

FLEETBENCH: A MULTI-AGENT PATHFINDING EVALUATION WORKFLOW

A Thesis

Presented to the faculty of the Department of Mechanical Engineering
California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Mechanical Engineering

by

Thomas Wayne Christian Carlson

FALL
2023

© 2023

Thomas Wayne Christian Carlson

ALL RIGHTS RESERVED

FLEETBENCH: A MULTI-AGENT PATHFINDING EVALUATION WORKFLOW

A Thesis

by

Thomas Wayne Christian Carlson

Approved by:

_____, Committee Chair
Dr. Hong-Yue “Ray” Tang

_____, Second Reader
Neal Levine

Date

Student: Thomas Wayne Christian Carlson

I certify that this student has met the requirements for format contained in the University format manual, and this thesis is suitable for electronic submission to the library and credit is to be awarded for the thesis.

_____, Graduate Coordinator
Dr. Farshid Zabihian

Date

Department of Mechanical Engineering

Abstract

of

FLEETBENCH: A MULTI-AGENT PATHFINDING EVALUATION WORKFLOW

by

Thomas Wayne Christian Carlson

Many modern industries have and continue to benefit from the integration of robots into the workspace. As the capabilities and cost efficiency of robots improve, it is increasingly desirable to use robots as the backbone of service or production chains. One such application which has become a billion-dollar industry is warehouse management. However, the question of how to optimally manage the actions of robot fleets in such large and dynamic workspaces is not solved.

Review of the literature indicated that there are myriad approaches to generating plans for individual robots, with different advantages and disadvantages. Without a definite solution available, implementing engineers must therefore make educated guesses and run tests to identify the best solution for their application. This process is clouded by a number of difficulties which make approaching the task of testing algorithms arduous.

To mitigate these burdens, this thesis introduces a strategy which intends to be a generalizable design pattern for implementation of multi-agent problem simulations. This approach turns out to be very much like a state machine.

Two decoupled multi-agent algorithms, Windowed Hierarchical Cooperative A* and Token Passing with Task Swaps, are presented and implemented using the state machine approach in a completely original computer application named FleetBench.

Using the application, the two algorithms are put on an equal playing field and compete to produce best solutions for several representative test cases which are sufficiently different in nature to provide insight into the advantages of each approach. The test cases were developed rapidly with minimal effort and produce similar results to the existing research, affirming the implementation.

FleetBench is designed to be extensible, accepting additional algorithms. All source code is exposed to the reader and user alongside documentation regarding extension techniques. The addition of new algorithms to FleetBench over time would increase its value as an end-to-end multi-agent testing workflow.

_____, Committee Chair
Dr. Hong-Yue “Ray” Tang

Date

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

| | Page |
|--|------|
| Acknowledgements | vii |
| List of Tables | xiii |
| List of Figures | xiv |
| List of Algorithms | xv |
| List of Abbreviations | xvi |
| Chapter | |
| 1. BACKGROUND | 1 |
| Motivation | 5 |
| 2. PROBLEM DEFINITIONS AND SOLUTIONS | 11 |
| Defining Multi-Agent Problems | 12 |
| Basic Approaches | 16 |
| The A* Algorithm | 17 |
| Local Repair A* | 21 |
| Multi-Agent Approaches | 23 |
| Windowed Hierarchical Cooperative A* | 24 |
| Cooperative A* | 24 |
| Hierarchical Cooperative A* | 27 |
| Windowed Hierarchical Cooperative A* | 30 |
| Token Passing | 31 |
| Well-Formed Problems and Completeness Guarantees | 32 |

| | |
|---|----|
| Token Passing | 33 |
| Token Passing with Task Swaps | 37 |
| 3. GENERALIZABLE APPROACH | 40 |
| Common Behaviors | 41 |
| Simulation Definition..... | 43 |
| New Timestep | 44 |
| Task Management | 45 |
| Action Planning | 46 |
| Action Execution | 48 |
| End of Step..... | 49 |
| 4. FRAMEWORK APPLICATION | 51 |
| WHCA* | 52 |
| TPTS | 55 |
| 5. IMPLEMENTATION, TESTING, AND RESULTS | 62 |
| Design of Experiment | 66 |
| Case 1 | 67 |
| Case 2..... | 69 |
| Case 3..... | 70 |
| Results..... | 72 |
| Case 1 | 75 |
| Case 2..... | 77 |
| Case 3..... | 79 |

| | |
|--|-----|
| 6. CONCLUSION..... | 82 |
| Future Work | 83 |
| Appendix A: FleetBench Code Overview..... | 87 |
| Appendix B: State Machine Implementation..... | 89 |
| State Machine Definition and Execution | 89 |
| New Sim Step | 90 |
| Select Agent Step | 90 |
| Task Assignment | 91 |
| Select Action | 93 |
| Task Interaction..... | 94 |
| Plan Move | 95 |
| Pathfinding..... | 96 |
| Movement Execution | 97 |
| Checking the Agent Queue | 98 |
| End Simulation Step | 99 |
| End Simulation State..... | 101 |
| Mandatory Script Functions..... | 101 |
| Appendix C: Test Data Recreation..... | 103 |
| Map Files | 103 |
| Task Schedule Files..... | 103 |
| FleetBench Session Files | 104 |
| Appendix D: Map File Creation and Format | 105 |

| | |
|---|-----|
| Loading to FleetBench | 107 |
| Appendix E: Task Schedule Creation and Format | 109 |
| Appendix F: Extending FleetBench..... | 111 |
| Adding New Algorithms | 111 |
| Adding Configuration Options | 113 |
| Appendix G: Visualization Utility | 118 |
| Agent Objects..... | 120 |
| New | 121 |
| Delete | 122 |
| Clear | 122 |
| Move | 122 |
| Rotate | 123 |
| Highlight Objects | 123 |
| New | 124 |
| Delete | 124 |
| Clear | 125 |
| Move | 125 |
| Canvas Line Objects | 125 |
| New | 126 |
| Delete | 126 |
| Clear | 127 |
| Text Objects | 127 |

| | |
|-------------------------------------|-----|
| New | 128 |
| Delete | 128 |
| Clear | 129 |
| Appendix H: Conflict Resolver | 130 |
| References | 132 |

LIST OF TABLES

| | |
|--|-----|
| Table 1. FleetBench simulation definition options..... | 52 |
| Table 2. System configuration for case 1 experiments..... | 68 |
| Table 3. System configuration for case 2 experiments..... | 70 |
| Table 4. System configuration for case 3 experiments..... | 71 |
| Table 5. Results of experimentation on case 1. | 77 |
| Table 6. Results of experimentation on case 2. | 79 |
| Table 7. Results of experimentation on case 3. | 81 |
| Table 8: Locations of algorithm scripts in FleetBench. | 88 |
| Table 9: Pathfinder script required functions. | 101 |
| Table 10: Tasker script required functions. | 101 |
| Table 11: Mover script required functions. | 102 |
| Table 12: Manager script required functions. | 102 |
| Table 13: Test case map file access locations. | 103 |
| Table 14: Test case task schedule file access locations. | 103 |
| Table 15: FleetBench test case session file access locations..... | 104 |
| Table 16: Example task schedule file..... | 110 |
| Table 17: Script objects and input arguments. | 112 |
| Table 18: Configuration tabs and respective builder code. | 116 |
| Table 19: TkInter variable types..... | 117 |

LIST OF FIGURES

| | |
|--|-----|
| Figure 1: Two independent agent plans resulting in a collision..... | 4 |
| Figure 2: An agent seeks a path to a node using the A* algorithm..... | 9 |
| Figure 3: Two conflict types agents may experience..... | 15 |
| Figure 4: Three different measures of distances on a grid..... | 18 |
| Figure 5: The LRA* replanning procedure..... | 22 |
| Figure 6: Three MAPD instances, each well- or badly-formed..... | 33 |
| Figure 7: Graphical overview of the proposed state machine..... | 42 |
| Figure 8: Graphical overview of the Task Management state. | 46 |
| Figure 9: Graphical overview of the Action Planning state..... | 48 |
| Figure 10: Graphical overview of the Action Execution state..... | 49 |
| Figure 11: Graphical Overview of the End Step state. | 50 |
| Figure 12: Modification of the state machine for simultaneous task and path planning . | 58 |
| Figure 13: An image of the simulation window in FleetBench. | 64 |
| Figure 14: Screenshot of the map creation process in GraphRendering..... | 65 |
| Figure 15: System map for test case 1. | 67 |
| Figure 16: System map for test case 2. | 69 |
| Figure 17: System map for test case 3. | 70 |
| Figure 18: Data panel of a simulation in FleetBench. | 72 |
| Figure 19: Sequential moves by two naïve agents..... | 80 |
| Figure 20: Map creation process in GraphRendering..... | 105 |
| Figure 21: An example of a small map created in GraphRendering..... | 106 |
| Figure 22: Task schedule generation window in FleetBench..... | 109 |
| Figure 23: Screenshot of the labeled UI elements. | 115 |
| Figure 24: Screenshot of the generated optionMenu elements..... | 116 |
| Figure 25: Visualization of the search process in FleetBench. | 118 |
| Figure 26: Screenshot of the FleetBench canvas and information bar..... | 120 |
| Figure 27: An agent rendered in FleetBench. | 121 |
| Figure 28: A sequence of highlight objects in FleetBench. | 123 |
| Figure 29: A canvas line object rendered in FleetBench. | 126 |
| Figure 30: A piece of text rendered in the northwest corner of a tile in FleetBench. | 127 |

