4.4.3 Core Agent Behaviours

The complex constraint recognizes **three fundamental movement behaviours**, each corresponding to a distinct geometric transition between gear and transition vertices. These behaviours define whether a movement induces clockwise (CW), counterclockwise (CCW), or steady rotations on specific gears

#### Behaviour 1 — Gear → Transition (Same Sector)

When an agent moves its head from a gear-contact vertex to its paired transition vertex, two adjacent gears must rotate in the same direction, while the third gear in the sector must remain steady. Any conflicting rotation assignments cause an immediate constraint violation.

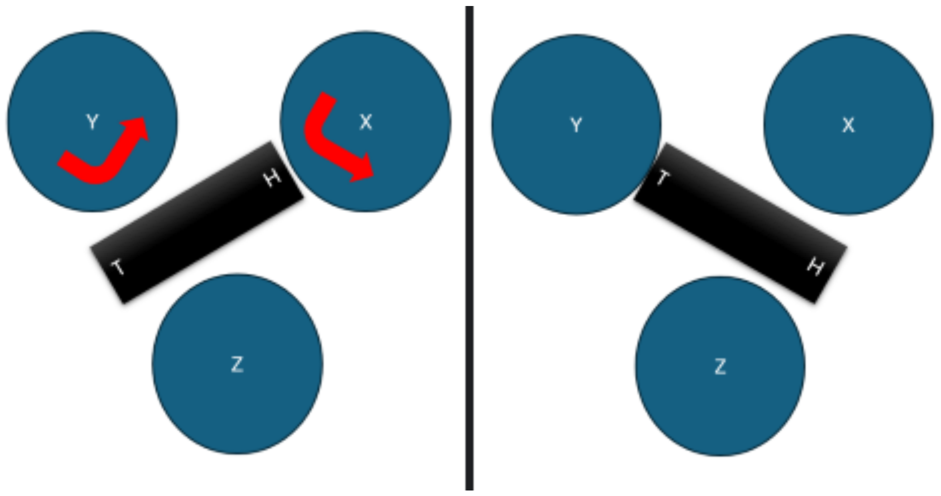


*Figure 18: Example of Behaviour 1*

### Behaviour 2 — Transition → Gear (Same Sector)

This is the reverse of Behaviour 1. Moving into a gear from its neighbouring transition vertex induces the inverse rotation direction. The same consistency checks apply to ensure no contradictions occur.

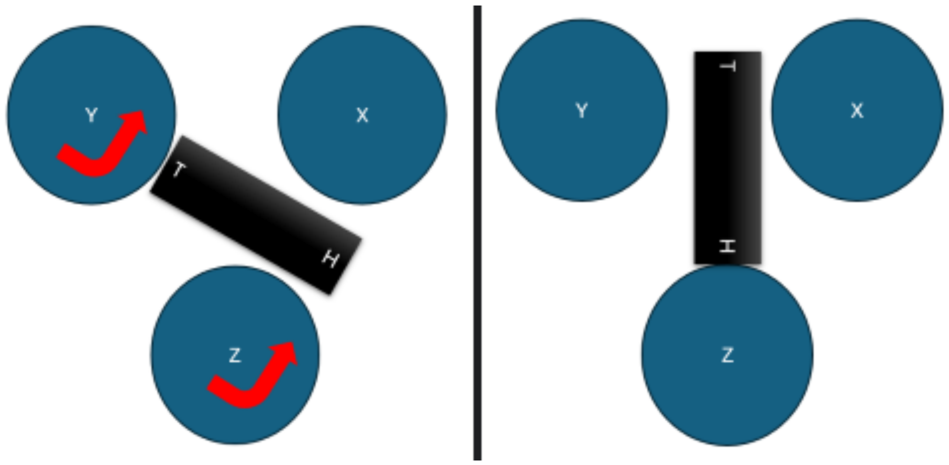


*Figure 19: Example of Behaviour 2*

### Behaviour 3 — Transition → Gear (Different Sector)

This movement occurs when the agent crosses sector boundaries. In this case, the left and right gears rotate in opposite directions, and the destination gear must remain steady.

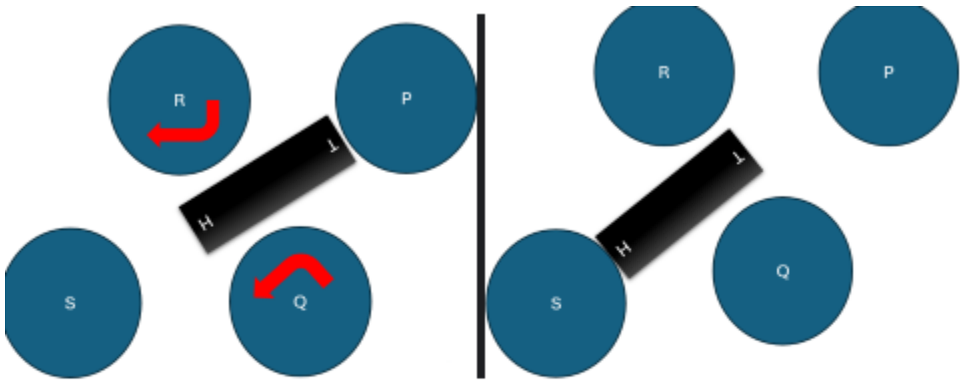


*Figure 20: Example of Behaviour 3*

### 4.4.4 Rotation Propagation and Forced Movement

After classifying all agents' movements at the current timestep, the constraint evaluates **propagation** effects. Any gear assigned a non-steady rotation may push agents that are still registered as static on that gear.

Each push attempt updates the static occupancy of gears accordingly. A conflict arises if:

- The target push position is already occupied by an agent in the same orientation role,
- The push would require stepping into an invalid vertex, or
- A gear rotation contradicts past assignments from this timestep.

The constraint is satisfied only when all agents' movements and all resulting propagation steps can be realized without violating any of these geometric, rotational, or positional requirements.

# 5. Parallelization Implementation

The system incorporates parallelism to accelerate the high-cost components of MAPF search. The parallelization strategy is built on the Python 3.14 free-threaded runtime, which enables true OS-level parallelism in CPU-bound workloads.

## 5.1 ICTS Parallelization

Looking at the ICTS implementation, it was clear that the search space explorer was the biggest bottleneck, and it was observed that as the input size grew, the search space expanded combinatorially, and therefore drastically reduced performance. Therefore, it was decided as the best option for implementing parallelism within ICTS.

The ICTS search space explorer uses a modified parallel depth-first search designed to take advantage of subtree independence within the Increasing Cost Tree Search framework. Since each subtree rooted at a sufficiently deep node is independent of the others, the algorithm divides the search space into partitions that can be processed in parallel.

The implementation follows a **three-phase depth-partitioned strategy**:

### 5.1.1 Controlled Sequential Expansion (Partition Formation)

Rather than launching threads at the start of the search (which would produce extremely small work units and heavy overhead), the algorithm incrementally expands the search tree *sequentially* until it reaches a suitable partition depth.

During this stage:

- A stack-based DFS explores the first few layers of the search tree.
- Each partial path at this depth becomes a *frontier node*.

Early pruning rules are preserved to reduce unnecessary branching:

- Paths exceeding the allowed cost or depth are discarded.
- Cycles can be eliminated based on configuration.
- Wait actions are added conditionally.

Once the frontier reaches the threshold, it becomes the set of independent tasks for the thread pool.

### 5.1.2 Parallel Subtree Exploration

Each frontier node represents an entire subtree whose exploration is independent of all others.
A worker thread receives exactly one subtree:

- It performs a complete DFS from that root.
- It applies all pruning rules locally.
- It accumulates results in a private ‘SearchSpacePaths’ instance.

Crucially, **no locks are required**, since:

- Each thread reads from shared structures but never mutates them.
- All generated paths are stored in local thread-private memory.
- There is no shared global frontier or global stack.

This yields good parallel scaling, especially on dense graphs.

### 5.1.3 Result Merging

When threads complete their work:

- The main thread collects each future as it finishes (using ‘as_completed’).
- Path collections are merged into a global result structure.
- Since merging only involves appending immutable path objects, the merge cost is minimal.

Load balancing naturally improves because threads complete at different times, and the 'as_completed' pattern allows fast workers to contribute earlier.

## 5.2 CBS Parallelization

The essential logic and the computation time of the current implementation of CBS are:

1. Precompute the heuristics (5-10% of computing time)
2. Generate initial paths for all agents (10-15% of computing time)
3. Detect conflicts and rerun A* to generate that the path is conflict-free (80-90% of computing time)

The overall runtime of CBS is dominated by the number of times that the low-level search is called. In a large or complex problem, the high-level search may expand thousands of nodes, requiring the low-level search to run hundreds of thousands of times across all agents, which is the biggest bottleneck in CBS logic. The current parallel CBS algorithm implementation has approached parallel node expansion in the A* search algorithm, as well as CBS initial path finding.

### 5.2.1 CBS Agent Initial Path Finding

In this initial path-finding phase, the focus is on precomputing heuristics for the agent's path and generating initial paths for all agents. The step-by-step process will be:

1. Heuristic precomputation and initialize the root CBS node with deterministic ordering of agents
2. Parallel execution is configured using a ThreadPoolExecutor, and the task is submitted to the executor.
3. Each agent with a multithread configuration will run the low-level A* search.
4. The main thread waits for the futures to complete using the as_completed function and collects resulting paths into root paths. If any agent fails, the path-finding process terminates early to reduce overhead and resource usage.

This process has not significantly affected performance, as CBS is more time-consuming in conflict resolution; however, this phase did not incur any time overhead and reduced computational time by 1/10.

### 5.2.2 A* Parallel Node Expansion

CBS primarily employs the A* search algorithm to determine the agent's path, making its architecture highly sensitive to the speed of its low-level search. If the performance of A* search improves, CBS is expected to benefit as well. One popular approach to improving A* search is parallel node expansion.

The core concept of A* parallel node expansion is that for a problem with N agents, the goal is to find N paths ($P_1$, $P_2$,..., $P_n$) and execute the individual A* search for each agent sequentially. The parallel node expansion launches all N searches simultaneously to reduce computational time, thereby improving the overall performance of A* search. By running these N searches in parallel, the time taken to expand that high-level node is reduced from the sum of all individual search times to the time taken by the most extended individual search. The substantial reduction in the time spent per node enables high-level search to explore the conflict space much more rapidly. Since CBS spends 80-90% of its computing time on low-level search, parallelizing this section reduces the total clock time required to find an optimal solution. After each agent calculates its own path, the results are merged in the as_completed(future) loop, and the CBS high-level search checks for conflicts.

Parallelizing A* searches at the low level doesn't affect CBS's optimality guarantees but significantly improves its scalability and runtime, making it viable for environments with more agents. In the test cases, it uses a 10x10 grid graph with 60 agents.

# 6. Complex Graph Generator

The complex graph generator provides automated construction of the specialized graph structures required for evaluating the mechanical, gear-based constraint model. The resulting graphs follow a strict geometric and labelling format, enabling the system to reason about sector layouts, transition vertices, and gear adjacencies without manual intervention. This ensures consistency across test cases and streamlines experimentation with larger or more intricate configurations.

## 6.1 What is a Complex Graph?

A **complex graph** is a structured, labelled environment designed specifically for constraints that depend on mechanical interactions between sectors and gears. Unlike standard MAPF grid or arbitrary graph inputs, a complex graph encodes:

**• Sectorized Layout**

The environment is divided into a grid of *sectors*, each containing exactly six vertices arranged in a circular pattern. Each vertex is labelled using a canonical notation such as:

- ${C^{i-j}}_k$ - a **gear vertex** within sector (i,j)
- ${T^{i-j}}_k$ - a **transition vertex** within sector (i,j)

Here, k ∈ {0,1,2,3,4,5} denotes the subscript that specifies the vertex's position within the sector.

**• Embedded Gear Network**

Every sector is surrounded by up to three mechanical gears. These gears exist at positions derived from the parity of the sector's coordinates and are referenced implicitly through the vertex labelling. Gear-contact vertices form the physical interface between agents and gears, enabling the complex rotation and push behaviours used by the constraint system.