LIST OF ALGORITHMS

| | |
|--|----|
| Algorithm 1. The A* algorithm | 20 |
| Algorithm 2. Neighbors function for LRA* | 23 |
| Algorithm 3. The Cooperative A* algorithm | 26 |
| Algorithm 4. The Reverse-Resumable A* algorithm | 29 |
| Algorithm 5. Finished function used in WHCA* | 30 |
| Algorithm 6. Finding nearest tasks to an agent for Token Passing..... | 35 |
| Algorithm 7. The Token Passing algorithm | 36 |
| Algorithm 8. The Token Passing with Task Swaps algorithm | 38 |
| Algorithm 9. WHCA* state machine representation | 54 |
| Algorithm 10. TPTS state machine representation | 55 |
| Algorithm 11. GetTask from TPTS state machine representation | 56 |
| Algorithm 12. Detangled TPTS state machine representation..... | 59 |

LIST OF ABBREVIATIONS

| Abbreviation | Definition |
|---------------------|--|
| CA | Cooperative A* |
| CBS | Conflict-Based Search |
| GUI | Guided User Interface |
| HCA | Hierarchical Cooperative A* |
| LRA | Local Repair A* |
| MAPD | Multi-Agent Pickup and Delivery |
| MAPF | Multi-Agent Pathfinding |
| PRIMAL | Pathfinding via Reinforcement and Imitation Multi-Agent Learning |
| RRA | Reverse Resumable A* |
| SAPF | Single-Agent Pathfinding |
| TP | Token Passing |
| TPTS | Token Passing with Task Swaps |
| UI | User Interface |
| WHCA | Windowed Hierarchical Cooperative A* |

1. Background

Mobile robotics is a rapidly maturing field with a wide range of applications including industrial settings [1], space exploration [2], and search and rescue [3]. Significant advancements have been made regarding robot capabilities, manufacturing costs, and safety which drive further growth and innovation in the fields which integrate these increasingly powerful robots.

One setting has garnered particular attention in recent years, expanding into a multi-billion-dollar industry: mobile warehouse robots. State of the art research and development in this area has been ongoing for some time at companies such as Boston Dynamics [4] and Agility Robotics [5], who seek to push the capability envelope of humanoid robots with their Atlas and Digit robots. Currently, however, the industry is dominated by simpler robots which are purpose-built for shifting specific loads in fulfillment centers by companies such as Amazon Robotics and Alibaba. Amazon fields over 750,000 robots in its fulfillment facilities, forming a complex problem of robot and human cooperation which must be solved continuously to deliver approximately 1 billion packages per year [6]. Such processes are typically viewed as the way of the future for warehousing projects [7] and Amazon, while the largest company, is far from the only group pursuing implementation of automated systems in fulfillment centers [8].

A key problem faced in warehouse robotics is that of finding appropriate paths and trajectories for robots (also called agents) in densely populated spaces with high throughput

requirements. Similar instances of this problem can be found in a great number of fields including videogames [9], search problems including rescue and evacuation [3], and air-traffic management [10]. This problem has received significant attention from both robotics and artificial intelligence researchers, becoming well characterized but remaining definitively unsolved [3].

The problem of guiding a fleet of agents to a set of individual targets in an efficient and collision-free manner through an environment is called the Multi-Agent Pathfinding (MAPF) problem [11]. Each instance of the problem is defined by a number of agents, each with their own goal, located in a defined space with known movement costs. The solution therefore optimizes the cost function of movement within the system with the constraint that there be no collisions and that all agents finish on their target locations. Typically, the cost in these instances is time, although it is sufficiently trivial to augment the cost functions with fuel costs or other abstract measures of performance during implementation.

The MAPF problem is considered to be a “single shot” problem and solution. Agents will move to their targets and simply remain there so long as no collisions are caused by doing so. This falls short of being analogous to real-world warehousing applications, where it is desirable for agents to chain together sequences of optimal movement for various tasks such as pickup and delivery. Therefore an extension of the MAPF problem to make it “lifelong” is developed, called the Multi-Agent Pickup and Delivery (MAPD) problem [12].

The MAPD problem augments agent targets to be tasks with a pickup location and a delivery location. The path of an agent completing its objective now involves movement to the delivery location via a path which includes the pickup location. New tasks are also periodically introduced to the system, and assignments to agents are made dynamically. Success is measured when all tasks in the system are marked as complete, with the key indicator of performance being the service time per task or batch of tasks. New challenges are introduced in the form of avoiding deadlock and cycling behaviors, where agents repeatedly move through the same motions without advancing toward their goals. Resiliency against disturbances is also of elevated concern [13]. This formulation of the problem more accurately captures challenges in real-world applications but requires different approaches and applies different constraints that make MAPF-solving algorithms ill-suited to the task without adaptation.

Single-agent pathfinding problems where there are no other moving objects in the space are trivial; a solution is known and returns provably optimal paths with known time and memory limits via the famous A* algorithm [14]. This approach cannot be naively extended for multiple agents moving simultaneously in the system space. Agents need to use knowledge of each other's positions and intent to avoid collisions during plan execution. One such collision case is shown in **Figure 1**. More complex strategies are needed.

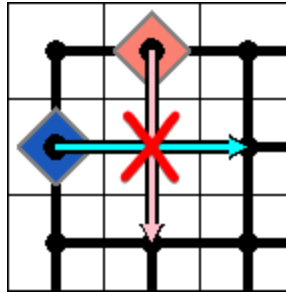


Figure 1: Two independent agent plans resulting in a collision, rendered in FleetBench.

Solutions to the MAPF and MAPD problems generally fall into one of two categories: centralized, where plans for agents are created all at once, or decoupled, where agents form their plans sequentially, avoiding collisions with the plans of other agents in the system.

Centralized approaches such as CBS [15], PRIMAL [16], and Push and Swap [17] generally produce solutions by searching a huge variety of motion options at every step. Thoroughly searching the valid state space where every agent represents a degree of freedom for possible state changes represents an immense endeavor, but the end result is a guarantee that the path is collision-free (complete) and the fastest possible solution (optimal) [17]. However, this approach is computationally intense, even for small scenarios, and is simply not scalable to larger applications [3]. If computation time begins to exceed action time systems will begin to lag, resulting in cascading underperformance as additional objectives are introduced. System resiliency is also an important area in which centralized systems struggle because unexpected disruptions cause enormous replanning operations [13]. Advancements in this area often involve trying to prune the search space, reducing search time [17].

Decoupled approaches such as Windowed Hierarchical Cooperative A* [18] and Token Passing [12] tend to experience the opposite; while the search space is greatly reduced by focusing on a single agent at a time—making the problem tractable even at large agent counts—the solution found cannot be guaranteed to be optimal, or even complete unless certain preconditions are met [12]. For example, an agent can plan a path which cuts off another agent’s access to its target, forcing excessive wait times or resulting in a failure to solve the problem. The lack of such guarantees makes the algorithms unreliable, and there often are not strong criteria which can identify problem instances that are solvable among those which are not [19]. As a result, it is difficult to justify the implementation of decoupled solutions in real-world systems without thorough testing and modification, especially when the system is one which carries a high failure cost such as air traffic control [20].

Motivation

Several factors contribute to difficulties in studying these algorithms, whether one is testing an implementation of an algorithm which exists in the research or designing a novel approach to the problem. These obstacles may prove especially difficult for those not already well-versed in the field and may act as a deterrent for industry implementation, hobbyist engagement, and prospective researchers.

By and large, practical implementations of algorithms found in research are not instantly accessible—the work must be requested from authors or reproduced by the reader.

In the latter case, the reader will need expertise in programming in order to implement both their own test cases and the algorithm itself. Further difficulties arise when attempting to produce visualizations. In both cases, there is a lack of standardization—algorithms could be implemented in any language (though many are created in C++), on any test case, with any style of input and output of data formatting. Available source code, particularly regarding what data structures are used, is not always well-documented which adds further difficulty to the process of adjusting an algorithm.

The presentation of data in research often falls into evaluation of categories such as *makespan* (maximum arrival time), *flowtime* (total time loss), or a count of the number of successfully completed tasks over a defined period of time. While these data are useful in developing general notions of success, further optimization is likely to lie in dealing with edge cases which an algorithm handles poorly. Such situations may be washed out in a longitudinal study spanning hundreds or thousands of instances. In these cases, it is useful to have access to historical data in the problem solution in order to investigate the interactions which produce the problem, and therefore develop ideas about restrictions or potential augmentations to the algorithm. This will necessitate implementation of various statistical tracking methods across every selected algorithm, further consuming the researcher's time.

When benchmarking the performance of an algorithm it is important that similar conditions are used in order to approach evaluating test results. The underlying characteristics of any multi-agent problem directly drive the performance of all algorithms.

Minor changes produce cascading effects which result in extremely different outcomes. For decoupled systems, varying the agent activity order has a pronounced effect on the solutions found. With no clear predictor that can identify when variation of agent order would matter, analysis is restricted to tedious variation of parameters. Topology of the system creates very different opportunities for solutions, such as in the case of BIBOX, which requires that the graphical abstraction of the system space be bi-connected [21]. The order in which tasks are inserted to the problem has significant downstream effects. Furthermore, the way in which they are assigned leaves open another axis for optimization. For example, assigning tasks in proximity order to the current agent may be efficient for the individual agent, but the system as a whole may suffer from the increased traffic in a region that other agents need to pass through. Small adjustments produce very different results which may obfuscate the behavior of algorithms in other situations. Due to non-standardized use of data structures, input and output formats, and use of different programming languages, it is difficult to assure equal playing fields for each algorithm.

Some strides have been made in these areas, defining a set of benchmark test maps (and map types) as well as strategies for generating tasks [11]. However, there is a reliance on large test sizes and pseudorandom generation which may not conform to real-world use-cases. There are also a number of publicly accessible code repositories which contain implementations of one or more algorithms, again in the author's preferred programming language and style. Some repositories are implemented in the Godot Videogame Development engine [22], while others have implementations in raw C++ [23], [24] or

Python [25], [26]. In many cases, it is not clear how the function of such implementations could be readily extended while maintaining a standard procession of logic across each implementation. Development of test cases is not done via any guided process, inviting errors at each step. Extension of the programs to include additional algorithms is not a straightforward process and often times it is unclear whether any behaviors are shared across implementations of each algorithm in the first place. The collection of empirical data in such works requires further modification much of the time.

In an effort to bridge these gaps between theoretical knowledge and practical test implementation, this thesis presents a framework through which decoupled MAPF and MAPD algorithms may be implemented in a manner which separates the different axes of optimization and allows for the insertion of various real-world constraints. The implementation strategy is designed to be maintainable and flexible, while anchoring key facets present in most algorithms. With this approach, algorithms may be implemented into one coherent system, ensuring ease of test and iteration. The resulting technique is presented in Chapter 3. Two families of decoupled multi-agent algorithms are introduced in Chapter 2, and adapted using the strategy to produce a behavior map in Chapter 4. This represents the primary intent of the work. Layers of abstraction are used to generalize the problem, so that with careful planning most situations may be represented and solved.

As proof of concept, Chapter 5 presents a pair of programs which implement the strategy for the selected algorithms. The first application, called GraphRendering, is a utility for generating the system maps over which a fleet of agents operates. It is a modified

version of Tile Basic, an application distributed under the GNU General Public License [27]. Changes to the source code of the original application are listed in Appendix D. The second application, called FleetBench, provides a graphical user interface (GUI) which enables intuitive implementation of test cases and data collection for the end user, while exposing the internals in an explainable and modular fashion for developers and researchers who are creating algorithms. Visualization and tabulation of results is provided via library-like functions which may be used from within algorithm scripts. An example of the visualization is provided in **Figure 2**. Configurable options are implemented to allow approximation of real-world constraints where possible. A state history is made accessible to aid in identification of problematic states.

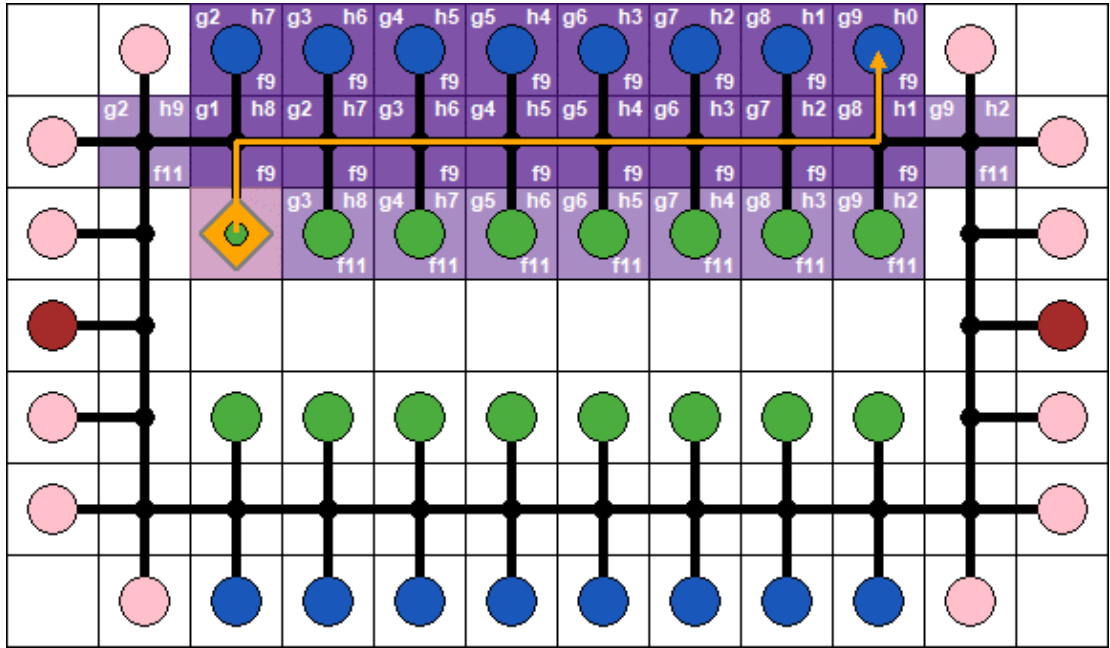


Figure 2: An agent seeks a path to a node using the A* algorithm, rendered in FleetBench. The map represents an MAPD situation, where blue nodes are for pickup, and green are for deposit. Pink and brown nodes are designated resting points.

Together, these two applications create a workflow for a user to rapidly test algorithms on a variety of test cases with various designable restrictions. The user will be able to quickly tweak aspects of their test case such as the map layout, task ordering, and restrictions on agents when the user chooses to investigate performance bottlenecks. Data for three test cases of interest are presented in Chapter 5 as evidence that the approach and resulting application are useful for analysis.

The applications are provided as-is in a public repository. Reference documentation and a guide for extending functionality are located in the appendices of this manuscript.

2. Problem Definitions and Solutions

Applications which benefit from implementing solutions to the MAPF and MAPD problems are often large, dynamic, continuous spaces throughout which many complex interactions must occur to realize a desired outcome. In real-world applications there are many concerns to be aware of which tend to cloud the central problem of finding optimal paths. Examples include robot turning radius, positional tolerances, momentum, acceleration, carry weight, and so on.

Fundamentally, solutions in MAPF and MAPD problems are designed to find collision-free paths for all agents while maximizing some metric which relates to the performance of the system. To begin developing a solution in such spaces it is best to restrict the number of confounding issues, starting from a basis of the most critical and well-defined issues. By implementing a few key assumptions, the problem space is reduced significantly. The two most useful changes to make to the problem statement involve discretization of spatial and temporal concerns.

With a discretized space, collisions occur at points in space, rather than needing to be evaluated on the whole spectrum of motion. This reduces the description and evaluation of motion to simple ideas of position rather than needing to consider speed, orientation, or turning radius.

With a discrete description of time, agent actions can be evaluated on a simple basis of the number of timesteps required to complete the action. This allows evaluations of the

system state to be made on a consistent basis without concerns about synchronization and timing needing to be settled.

Of course, this also means that solutions found in discrete spaces cannot be blindly applied to the real-world, continuous, situation. However, with a method for approaching the process established, it becomes easier to augment the solution to handle kinematic restrictions and respect tolerances in motion [28].

With the above restrictions, it becomes possible to express the problem spatially in terms of a mathematics construction called a *graph*. In doing so, the problem can be formally stated and solutions from other areas of mathematics may be applied to the model. Finding the shortest path, which is a subset of graph traversal problems, is a well-studied problem with many useful results and techniques. A graph is therefore the standard model used to solve many problems in pathfinding, including MAPF and MAPD problems.

Defining Multi-Agent Problems

A graph is composed of two primary objects: *vertices* and *edges*. Abstractly, a vertex (also called a node) is a representation of some *thing* which is possible to reach via some *process*. It could be a state, a location, or an object.