## 6.2 Purpose of Graph Generator

The graph generator automates the creation of these complex graphs, ensuring the mechanical model is represented accurately and consistently. Its key roles include:

**1. Eliminating Manual Construction Effort**

Manually writing out the vertices, labels, and edges for each sector would be error-prone and scale poorly. The generator programmatically constructs:

- all six vertices per sector,
- all correct edges within and across sectors,
- all label strings in the required ${C^{i-j}}_k$ or ${T^{i-j}}_k$ format.

This guarantees correctness and conformity to the expected structure.

**2. Ensuring Compatibility With Complex Constraints**

The complex constraint module relies on specific structural assumptions:

- precise gear placement,
- consistent subscript interpretation,
- correct mapping from vertices to gears,
- predictable adjacency relationships.

The generator ensures these assumptions are always satisfied, enabling the constraint logic (e.g., initialization maps, rotation propagation, sector geometry) to operate reliably on any generated graph.

**3. Enabling Scalable Testing and Benchmarking**

Because the generator accepts dimensions (M × N) and produces fully structured sector grids, users could use the resulting graphs for debugging or performance tests. This flexibility is essential, especially when working with the Complex Constraint.

## 6.3 Current Requirement for the input graph

The current MAPF system takes two types of input graphs: JSON and TOML graph files. Although the two file types differ, their structures are the same. This can be visible in README.md in the project repository.

### 6.3.1 Graph File

All graphs must be in SNAP format and cannot be modified after being parsed by the pipeline.

Supported graph types are:

- Directed / Undirected
- Weighted / Unweighted

It needs to be specified in the file:

```
{
  "graph_file": "./examples/example_snap_graphs/UW_D_graph.snap"
}
```

*Figure 21: Graph file path in input_graph.json*

```
graph_file = "./examples/example_snap_graphs/UW_D_graph.snap"
```

*Figure 22: Graph file path in input_graph.toml*

### 6.3.2 Vertex Label File

It maps the vertex id to a human-readable label. This section is only used in the JSON graph file.

```json
{
  "vertex_labels_file": "./examples/example_snap_graphs/graph_labels.txt"
}
```

*Figure 23: Vertex Label file path in input_graph.json*

### 6.3.3 MAPF Algorithm Selection

The system uses ICTS or CBS to process the input graph file.

```json
{
  "mapf_algorithm": "CBS"
}
```

*Figure 24: MAPF algorithm in input_graph.json*

```toml
mapf_algorithm = "ICTS"
```

*Figure 25: MAPF algorithm in input_graph.toml*

### 6.3.4 Constraints

The constraints list contains constraint objects and must be initialized with the file path to the Python file that defines them.

```json
{
  "constraints": [
    {
      "constraint_class_name": "VertexConflict",
      "file_path": "./mapf/constraints/implementations/vertex_conflict.py"
    },
    {
      "constraint_class_name": "SwappingConflict",
      "file_path": "./mapf/constraints/implementations/swapping_conflict.py"
    }
  ]
}
```

*Figure 26: Constraints list in input_graph.json*

```
[[constraints]]
constraint_class_name = "VertexConflict"
file_path = "./mapf/constraints/implementations/vertex_conflict.py"

[[constraints]]
constraint_class_name = "SwappingConflict"
file_path = "./mapf/constraints/implementations/swapping_conflict.py"
```

*Figure 27: Constraints list in input_graph.toml*

## 6.3.5 Agents

The agent field is the list of agent objects with specific fields:

- agent_id: identifier
- start: Starting vertex
- goal: Target vertex
- wait_cost: cost for waiting
- wait_at_goal_cost: cost for waiting at goal
- constraints: list of applicable constraints
- attributes: custom properties about each agent

```
{
  "agents": [
    {
      "agent_id": "i",
      "start": 0,
      "goal": 2,
      "wait_cost": 0,
      "wait_at_goal_cost": 0,
      "constraints": [
        {
          "constraint_class_name": "VertexConflict",
          "competing_agents": ["j"]
        },
        {
          "constraint_class_name": "SwappingConflict",
          "competing_agents": ["*"]
        }
      ],
      "attributes": {
        "height": 10,
        "speed": 2,
        "states": ["ACTIVE", "WAITING", "SHUT_DOWN"],
        "colors": {
          "top": "blue",
          "bottom": "black"
        }
      }
    },
    {
      "agent_id": "j",
      "start": 1,
      "goal": 5,
      "wait_cost": 0,
      "wait_at_goal_cost": 0,
      "constraints": [
        {
          "constraint_class_name": "VertexConflict",
          "competing_agents": ["i", "k"]
        }
      ],
      "attributes": {}
    }
  ]
}
```

*Figure 28: Agents list in input_graph.json*

```
# agent i
[[agents]]
agent_id = "i"
start = 0
goal = 2
wait_cost = 0
wait_at_goal_cost = 0

[[agents.constraints]]
constraint_class_name = "VertexConflict"
competing_agents = ["j"]

[[agents.constraints]]
constraint_class_name = "SwappingConflict"
competing_agents = "*"

[[agent.attributes]]
height = 10
speed = 2
state = ["ACTIVE", "WAITING", "SHUT_DOWN"]
```

*Figure 29: Agents list in input_graph.toml*

## 6.4 Structure of the Complex Graph Generator

The structure of the complex graph generator is shown as follows:

```
├── mapf_graph_generator/
│   ├── __init__.py        # Initialize the graph generator module
│   ├── agent_parser.py    # Agent parser
│   ├── arg_parser.py      # Argument parser
│   ├── graph_gen.py       # Generate the complex graph
│   ├── complex_graph_config.py # configuration for the complex graph
│   ├── mapf_determination.py   # Parser that determines the MAPF algorithm
│   ├── vertex_id.py       # Vertex id that formats the complex graph
```

## 6.5 Parameter Requirement

In order to run the complex graph generator, the user needs to pass in several parameters:

```
class Args:
    column: int # (Required) Column of the Gear Graph
    row: int # (Required) Row of the Gear Graph
    mapf_algorithm: str # (Required) Which MAPF algorithm want to use
    file_path: str # (Required) File path with agent list's detail
    sector_labels: bool # Includes the vertices' sector label
```

*Figure 30: Argument requirement for the complex graph generator*

- column: column of the gear graph
- row: row of the gear graph
- mapf_algorithm: Which MAPF algorithm does the user want to use
- file_path: file path with agent lists detail
- sector_labels: includes the vertices’ sector label

### 6.5.1 Agent List Detail

One of the complex graph generator requirements is the file path of the agent’s list. Each agent has their own properties to configure. Since each agent’s configuration list is hard to configure in the parameter, the user needs to pass in the text file path with the specific agent’s details.