An edge (sometimes called a link) is used to represent a connection between two vertices. Such a relationship could be the path walked by a person to reach location A from location B, the set of actions taken in a system to reach state A from state B, or the machining process used to create a part from stock material. An edge can be said to be directed if it is only possible to move from vertex A to vertex B along the edge, and not

from B to A. If travel in either direction is allowed the edge is said to be undirected. An edge may also have a weight, or cost, associated with it, possibly representing the fuel expense of driving from city A to city B along a particular route.

These two components form the non-linear structure called a *graph* and are typically expressed as a graph G composed of a set of n vertices V and a set of edges E . The edge set is built from a subset of vertices which are connected. An edge is represented by an unordered pair of connected vertices. For this work, the graph is constructed with the following definition:

$$G = (V, E),$$

where

- $V = \{v_1, v_2, \dots, v_n\}$ is a finite set of nodes,
- $E \subseteq \{(v_i, v_j) | (v_i, v_j) \in V^2 \text{ and } v_i \neq v_j\}$ is a finite set of edges.

This construction disallows the existence of multiple edges connecting a node, which would be called a *multigraph*. Further, the representation of edges as a set rather than an ordered pair means that an edge is agnostic to ideas of direction. If the edge was an ordered pair, the graph would be considered *directed*. The set-builder notation for the edge set also contains an assertion that an edge cannot connect a node to itself. Without this restriction, the graph is called a *graph with loops*. The above definition, used throughout this work, is therefore termed a *simple undirected graph without loops*.

The exclusion of graphs with loops is adhered to for simplicity of the graph's structure, but the restriction is not necessarily required. In fact, the ability of an agent to

travel from its current location to the same location (but one timestep into the future, effectively a waiting move) will be critical in allowing other agents to maneuver with minimal disruption.

A few additional components are needed to fully define an MAPF or MAPD problem. A set of k agents A contains information about how many agents are in the problem's system and their individual properties. The task set O contains j tasks which have not yet been completed, driving the solving of the problem. The task set must be finite in order for a solution to exist but is not a fixed quantity. Tasks may be freely added and removed from the set in an on-line fashion, simulating an infinite set. For the purpose of formality, two mapping functions λ_A and λ_O are used to define the positions of all agents and tasks. In this way, a multi-agent problem M is defined:

$$M = (G, A, O, \lambda_A, \lambda_T),$$

where

- $G = (V, E)$ is a simple undirect graph,
- $A = \{a_1, a_2, \dots, a_k\}$ is a finite set of agents,
- $O = \{o_1, o_2, \dots, o_j\}$ is a finite set of tasks,
- $\lambda_A: A \rightarrow V$ is a function mapping agents to vertices,
- $\lambda_O: O \rightarrow V$ is a function mapping objectives to vertices.

With a formal expression of the problem, it becomes possible to formally express conflicts. **Figure 3** shows the two basest cases of conflict possible in the construction of the MAPF and MAPD problems used in this work: a vertex conflict (left) and a swapping

conflict (right). Each expresses the idea that some resource (in this case, space) is being used by more than one agent, which in the real-world system corresponds to an undesired collision.

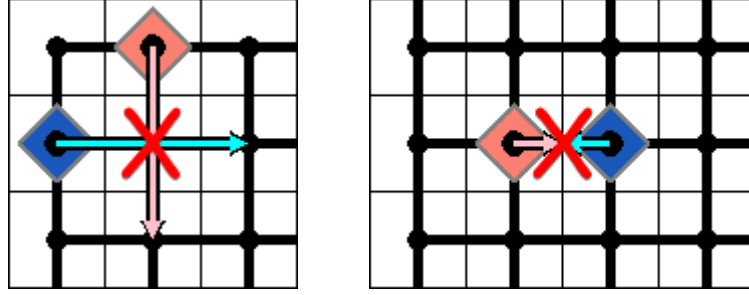


Figure 3: Two conflict types agents may experience when moving within a multi-agent problem instance, rendered in FleetBench. Left: Vertex conflict. Right: Edge conflict.

Further conflicts exist but in general there is no unified requirement for certain conflicts to be observed throughout the literature [11].

Formally expressing these conflicts in the context of MAPF and MAPD systems is done by representing a moment in time where two agents will occupy the same space. In the typical construction each agent has a series of positions it intends to inhabit, separated by timesteps, called a plan. The space-time position of agent a_i located at some node v at any timestep t is given as $a_i(v, t)$.

A vertex conflict arises when two agents attempt to occupy the same node v at the same time:

$$a_1(v, t) = a_2(v, t)$$

Swapping conflicts occur when two agents attempt to use the same edge E to reach new nodes. Because an edge connects only two nodes, the agents are attempting to swap node positions and would collide on the way to their planned locations:

$$a_1(x, t + 1) = a_2(x, t) \text{ and } a_2(x, t + 1) = a_1(x, t)$$

With these restrictions in place, it is now possible to evaluate which nodes and edges are available for use at any given time. Actions which would cause vertex or swapping conflicts are marked off as impossible during search and traversal.

In MAPF problems, objectives or tasks are expressed as the need for an agent (which may or may not be a particular agent from the set of agents in the system) to be in a specific location. The MAPD problem is an augmentation of the MAPF statement to include a two-part task. This task includes a demand for an agent to be present at some location for a “pickup” action, followed by a need for the same agent to be in a location afterwards to perform a “delivery” action. Therefore, any agent in an MAPF or MAPD problem with an assigned task will have some target node to reach, ideally in the fewest number of timesteps possible.

With these definitions and restrictions in place, it is possible to begin seeking optimal paths.

Basic Approaches

The single-agent pathfinding (SAPF) problem has been thoroughly investigated in literature for some time. The SAPF problem shares similar concepts with the MAPF or MAPD problems, in that a path must be found for each agent in the system (in this case, just one). However, the results do not generalize to the multi-agent case—single-agent solutions make no effort to avoid other agents, by definition. There are sufficiently many similarities that solutions to the SAPF problem can be adapted as a starting point for

addressing multi-agent problems. This section presents the techniques which serve as primitives for the algorithms which will be used in Chapters 4 and 5.

The A* Algorithm

One of the most useful results in the study of optimal graph traversal is the A* algorithm [14]. This algorithm can be viewed as an extension of Dijkstra's algorithm, which finds the shortest path between nodes (in terms of edge costs) via an exhaustive search. A* augments this approach with the use of a heuristic that guides the search by attaching an idea of proximity to the goal to each node explored, called a node's *h-Score*. Choosing the next nodes to be explored using the best h-Score guides the algorithm to explore paths which approach the goal, until they cannot be further advanced. So long as certain qualities about the heuristic being used are guaranteed, A* returns provably optimal paths without over-processing the graph [29]. The primary requirement for the heuristic function to do so is that it does not overestimate the distance to the goal.

Several simple and intuitive heuristics exist. Dijkstra's algorithm is functionally equivalent to the A* algorithm when the heuristic always returns 0, making no effort to distinguish best nodes. The Manhattan distance is a good model for 4-neighbor connected spaces, while the Chebyshev distance is well-suited for 8-neighbor connected spaces. In a continuous space, the Euclidean distance may be used. A visual comparison of what notions of distance look like in these three geometries is provided in **Figure 4**.

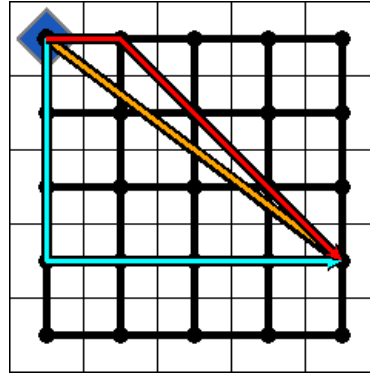


Figure 4: Three different measures of distances on a grid rendered in FleetBench: Euclidean (orange), Manhattan (cyan), and Chebyshev (red).

Each heuristic provides some quantitative value describing the distance between two points in their respective geometries and each is considered to be valid for use in the A* algorithm, provided they do not overestimate distance. For example, while the Euclidean distance may be employed in a 4-neighbor connected space (a straight line to the goal will always have less or equal length to any Manhattan distance), the Manhattan distance is not a usable heuristic in continuous spaces as it will overestimate the distance to the goal. In this work, the graph is assumed to describe a system with a Manhattan geometry, where a node may have up to four neighbors. Further, the edge cost for each movement is assumed to be the same, such that an agent exerts the same effort in moving any direction.

Two other scoring mechanisms are employed to inform the search. A node's *g-Score* is the length of the best path found to reach a node from the starting position. At any point during the search this can be used to find the best path from start to the node. However, it may not be the best possible path in the whole graph, so a node's *g-Score* may be updated over the course of the algorithm as improvements to the path are found. A node's *f-Score*

is the sum of a node's *g-Score* and *h-Score*, representing the estimate of the total length of a path which passes through the node on its way to the goal node.

There are five subroutines called during the execution of A* which are defined and identified here for ease of reference:

- ***HDistance***(v_1, v_2) returns an estimate of the distance between nodes v_1 and v_2 , determined by using an appropriate heuristic function.
- ***Neighbors***(v) returns the set of nodes which are connected to node v via an edge. This function may be augmented to include the node v in the returned values if an agent may be interested in waiting in the same position.
- ***Cost***(v_1, v_2) returns the edge cost of a movement from v_1 to v_2 . With the assumption that all edge costs are the same this function does not perform any calculation, but it may easily be augmented.
- ***IsGoal***(v_1, v_2) returns a Boolean *True* value if the input node v_1 is the goal node v_2 , and Boolean *False* otherwise.
- ***BuildPath***(v_1, v_2) is used when the goal node is explored by the algorithm. Using saved *g-Score* data, the optimal path to v_1 from v_2 is reconstructed, returning the ideal path to the goal.

The operating procedure for the A* algorithm is described in **Algorithm 1**. Lines 2-6 describe the setup for the algorithm. The search is primarily driven by the *open* set, which acts as a priority queue containing all nodes which are available for exploration, ordered in favor of the least remaining heuristic distance from the node to the goal. The

first node added to this set is the starting position, whose *g-Score* is clearly zero. At this point in time, *g-Scores* for other nodes are unknown and assumed to be infinite. The *f-Score* for any node is simply the calculated heuristic distance, which for all nodes other than the start node is unknown and also assumed to be infinite.

Algorithm 1. The A* algorithm

```

1: function AStar(start, goal)
2:   closed  $\leftarrow \emptyset$ 
3:   open  $\leftarrow \{start\}$ 
4:    $g[*] \leftarrow \infty$ ;  $g[start] \leftarrow 0$ 
5:    $f[*] \leftarrow \infty$ ;  $f[start] \leftarrow \text{HDistance}(start, goal)$ 
6:   parent[*]  $\leftarrow \emptyset$ 
7:   while open  $\neq \emptyset$ :
8:     current  $\leftarrow v \in open$  with minimal  $f[v]$ 
9:     open  $\leftarrow open \setminus \{current\}$ 
10:    if IsGoal(current, goal):
11:      path  $\leftarrow \text{BuildPath}(current, start)$ 
12:      return path
13:    for neighbor  $\in \text{Neighbors}(current)$ :
14:       $g_{estimate} \leftarrow g[current] + \text{Cost}(current, neighbor)$ 
15:      if  $g_{estimate} < g[neighbor]$ :
16:        parent[neighbor]  $\leftarrow current$ 
17:         $g[neighbor] \leftarrow g_{estimate}$ 
18:         $f[neighbor] \leftarrow g_{estimate} + \text{HDistance}(neighbor, goal)$ 
19:        if neighbor not in open:
20:          open  $\cup \{neighbor\}$ 
21:  return failure

```

So long as there is at least one node in the open set, the search has not exhausted potential options, and continues with the process of removing a new node from the open set and evaluating it on lines 8-9. Should the removed node turn out to be the goal node, the algorithm recursively checks its memory of parent nodes, reconstructing the path with the lowest *g-Score* to the node and returning it in lines 10-12. Otherwise, the algorithm

obtains a set composed of all the nodes which share an edge with the current node, which are called neighbors.

On lines 13-20, each neighbor's *g-Score* is calculated by summing the *g-Score* of the current node with the edge cost of traveling from the current node to the neighbor. If this cost is lower than the currently stored *g-Score* for the neighbor node, then an improved path from the start to the neighbor node has been found, and the current node is recorded as the best way to reach the neighbor node in the *parent* datastructure. The neighbor node updates its *f-Score* by adding the new *g-Score* together with the heuristic distance from the goal. If the neighbor is not already included in the open set for exploration, then it is added to the open set and the process repeats. This algorithm can only end in success, with an optimal path found (line 12), or failure in the case that no such path is possible because all accessible nodes have been explored and the goal was not reached (line 21).

This algorithm forms the basis for all pathfinding operations described in this chapter. There are alternatives which offer different properties and advantages such as D* Lite, which maintains a memory of paths and adapts to changes in the graph [30]. Ultimately, however, the choice of path planning algorithm underlying the multi-agent problem solving algorithm is a degree of freedom for the designer and does not significantly impact the procedures outlined in Chapter 3.

Local Repair A*

Naively planning paths for all agents in a system quickly turns out to be insufficient in many cases. For example, two agents which must pass each other could very easily plan

intersecting paths which result in a vertex or swap conflict as shown in **Figure 5**, left and center.

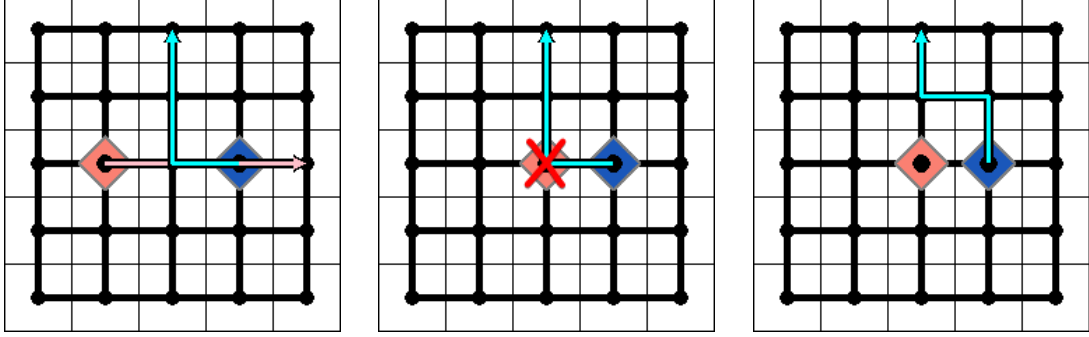


Figure 5: The LRA* replanning procedure. On the left, two agents plan optimistic paths to their goals. A collision is detected, and the lower priority agent (blue) plans a new route, avoiding the immediate neighbor to its left.

A simple approach, which is widely used in the videogame industry, is to allow A* to execute until a collision is inevitable [18]. At the timestep where a collision is detected, the agent whose movement is blocked because of the collision instead re-plans its path by running a new instance of the A* pathfinding algorithm. The starting node is the current location, while the goal node remains the same. To avoid the collision, the *Neighbors* function is modified so that the agent is unable to consider nodes in its immediate proximity which would lead to a collision. An example of this procedure is provided in **Figure 5**. To execute the new search, the pathfinding algorithm needs access to the system state at the current timestep t to evaluate the position of other agents in the system: $M_t = (G_t, A_t, O_t, \lambda_{A_t}, \lambda_{T_t})$. The updated function is described in **Algorithm 2**.

Algorithm 2. Neighbors function for LRA*

```

1: function Neighbors-LRA(node, M)
2:   result  $\leftarrow \emptyset$ 
3:   for n  $\in$  Neighbors(node):
4:     if n  $\notin$  {Position( $\forall a; a \in A$ )}:
5:       result  $\leftarrow$  result  $\cup$  {n}
6:   return result

```

This approach is not very powerful for a host of reasons. The replanning procedure means that in densely populated graphs agents frequently undergo recalculation of their entire paths. The low amount of foresight given by the simple *Neighbors* function augmentation also frequently results in cycling or jostling behaviors in which agents shuffle back and forth attempting to find routes around each other. As the number of agents in the bottleneck increases, the situation grows to take arbitrarily long to resolve, never guaranteeing a solution will be found [18]. The algorithm is provably unable to resolve bottlenecks at all in certain graph conditions and provides no predictive power to prevent its use in such cases. These behaviors are immediately obvious in experimentation and quickly prove to yield insufficient results. A greater degree of inter-agent cooperation is needed.

Multi-Agent Approaches

A* serves as a powerful tool for finding paths in any given instance of a graph traversal problem but, as discussed in the previous section, cannot be naively used to solve problems in which multiple agents share the same space.

This section presents the strategies employed by two families of algorithms which attempt to solve multi-agent problems. The first is called Windowed Hierarchical Cooperative A* (WHCA*), which combines the principles of three algorithms into one and is presented in [18]. The second family is built on a principle called Token Passing (TP). Its authors also present a set of criteria for determining whether an instance of an MAPD problem is guaranteed to be solvable [12].

These algorithms will be adapted via the procedure laid out in Chapter 3 as examples of the methodology so as to provide points of comparison. Their usage within the framework laid out in Chapter 3 to produce experimental data in Chapter 5 will reveal each approach's advantages and restrictions.