The structure is shown below:

```
agent_id: # Agent id
start: # Start point
goal: # Goal point
wait_cost: # Wait Cost
wait_at_goal_cost: # Wait Cost at Goal
constraints: # List of constraints
  ConflictName: # Agent id to apply this conflict
attributes: # Attribute detail for this agent
  height:
  speed:
  states:
  colors:
    top:
    bottom:
```

*Figure 31: Structure for the Agent List Detail*

```
agent_id: i
start: 0
goal: 5
wait_cost: 1
wait_at_goal_cost: 1
constraints:
  VertexConflict: j, k
  SwappingConflict: *
attributes:
  height: 10
  speed: 2
  states: ACTIVE,WAITING,SHUT_DOWN
  colors:
    top: blue
    bottom: black
```

*Figure 32: Example for the Agent List Detail*

## 6.6 How to run the Graph Generator

The complex graph generator can be run through the terminal. It has to follow this structure:

```
python mapf_graph_generator.py \
    -SC column \
    -SR row \
    -MAPF algorithm \
    -F file-path \
    -SL
```

*Figure 33: Argument requirement for the complex graph generator*

Example for the command line:

```
python mapf_graph_generator.py \
    -SC 3 \
    -SR 2 \
    -MAPF CBS \
    -F in/agent_list.example.txt \
    -SL
```

*Figure 34: Example command for the complex graph generator*

## 6.7 Output

This program will generate in this format:

```
├── in/
    ├── input_graph_file.json
    ├── graph_<row>x<col>.snap
    ├── labels_<row>x<col>.txt
```

*Figure 35: Output structure of the complex graph generator*

# 7. Performance

Performance evaluation focused on measuring the impact of parallel execution across the two supported MAPF solvers. Because ICTS and CBS differ substantially in their search structures, each algorithm exhibits unique scaling behaviour. Throughout development, sequential versions of both algorithms were benchmarked to establish baselines, followed by an assessment of the parallel implementations under identical input conditions.

The results demonstrate that parallelism substantially reduces computation time for both ICTS and CBS. However, the performance of ICTS remains substantially worse than that of CBS, particularly on moderate-to-large paths. The fundamental architectural difference between ICTS's exhaustive precomputation and CBS's lazy incremental approach creates an insurmountable performance gap in practical multi-agent pathfinding applications. ICTS's requirement to enumerate all feasible paths before conflict detection renders it unsuitable for graphs containing significant path diversity. This makes direct comparison between the two algorithms very difficult.

## 7.1 ICTS Sequential vs Parallel

Prior to optimization, ICTS exhibited significantly poorer performance than CBS, making direct comparisons between the two algorithms impractical. The search-space explorer within ICTS grows combinatorially with depth and cost. These factors caused the sequential version to become prohibitively slow on even moderately sized inputs.

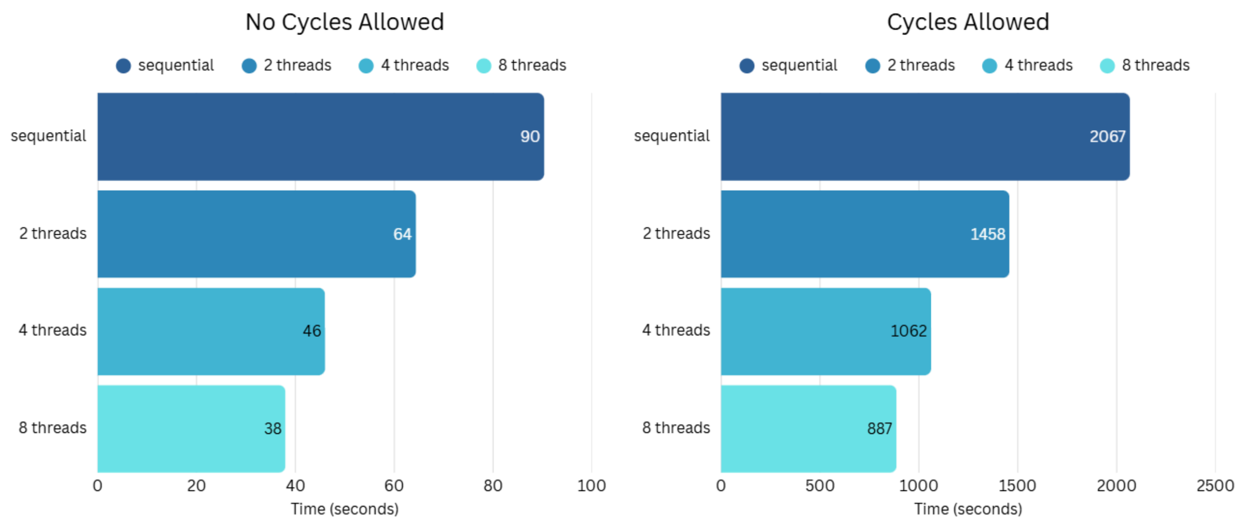


*Figure 36:* Sequential vs. parallel ICTS performance with and without cycles using ‘large_input_file.toml’

The benchmark results show us that parallel ICTS outperforms sequential ICTS across all non-trivial cost and depth budgets. It is also important to note that disallowing cycles provides an enormous performance boost, as it prevents the search from revisiting previously explored vertices and significantly reduces branching. In this configuration, both sequential and parallel modes execute considerably faster due to reduced path redundancy. However, forbidding cycles is not universally applicable. Some input instances require temporary backtracking or revisiting vertices to reach feasible solutions. Thus, while “no cycles allowed” improves runtime, it cannot be treated as a global optimization.