Windowed Hierarchical Cooperative A*

Windowed Hierarchical Cooperative A* (WHCA*) is a combination of three algorithms presented by David Silver in his 2005 paper “Cooperative Pathfinding” [18]. Building directly from the A* and LRA* implementations, Silver develops additional augmentations which enable agents to be aware of each other's intentions, better analyze the space in which they travel via altering the heuristic function of A*, and increase the flexibility and computational speed of the pathfinding algorithm via the addition of a windowing function.

Cooperative A*

Cooperative A* (CA*) is the first algorithm which achieves a degree of direct agent cooperation. By implementing a reservation table which is stored at some central authority

which all agents may access, agents are able to retrieve knowledge of each other's plans. The reservation table is a representation of the graph structure which is augmented with a third dimension: time. Agent paths through the system are represented by a series of n-tuples with increasing time depth composed from the position and the time at which an agent will occupy that position. By including the time dimension the reservation table has predictive power up to the time horizon of the system which can be accessed by any agent intending to plan a path.

To make use of this shared information, the A* algorithm must be modified in several ways:

- Found paths need to be logged into the reservation table.
- Some notion of time depth must be included in the searching strategy, increasing for each new node explored. To do this, *BuildPath* must be expanded to include the time in its characterization of a node.
- The *Neighbors* subroutine must be modified to only return nodes which are unreserved in the next time step. It should also return the agent's current node, if it is not reserved in the next time step (this is the *wait* action). This new routine is named *FreeNeighbors*.
- In the case of a replan, the old plan must be removed from the reservation table so as not to falsely impact the performance of new searches. This procedure is implemented in a new routine: *Replan*.

With these changes implemented, the algorithm achieves a degree of cooperation among agents such that no agent should be able to find a path which intersects another agent. This may mean that an agent is unable to find any path to the goal. In such cases, the agent must be allowed to not act, which exposes a weakness in the algorithm that can be felt in densely populated graphs.

Should an agent fail to find a path, it has no default behavior to fall back on. Its inability to create a plan may interfere with the plans of other agents. This behavior also indicates a high degree of ordering sensitivity: the first plans have the least restrictions, while the last plans may be impossible to create. Naively implementing a similar solution to LRA* for agents which are disrupted by another agent's failure to find a path, the *Replan* routine is introduced.

The CA* process is described by **Algorithm 3**.

Algorithm 3. The Cooperative A* algorithm

```

1: function CASTar(start, goal)
2:   closed  $\leftarrow \emptyset$ 
3:   reserved  $\leftarrow \emptyset$ 
4:   open  $\leftarrow \{start\}$ ; time  $\leftarrow 0$ 
5:   g[*]  $\leftarrow \infty$ ; g[start]  $\leftarrow 0$ 
6:   f[*]  $\leftarrow \infty$ ; f[start]  $\leftarrow \text{HDistance}(start, goal)$ 
7:   parent[*]  $\leftarrow \emptyset$ 
8:   while open  $\neq \emptyset$ :
9:     current, time  $\leftarrow (v, t) \in open$  with minimal f[v]
10:    open  $\leftarrow open \setminus \{current\}$ 
11:    if IsGoal(current, goal):
12:      path  $\leftarrow \text{BuildPath}(current, start, time)$ 
13:      reserved  $\leftarrow \{(node, time) | (\forall (node, time); node \in path)\}$ 
14:      return path
15:    for neighbor  $\in \text{FreeNeighbors}(current, reserved, time)$ :
16:      gestimate  $\leftarrow g[current] + \text{Cost}(current, neighbor)$ 
17:      if gestimate < g[neighbor]:

```

```

18:         parent[neighbor, time + 1] ← current
19:         g[neighbor, time + 1] ← gestimate
20:         f[neighbor, time + 1] ← gestimate + HDistance(neighbor, goal)
21:         if neighbor not in open:
22:             open ∪ {neighbor, time + 1}
23:         return failure
24: function FreeNeighbors(node, reserved, time)
25:     result ← ∅
26:     for n ∈ Neighbors(node):
27:         if (n, time + 1) ∉ reserved:
28:             result ← result ∪ {n}
29:     return result
30: function Replan(start, goal, reserved, oldPath)
31:     reserved ← reserved \ {∀(node, time) ∈ oldPath}
32:     newPath ← CASTar(start, goal)
33:     return newPath

```

Hierarchical Cooperative A*

Silver notes that the choice of heuristic may cause problems in more challenging environments, where searches generate complicated paths which are vulnerable to being replanned when the dynamics of the system cause interruptions. He proposes the use of simple hierarchy to represent the search space abstractly, reducing the difficulty of the search operation. This new algorithm, in combination with an optimization in the form of a reversed-direction A* search which is kept in memory, constitutes the Hierarchical Cooperative A* (HCA*) algorithm.

The key to the hierarchy is that it is an abstraction of the graph's state which does not include other agents or obstacles. The distance from starting position to the goal is therefore a perfect estimate. Provided that the chosen heuristic does not inherently

overestimate the geometry of the space this clearly cannot overestimate the distance to the goal, making it a sufficient heuristic to assure optimality in A* searches.

The proposed hierarchy replaces the ***HDistance*** subroutine with a search in the reverse direction, from goal to current position, which ignores the presence of agents and obstacles in the graph. Employing an A* search in place of the standard ***HDistance*** subroutine means that the distance from the currently evaluated node to the goal is a more precise estimate. This increase in accuracy reduces the number of search operations needed by better guiding the agent's primary CA* search toward the goal.

Further, the results of this reversed search are kept in memory as an optimization to avoid performance hits during operation due to the extra searches being performed. As an agent advances further along its plan, the hierarchical search data remains relevant because it is advancing further into already known data. Because the motion of agents does not impact the hierarchical search the search data is never invalidated by agent activity. If the properties of the graph change, then the stored data cannot be assured to be accurate. This is a known weakness of A* which other algorithms in the research overcome [31]. In this work the map is considered static, which mitigates the need for lifelong methods.

Taken together, these modifications to the heuristic function of the CA* search are termed Reverse Resumable A*. The procedure is outlined in **Algorithm 4**, serving as a drop-in replacement for the ***HDistance*** routine in CA* (**Algorithm 3**). In this case, the goal is still the target node for the agent, but the reversed direction of the search means that the starting position for calculating distances is the goal node. The search progresses toward the agent's current position. When it is completed, the *g-Score* of the search from target to

current position is equivalent to the hierarchical distance from agent to target and is returned.

Before executing the search procedure in lines 9-22, the algorithm checks to see if it has already stored the appropriate *g-Score* in its closed set of nodes. If the data is already available, the algorithm immediately returns the distance (lines 7-8). Declarations of the scoring, open, and closed sets must be moved out of the function in order to remain in memory after calls to the routine, as shown in lines 1-5.

Algorithm 4. The Reverse-Resumable A* algorithm

```

1:  $closed \leftarrow \emptyset$ 
2:  $open \leftarrow \{goal\}$ 
3:  $g[*] \leftarrow \infty; g[goal] \leftarrow 0$ 
4:  $f[*] \leftarrow \infty; f[goal] \leftarrow HDistance(position, goal)$ 
5:  $parent[*] \leftarrow \emptyset$ 
6: function RRA( $position, goal$ )
7:   if  $position \in closed$ :
8:     return  $g[position]$ 
9:   while  $open \neq \emptyset$ :
10:     $current \leftarrow v \in open$  with minimal  $f[v]$ 
11:     $open \leftarrow open \setminus \{current\}$ 
12:    if  $IsGoal(current, position)$ :
13:      return  $g[position]$ 
14:    for  $neighbor \in Neighbors(current)$ :
15:       $g_{estimate} \leftarrow g[current] + Cost(current, neighbor)$ 
16:      if  $g_{estimate} < g[neighbor]$ :
17:         $parent[neighbor] \leftarrow current$ 
18:         $g[neighbor] \leftarrow g_{estimate}$ 
19:         $f[neighbor] \leftarrow g_{estimate} + HDistance(neighbor, goal)$ 
20:        if  $neighbor$  not in  $open$ :
21:           $open \cup \{neighbor\}$ 
22: return failure

```

Windowed Hierarchical Cooperative A*

A final alteration is made to solve a set of practical concerns with the previous algorithms. By imposing a windowing restriction that prevents the algorithm from searching too deeply the time spent searching for a path is potentially reduced dramatically, time is not wasted evaluating potential collisions which may not occur in a more loosely scheduled system, and the sensitivity of the algorithm to agent ordering is significantly reduced [18]. This final change, taken together with the changes found in CA* and HCA*, comprises the Windowed Hierarchical Cooperative A* (WHCA*) algorithm.

The windowing restriction is an alteration of the *IsGoal* function which compares the current search depth to the window size parameter w , whose value is to be selected by the designer. If the search depth is equal to or greater than the size of the window, the search is terminated in the same fashion as if the agent had found a complete route to the goal node. The replacement algorithm, called *Finished-WHCA*, is shown in **Algorithm 5**.

Algorithm 5. Finished function used in WHCA*

- 1: **function** Finished-WHCA(*current*, *goal*, *time*, *w*)
 - 2: **return** $time \geq w \vee current \equiv goal$
-

Because of the HCA* implementation providing a heuristic which guides the search in the direction of the optimal path toward the goal, forward progress is still assured so long as a path to the goal is possible. This effectively means that only for search depths less than the size of the window the agent is navigating the base state of the graph where it considers other agents plans. Beyond the window the path is equivalent to the hierarchical abstraction from HCA*. Only the base graph search path is logged into the reservation

table of CA* so as to avoid unnecessarily obstructing other agents with the theoretical path given by the abstraction—which is not guaranteed to be followed.

Notably, this implementation of WHCA* is designed to solve the MAPF problem. When an agent reaches its goal, it is assumed that its objective is to remain on the goal as much as possible, moving only to allow another agent to reach its own goal. However, the warehousing environment is much more analogous to the MAPD problem. This leaves the process vulnerable to the same problems present in LRA* because it is essentially the same at its core: new plans are made to avoid agents on a regular basis without deeply considering the disruptive impacts of doing so or what should occur when an agent has no possible actions to take according to the reservation table. To implement WHCA* in an MAPD scenario, additional care must be taken to process such cases in a manner which does not terminate the simulation of the problem in an error state. The general collision resolution strategy which was developed in this work is described in Appendix H.

Token Passing

A second family of approaches tackles the MAPD problem directly by making assertions about the class of problems which are solvable and therefore being provably complete, but not necessarily optimal. These algorithms employ a Token Passing strategy in which the token is a block of memory which represents combined knowledge of the state of the system at the current time step and into the future. By passing the token to agents one at a time, each agent can find and plan paths in a decoupled fashion before passing the modified token back to the central authority.

Well-Formed Problems and Completeness Guarantees

The authors argue and prove that for a certain class of MAPD problems, solutions may be guaranteed [12]. Because of this, any algorithm which makes proper use of the conditions defining MAPD problems of this type can be said to be *complete* such that it will not fail to eventually find a solution to the problem.

The concept of an endpoint is introduced. An endpoint is defined as a node in which an agent could freely rest until the end of the time horizon. This amounts to an extension of the reservation table which reserves a node for an agent for all known timesteps in the future. This definition must be applied to pickup nodes, delivery nodes, and designated locations for agent parking in order to assure solutions can be found. Three sets of endpoints are defined:

- $V_{\text{endpoints}}$, which contains all nodes satisfying the endpoint definition.
- V_{task} , which contains all pickup and delivery nodes.
- V_{ntask} , which contains all endpoints which are not pickup or delivery nodes.

According to the authors, to guarantee the existence of a solution for a given MAPD problem, several prerequisites must be met:

1. The number of tasks in the system is finite. This does not preclude the addition of tasks in an on-line fashion, merely that the act of searching for a task does not take arbitrarily long.
2. V_{ntask} must contain at least as many endpoints as there are agents in the system.

3. For any two endpoints, a path in the graph exists which does not require passing over another endpoint.

MAPD problems satisfying these requirements are considered “well-formed” and solutions are guaranteed when using certain algorithms, two of which are presented in the next sections. Optimality of solutions is not guaranteed, as greedier algorithms may find faster solutions while sacrificing the completeness guarantee. **Figure 6** presents three instances of an MAPD problem. The instance on the left shows a well-formed problem: there are two agents, two non-task endpoints, and no path to any endpoint requires passing over another endpoint. The middle instance is not well-formed as there are fewer non-task endpoints than there are agents, violating the second requirement. The instance on the right is not well-formed because the path from any endpoint in the top row to an endpoint in the bottom row is required to pass through the endpoint in the middle.

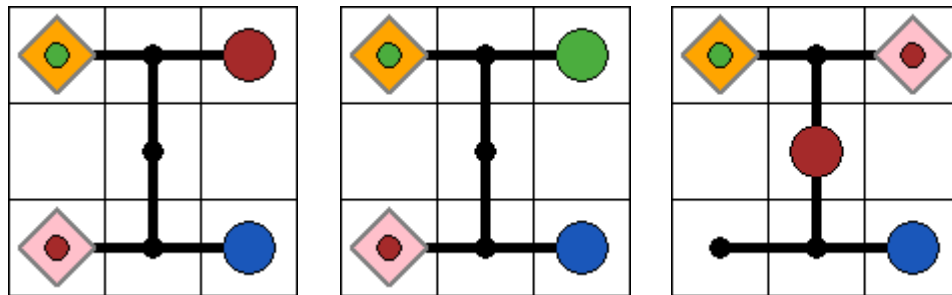


Figure 6: Three MAPD instances, each well- or badly-formed. Green circles indicate delivery endpoints. Blue circles indicate pickup endpoints. Brown nodes indicate non-task endpoints. Two agents, shown as diamonds, are in the problem space.

Token Passing

Token Passing (TP) is very similar to CA* in that it uses reservations to govern the set of nodes which are accessible to the agent seeking a path. However, using the endpoint

definition, the reservation table must now reserve endpoints which either contain an agent or will contain an agent for all timesteps into the future. The authors prove that an agent is only able to plan paths ending in an endpoint, so the final node in any agent’s path is not allowable into the path of any agent’s search operations. This imposes a restriction on the set of tasks to which an agent can be assigned: no task whose pickup node \mathbf{p} or delivery node \mathbf{d} are the end of another agent’s path in the token may be considered a valid assignment. The resulting set is called \mathbf{O}_{viable} .

Unlike WHCA*, TP takes advantage of a precompute step which executes before the act of solving the problem begins. This is only possible on maps which remain static during the solving process, else the information from the precompute step would become unreliable. During the preprocessing step distances and paths from every node in the graph to each endpoint are found. Because an agent operating in this algorithm will only ever find paths toward an endpoint, this data may also be used in place of the *HDistance* routine for notions of distance from an agent’s goal. For large graphs this may take a significant amount of time but need only be done once, yielding results which are reusable by the algorithm for the lifetime of the problem. This shortest-path data is stored in the problem data \mathbf{M} as δ .

The primary benefit of this preprocessing is that it becomes trivial to select the task with the nearest pickup location. Finding “best” tasks has the effect of significantly reducing the amount of travel time before an agent is performing useful work. The act of finding this task is given as a named subroutine, *BestTask*, in **Algorithm 6**. By using the

stored data from the preprocessing step, this process is made computationally efficient. This information cannot be used to quickly generate the path an agent will take while executing the task, however, because the agent must still be conscious of other agents and avoid collisions.

Algorithm 6. Finding nearest tasks to an agent for Token Passing

```

1: function BestTask(node, viableTasks)
2:   task  $\leftarrow \operatorname{argmin}_{o_j \in O_{viable}} \delta(\text{node}, p_j)$ 
3:   return task

```

Pathfinding operations in TP are classed as one of two types; a search for a path to a task endpoint, called ***Path1***, or a search for a path to a non-task endpoint, called ***Path2***. If the agent is content to remain in its current position then the trivial path is found using ***Stay***, and the agent waits in its current position if able. ***Path2*** is executed when the agent is not able to complete any task, while also unable to remain in its current position due to the path of another agent. In either case, the pathfinding is done in a manner identical to CA*, using the augmented reservation table which respects an agent's intent to rest forever in a location when it reaches the end of its planned path. If it is ever impossible to find a path, the agent will use its ability to remain in place, waiting until a path can be found for its assigned task.

The complete TP algorithm is described in **Algorithm 7**. Unlike other algorithms presented in this chapter, TP is not designed exclusively to find paths through the system state. It also performs operations on the set of tasks in the system before seeking paths. A named routine, called ***Preprocess***, is introduced to represent the precomputation step,

which returns all information necessary to define \mathbf{M} , including all computed distances (line 1). From then on, TP is executed continuously, using *Update* to add new tasks to the task set (lines 3-4). It first assigns the best task in the system to each agent, removing the task from the task set and planning an optimal path (lines 5-11). If there is no assignable task, an agent should be allowed to rest in place so long as its current position is not the endpoint of another task in the system to avoid future blockages (lines 12-13). Otherwise, if the agent obstructs another agent's plan, the agent should navigate to a non-task endpoint to make space for other agents (lines 14-15). At the end of the loop, the simulation advances by executing agent plans and incrementing the system timestep (line 16).