Overall, the experiments demonstrate that the parallel ICTS implementation is substantially more scalable than the original sequential version. With cycle elimination enabled (when valid), ICTS becomes dramatically faster and significantly more competitive within the overall system, enabling it to handle instances that were previously infeasible to compute.

## 7.2 CBS Sequential vs Parallel

The sequential version of CBS performed significantly worse than its parallel counterpart, rendering it impractical for real-time, large-scale scenarios. The execution time of CBS is dominated by the low-level search, where numerous A* pathfinding queries for individual agents must be resolved. These factors rendered the sequential version prohibitively slow for moderately sized inputs. The current parallel CBS uses A* parallel node expansion, distributing the low-level search across N agents across multiple threads, to evaluate performance gains relative to the sequential version.

The performance testing environment is a 10x10 complex graph generated by the complex graph generator with 60 agents. The runtime was compared with a sequential implementation (1 thread) and multiple-threaded implementations (using 2, 4, and 8 threads).

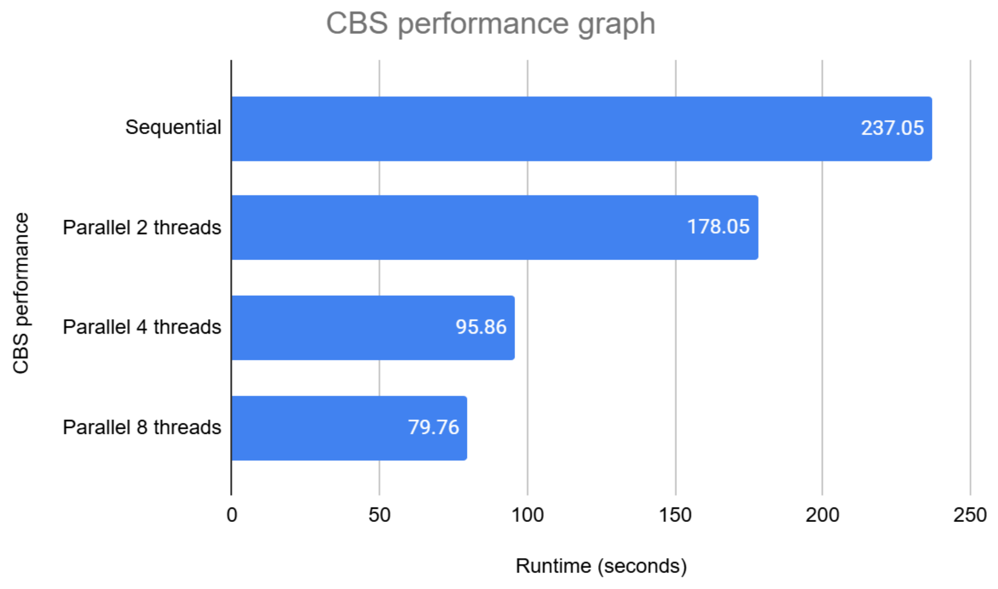


*Figure 37:* Sequential vs. parallel CBS performance (10x10 graph) with 60 agents

The benchmark results clearly show that parallel CBS outperforms sequential CBS across all tested thread counts. The measured speedup confirms that the parallel approach effectively distributes the dominant low-level A* search workload, thereby significantly reducing the overall runtime. Since CBS pathfinding is non-deterministic, the runtime may vary across scenarios; however, the results indicate that performance increases with thread count. With smaller input sizes, there may be overhead due to thread initialization, and the parallel version may be slower than the sequential version. Overall, the experiment demonstrates that the parallel CBS implementation is substantially more scalable than the sequential version. This optimization is crucial, making CBS dramatically faster and significantly more competitive within the overall system, enabling it to handle instances that were previously infeasible to compute.

# 8. Testing

Testing within this project focuses on verifying the correctness of solutions produced by both the sequential and parallel MAPF solvers. Since parallel execution introduces non-determinism in scheduling and branching order, dedicated validation procedures are essential to ensure that the solver still produces conflict-free solutions regardless of thread layout or timing.

Two layers of testing are employed:

1. **Unit tests (pytest)** – These are implemented throughout the development process, and validate individual components such as search-space utilities, constraint evaluation, etc.
2. **End-to-end validation scripts** – These repeatedly generate random inputs, run the solver in parallel mode, and confirm that every produced solution satisfies all constraints.

## 8.1 Testing Scripts for Windows

A dedicated PowerShell script is provided for Windows users to automate large-scale testing of the parallel MAPF solver. The script repeatedly generates random input instances, invokes the chosen MAPF algorithm in parallel mode, and uses an independent validation module to confirm that the output solution is correct.

**Purpose of the Script**

The Windows test script is designed to:

- Generate a large number of randomized MAPF problem instances.
- Measure solution reliability.

**How the Script Works**

The script performs the following sequence for each run:

1. **Environment setup**
   Ensures the output directory exists and initializes counters for benchmark reporting.
2. **Graph generation**
   Creates a consistent complex-graph environment using the project's graph generator.
3. **Randomized instance creation**
   Picks a random number of agents and uses the input generator to produce a new configuration for the selected algorithm.
4. **Parallel MAPF solving**
   Executes the solver with the ‘--parallel‘ flag enabled, ensuring that the test checks the multi-threaded execution path.
5. **Solution validation**
   Runs the validation tool that checks if a solution is conflict-free, if so it is counted as a valid solution
6. **Progress reporting**
   Every fixed number of iterations, the script prints the current validation rate to give a real-time view of solver correctness.
7. **Benchmark summary**
   After completing all runs, it outputs:
   - total successful validations
   - success percentage
   - total execution time
   - average runtime per instance

This provides an empirical measure of the parallel correctness over a large number of randomized test cases.

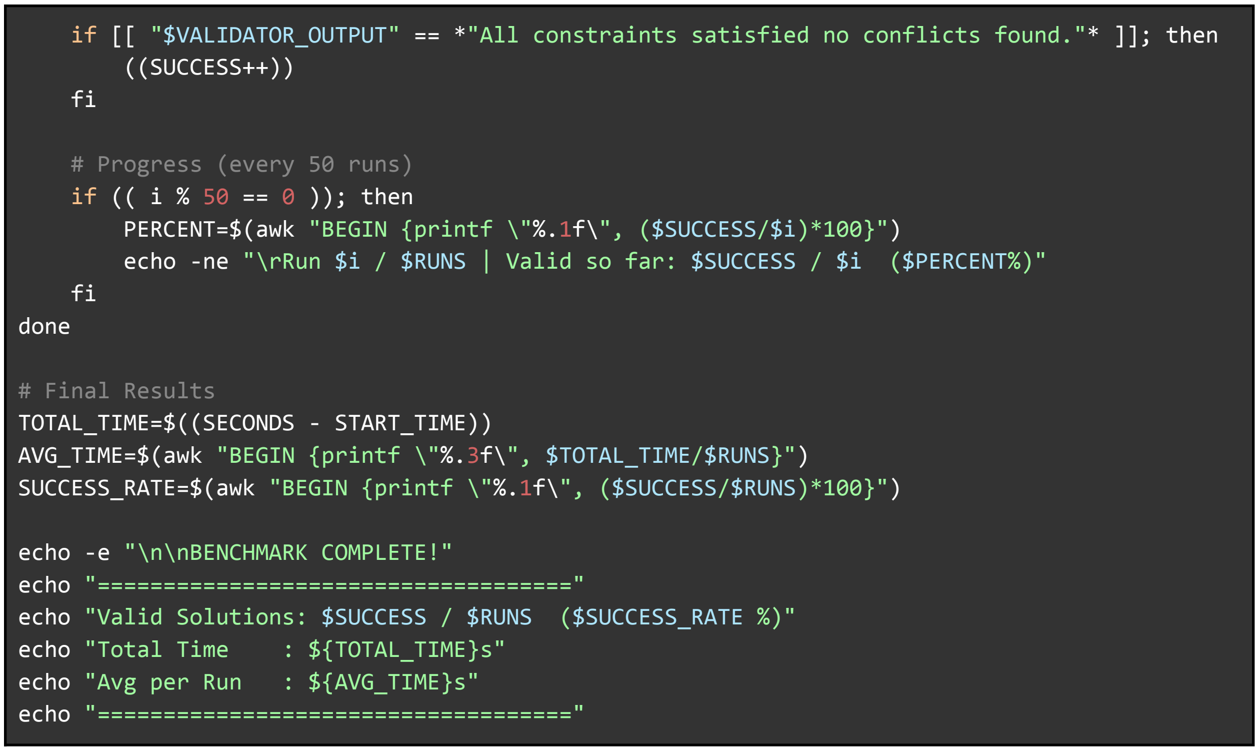
```powershell
param(
    [Parameter(Mandatory = $true, Position = 0)]
    [string]$mapf_algorithm
)

Write-Host "Starting MAPF Benchmark (1000 runs)"
Write-Host "===================================="

# Setup
if (!(Test-Path "out")) { New-Item -ItemType Directory -Name "out" -Force | Out-Null }
$success = 0
$stopwatch = [System.Diagnostics.Stopwatch]::StartNew()
$runs = 1000

pipenv run python mapf_graph_generator/graph_gen.py 3 2
for ($i = 1; $i -le $runs; $i++) {

    $numAgents = Get-Random -Minimum 3 -Maximum 16
    pipenv run python parallelization_validator/generate_input.py $numAgents $mapf_algorithm
--output in/input.toml
    pipenv run python3.14t mapf_solver.py --config in/input.toml --parallel

    $validatorOutput = pipenv run python parallelization_validator/validate_solution.py
out/solution.json 2>$null

    if ($validatorOutput -match "All constraints satisfied no conflicts found.") {
        $success++
    }

    # Progress (every 50 runs)
    if ($i % 50 -eq 0) {
        $percent = [math]::Round(($success / $i) * 100, 1)
        Write-Host "`rRun $i / $runs | Valid so far: $success / $i  ($percent%)" -NoNewline
    }
}

$stopwatch.Stop()
$totalTime = $stopwatch.Elapsed.TotalSeconds
$avgTime = $totalTime / $runs
$successRate = [math]::Round(($success / $runs) * 100, 1)

Write-Host "`n`nBENCHMARK COMPLETE!"
Write-Host "===================================="
Write-Host "Valid Solutions: $success / $runs  ($successRate %)"
Write-Host "Total Time    : $([math]::Round($totalTime, 1))s"
Write-Host "Avg per Run   : $([math]::Round($avgTime, 3))s"
Write-Host "===================================="
Write-Host "Last Solution: out/solution.json"
Write-Host "Rerun: .\mapf_benchmark.ps1"
```

*Figure 38:* mapf_benchmark.ps1

## 8.2 Testing Scripts for Linux and macOS

The testing scripts for Linux and macOS use Bash, and their purpose and workflow are the same as that of the PowerShell script. The script runs 1000 iterations, checks whether the search algorithm returns a valid solution, and updates the progress every 50 iterations.

```bash
#!/bin/bash
MAPF_ALGORITHM=$1
PARALLEL_ARG=$2

if [ -z "$MAPF_ALGORITHM" ]; then
    echo "Error: MAPF algorithm argument is required."
    echo "Usage: ./mapf_benchmark.sh <mapf_algorithm> [parallel]"
    exit 1
fi

PARALLEL_FLAG=""
if [[ "$PARALLEL_ARG" == "parallel" || "$PARALLEL_ARG" == "--parallel" || "$PARALLEL_ARG" ==
"true" ]]; then
    PARALLEL_FLAG="--parallel"
    echo "Parallelization: ENABLED"
else
    echo "Parallelization: DISABLED"
fi

echo "Starting MAPF Benchmark (1000 runs)"
echo "===================================="

# Setup
mkdir -p out
SUCCESS=0
RUNS=1000
START_TIME=$SECONDS

# Create graph for benchmarking
pipenv run python mapf_graph_generator/graph_gen.py 5 3

for ((i=1; i<=RUNS; i++)); do
    # Random number of agents between 3 and 15
    NUM_AGENTS=$(( ( RANDOM % 13 ) + 3 ))

    pipenv run python parallelization_validator/generate_input.py $NUM_AGENTS
$MAPF_ALGORITHM --output in/input.toml > /dev/null 2>&1
    pipenv run python3.14t mapf_solver.py --config in/input.toml $PARALLEL_FLAG > /dev/null
2>&1

    VALIDATOR_OUTPUT=$(pipenv run python parallelization_validator/validate_solution.py
out/solution.json 2>/dev/null)
```



```
    if [[ "$VALIDATOR_OUTPUT" == *"All constraints satisfied no conflicts found."* ]]; then
        ((SUCCESS++))
    fi

    # Progress (every 50 runs)
    if (( i % 50 == 0 )); then
        PERCENT=$(awk "BEGIN {printf \"%.1f\", ($SUCCESS/$i)*100}")
        echo -ne "\rRun $i / $RUNS | Valid so far: $SUCCESS / $i  ($PERCENT%)"
    fi
done

# Final Results
TOTAL_TIME=$((SECONDS - START_TIME))
AVG_TIME=$(awk "BEGIN {printf \"%.3f\", $TOTAL_TIME/$RUNS}")
SUCCESS_RATE=$(awk "BEGIN {printf \"%.1f\", ($SUCCESS/$RUNS)*100}")

echo -e "\n\nBENCHMARK COMPLETE!"
echo "===================================="
echo "Valid Solutions: $SUCCESS / $RUNS  ($SUCCESS_RATE %)"
echo "Total Time    : ${TOTAL_TIME}s"
echo "Avg per Run   : ${AVG_TIME}s"
echo "===================================="
```

*Figure 39:* mapf_benchmark.sh

# 9. Visualization

The visualizer is a tool that is designed to animate MAPF solutions on the graph. It provides an interactive interface for analyzing agent movements, detecting conflicts, and verifying solution validity in graphics.

## 9.1 Input Requirement

The visualizer accepts two types of JSON input files: graph and solution. The user can manually upload these files directly to the visualizer.

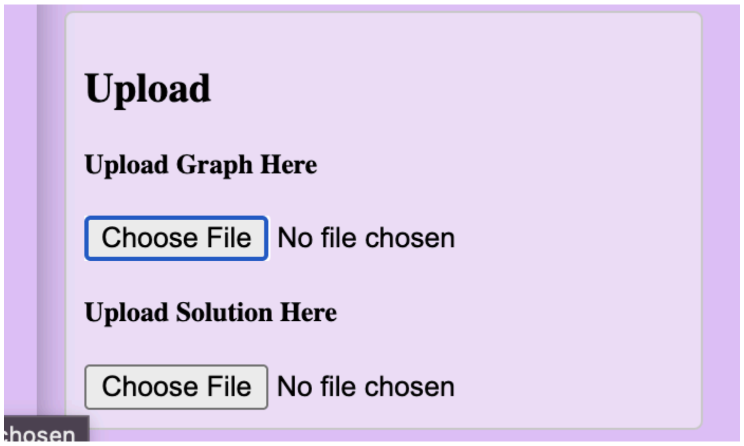
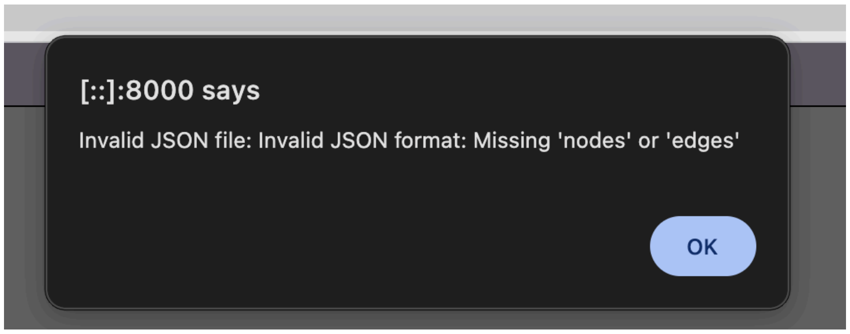


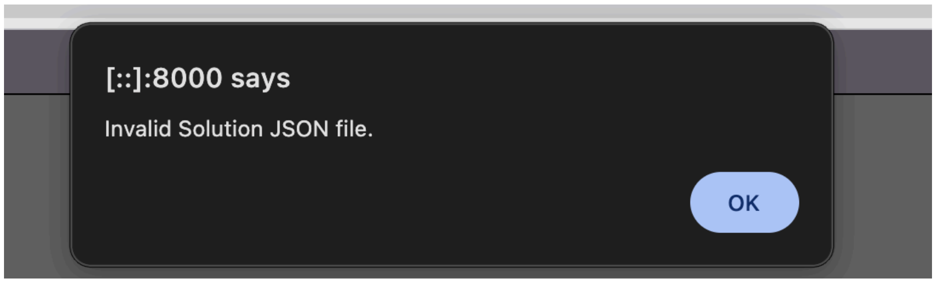
*Figure 40:* Section to upload JSON files

If the user misuploads the JSON files, such as uploading the solution.json to the graph section or uploading the graph.json to the solution section, the visualizer will alert to an invalid JSON file.


[::]:8000 says

Invalid JSON file: Invalid JSON format: Missing 'nodes' or 'edges'

*Figure 41:* Alert when an incorrect graph.json



[::]:8000 says

Invalid Solution JSON file.

*Figure 42:* Alert when an incorrect solution.json

After the user provides the correct graph.json file, the visualizer renders the graph with vertices and edges. When the user provides the correct solution.json file and the visualizer can parse it, the vertices occupied by agents will be coloured, and the vertices with the outline coloured will indicate the goal nodes to reach.

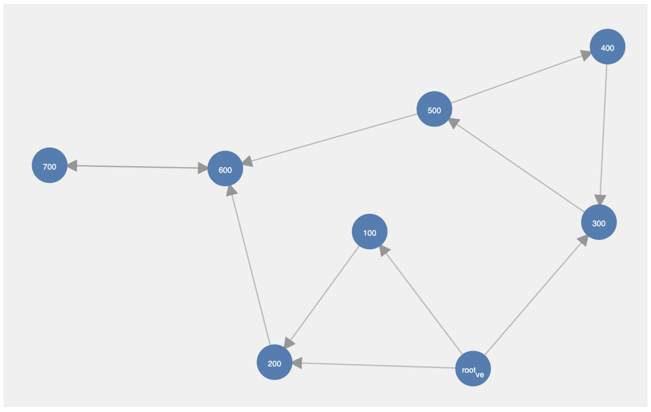


*Figure 43:* Rendered graph by the visualizer

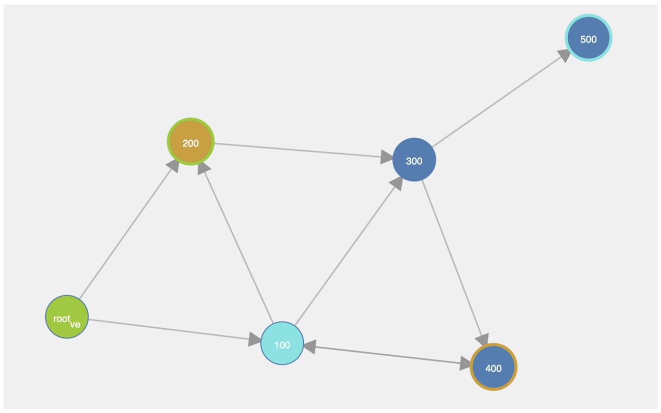


*Figure 44:* Rendered graph with the solution by the visualizer

## 9.2 Bug Fix

The first issue encountered was that the visualizer could not parse the solution.json file from the TOML graph file. The bug was that the solution.json from the TOML graph contained only the node id, not the label; however, the visualizer expected both the node id and the label, so it could not be parsed. This issue has been addressed and fixed by adding the function to create the label based on the node id.

Another bug is that when more vertices can be displayed within the frame, they extend beyond the frame, preventing the user from observing the solution process. This issue has been addressed by limiting the space to the visible area.

## 9.3 Additional Features

Initially, neither the output pipeline nor the visualizer supported the weighted edge. This feature has been added to the output pipeline and the visualizer to indicate whether the edge configuration includes an edge weight.

When previous students worked on the complex graph vertices, they used special characters to describe subscripts and superscripts. However, that implementation has a serious issue with the visualizer. The visualizer was unable to parse special characters, which caused problems for both the graph and the solution JSON files. Instead, the system has changed its label format to LaTeX, such as C^{i-j}_{i} and T^{i-j}_{j}. The complex graph generator will return the graph.json and the solution.json in LaTeX format, and the visualizer can now parse the LaTeX format into subscript and superscript.

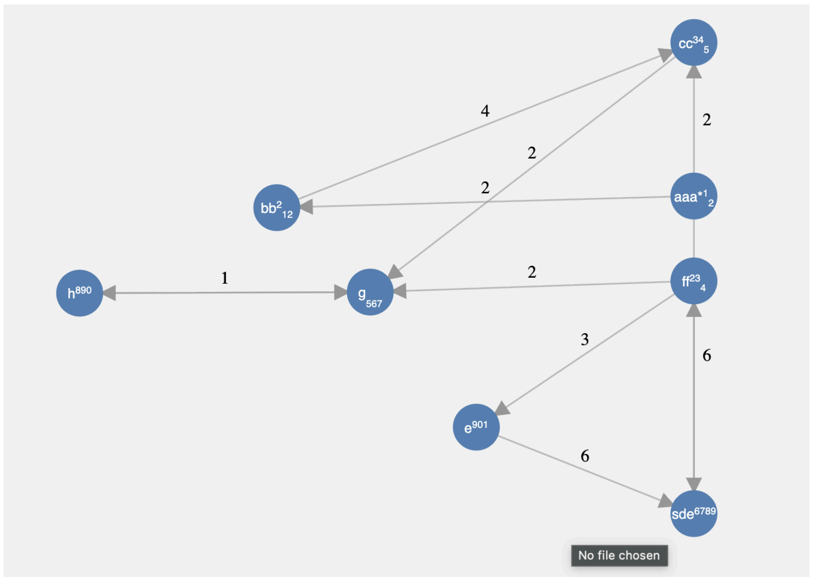


*Figure 45:* Rendered graph formatted with subscript and superscript and weighted edges

# 10. Conclusion

This project delivers a complete, extensible, and rigorously validated framework for solving Multi-Agent Path Finding (MAPF) problems in both sequential and parallel environments. Through a combination of algorithmic refinement, structural enhancements, and infrastructure development, the system advances the reliability, performance, and expressiveness of MAPF solutions beyond prior iterations of the codebase.

A significant portion of the effort focused on strengthening the two core algorithms, ICTS and CBS. ICTS received substantial performance improvements through memory optimizations, early-pruning strategies, and an entirely new parallel depth-partitioned search mechanism. CBS, on the other hand, required extensive debugging and revision of its conflict-handling logic, normalization procedures, and high-level search-loop invariants to ensure correctness under all constraint types. Both algorithms now operate robustly under the expanded set of supported constraints.

The constraint subsystem itself was substantially extended. Two new distance-based constraints (K-Hop and Radius Conflict) were implemented. Most notably, the complex mechanical constraint system (built on gears, sectors, and rotation propagation) required the introduction of new data structures, agent-behaviour modeling, and multi-stage consistency checks. These features allow the solver to reason about physically influenced movements far beyond standard MAPF formulations.

To support this advanced constraint model, a fully automated complex graph generator was created. This tool ensures consistent structure, correct gear placement, and scalable generation of test environments, enabling systematic experimentation and stress-testing without manual graph construction.

Parallelization forms another major contribution of the project. By leveraging Python 3.14's free-threaded runtime, ICTS and CBS both incorporate parallel execution paths that reduce runtimes while preserving correctness. A set of large-scale testing scripts—spanning Windows, Linux, and macOS—provides continuous verification of solution validity under randomized workloads, ensuring that parallelism does not introduce race conditions or inconsistent results.

Overall, this project establishes a unified, dependable, and performant MAPF system capable of handling complex environments, diverse constraint types, and multi-core execution. The combination of algorithmic refinement, structural engineering, and thorough validation creates a solid foundation for future extensions, whether they involve additional constraints, larger environments, heuristic improvements, or further optimization of parallel execution.