Algorithm 7. The Token Passing algorithm

```

1:  $M \leftarrow \text{Preprocess}(\text{MAPD})$ 
2:  $\text{reserved} \leftarrow \emptyset$ 
3: while true:
4:    $M \leftarrow \text{Update}(M)$ 
5:   for  $\forall a_i \in A$  with no assignment:
6:      $O' \leftarrow \{o_j \in O \mid \text{no } \text{path} \in \text{reserved} \text{ ends in } p_j \text{ or } d_j\}$ 
7:     if  $O' \neq \emptyset$ :
8:        $\text{task} \leftarrow \text{BestTask}(\text{Position}(a_i), O')$ 
9:       Assign  $a_i$  to  $\text{task}$ 
10:       $O \leftarrow O \setminus \text{task}$ 
11:       $\text{Path1}(a_i, \text{task})$ 
12:    else if  $\{o_j \in O \mid \text{Position}(a_i) \in \{p_j, d_j\}\} = \emptyset$ :
13:       $\text{Stay}(a_i)$ 
14:    else:
15:       $\text{Path2}(a_i)$ 
16:  All agents execute plans;  $\text{time} \leftarrow \text{time} + 1$ 
17: function  $\text{Path1}(\text{agent}, o_j)$ 
18:   if  $\text{agent}$  reached  $p_j$ :
19:      $\text{CAStar}(\text{Position}(\text{agent}), d_j)$ 
20:   else:
21:      $\text{CAStar}(\text{Position}(\text{agent}), p_j)$ 

```

```

22: function Path2(agent)
23:   node  $\leftarrow \operatorname{argmin}_{v \in V_{ntask}} \delta(\operatorname{Position}(\operatorname{agent}), v)$ 
24:   CAGStar(Position(agent), node)
25: function Stay(agent)
26:   node  $\leftarrow \operatorname{Position}(\operatorname{agent})$ 
27:   reserved  $\leftarrow \{\operatorname{agent}(\operatorname{node}, \operatorname{time}), \operatorname{agent}(\operatorname{node}, \operatorname{time} + 1)\}$ 

```

Token Passing with Task Swaps

TP is simple and efficient in many cases but shows an algorithmic inefficiency in the way tasks are assigned: a task which may be completed much more quickly by an agent later in the priority queue may be claimed by an agent which would complete the task more slowly.

It is possible to further optimize the selection of tasks by enabling agents to exchange tasks when the time to complete the task is reduced in doing so. To preserve the pickup and delivery analogy, agents may only swap task assignments before an agent has interacted with the pickup portion of the task. Before that occurs, a task is considered to be “unexecuted”. Afterwards, the task is being “executed” and can no longer be handed off to another agent. This procedure is implemented in Token Passing with Task Swaps (TPTS).

The primary concern in expressing this process as a single algorithm is the recursive nature in which task swaps must occur. An agent \mathbf{a}_1 may be better suited to complete task \mathbf{o}_1 than \mathbf{a}_2 , but if \mathbf{a}_2 is not able to make its way to a free endpoint from its current position, the task swap should fail and \mathbf{a}_2 should be allowed to continue with its previously planned actions. This process is represented by the *GetTask* procedure in **Algorithm 8**.

Algorithm 8. The Token Passing with Task Swaps algorithm

```

1:  $M \leftarrow \text{Preprocess}(MAPD)$ 
2:  $reserved \leftarrow \emptyset$ 
3: while true:
4:    $M \leftarrow \text{Update}(M)$ 
5:   for  $\forall a_i \in A$  with no assignment:
6:      $\text{GetTask}(a_i)$ 
7:   All agents execute plans;  $time \leftarrow time + 1$ 
8: function  $\text{GetTask}(agent)$ 
9:    $O' \leftarrow \{o_j \in O \mid \text{no } path \in reserved \text{ ends in } p_j \text{ or } d_j\}$ 
10:  while  $O' \neq \emptyset$ :
11:     $task \leftarrow \text{BestTask}(\text{Position}(agent), O')$ 
12:     $O' \leftarrow O' \setminus task$ 
13:    if no agent assigned to  $task$ :
14:      Assign  $agent$  to  $task$ 
15:       $\text{Path1}(agent, task)$ 
16:      return true
17:    else:
18:       $oldState \leftarrow (assignments, reserved, M)$ 
19:       $a_i' \leftarrow$  agent assigned to  $task$ 
20:       $path' \leftarrow \text{Path}(a_i')$ 
21:      Unassign  $a_i'$  from  $task$ ; Remove  $path'$  from  $reserved$ 
22:       $path \leftarrow \text{Path1}(agent, task)$ 
23:      if  $|path| < |path'|$ :
24:         $success \leftarrow \text{GetTask}(a_i')$ 
25:        if success:
26:          return true
27:        else:
28:           $(assignments, reserved, M) \leftarrow oldState$ 
29:      if  $\text{Position}(agent) \notin V_{endpoints}$ :
30:         $path \leftarrow \text{Path2}(agent)$ 
31:        if  $path \neq \emptyset$ :
32:          return true
33:      else:
34:        if  $\{o_j \in O \mid \text{Position}(agent) \in \{p_j, d_j\}\} = \emptyset$ :
35:          Stay( $agent$ )
36:        else:
37:           $\text{Path2}(agent)$ 
38:        return true
39:  return false

```

Once again, the data found by preprocessing the graph before starting to solve the MAPD problem is reused to supply distance information to the *BestTask* and *HDistance* routines (lines 1-2). TPTS is then executed continuously, seeking assignments for free agents using *GetTask* (lines 5-6). As before, a subset of viable tasks is taken from the set of all tasks, evaluated based on the reservations of other agents in the system (line 9). One by one, tasks in this subset are evaluated and assigned to agents if they do not have an assigned agent, or an attempt is made to swap task assignments between the current agent and the one currently assigned to the task (lines 10-26). Assignments are made assuming that the task swap will succeed, however the information before the swap occurs must be stored in case the swap fails. If the current agent would outpace previously assigned agent on the way to the pickup node, then the swap takes place by recursively calling *GetTask* until no more tasks can be assigned to agents in the recursion. This can end when all tasks in the viable task set are assigned (lines 32, 38) or when an agent is offered a task with no assigned agent (line 16). If either of these occur, then the recursion on line 24 resolves until the primordial *GetTask* call returns. At this point the swaps are successful and a new valid system state has been reached where the agents involved in swapping have the best assignments they can have for the current state.

So long as the requirements for well-formedness are met, the authors of TP and TPTS guarantee that each algorithm solves all MAPD instances [12].

3. Generalizable Approach

The collection of data for any studied algorithm inevitably involves simulation over a defined multi-agent problem space. As there is little agreement in the literature regarding fundamental choices in the design of the simulation space, it can be difficult to determine whether results from an algorithm are generalizable [3]. For instance, the WHCA* algorithm presented in [18] is tested on a randomly generated grid, with a random distribution of agents, and randomly selected task points. TPTS, on the other hand, is tested on problems which mimic the layout of a storage facility, with long corridors through which agents must avoid each other to reach their destinations [12]. Some standard test cases used in MAPF problems are offered as possible reference benchmarks, but there is little agreement on the usage and limited applicability of results from these cases [11].

As a result, those seeking to use multi-agent pathfinding algorithms in their own work must implement the algorithm and design test cases which best represent their use case. An engineer wishing to compare several algorithms may find a need for a high level of knowledge in several programming languages, the skill to modify existing code, and the ability to script the generation of test cases.

In an attempt to ease this knowledge burden and reduce the development time requirement, this chapter presents a general approach to the implementation of algorithms. The approach presented here is a procedure which keeps the system driving the simulation coherent and easy to configure for testing. Results of the implementation and testing of

algorithms from Chapter 2 using this strategy are presented in Chapter 5 to demonstrate its utility.

Common Behaviors

By identifying a set of behaviors which must be common to all algorithms attempting to solve multi-agent problems the process can be reframed as a set of behaviors taken when certain conditions are met. The implementation of an algorithm is a problem which can then be reduced to identifying when certain behaviors occur during the lifetime of the algorithm. Decomposing the algorithm in this way provides clarity of function, modularity of implementation, and ease of adaptation for future experiments.

Separating behaviors in the algorithm in this manner requires careful consideration of what multi-agent algorithms are meant to do at the basest level:

- Agents are assigned tasks.
- Agents must take actions which work toward completion of assigned tasks.
- Agents must not collide.
- The system seeks to minimize the cost of reaching the success state.

As an example, in the case of TPTS it is easy to see that there are provisions laid out for each of these desired behaviors. The algorithm efficiently finds paths for agents by using the optimal A* search. During the search a reservation table is employed which avoids collisions. Agents are able to eventually find paths to their goals, which are assigned in an optimized fashion using proximity and task swaps.

By anchoring these behaviors and isolating the portions of the algorithm that enact them, the process of executing the algorithm can be abstracted as an implementation of a finite state machine. This well-studied concept in programming offers a concrete method for implementing the logical processes involved in a multi-agent system's progression. It promotes the desired modularity and extensibility while presenting a simplified programming interface. The proposed state machine diagram is presented in **Figure 7**.

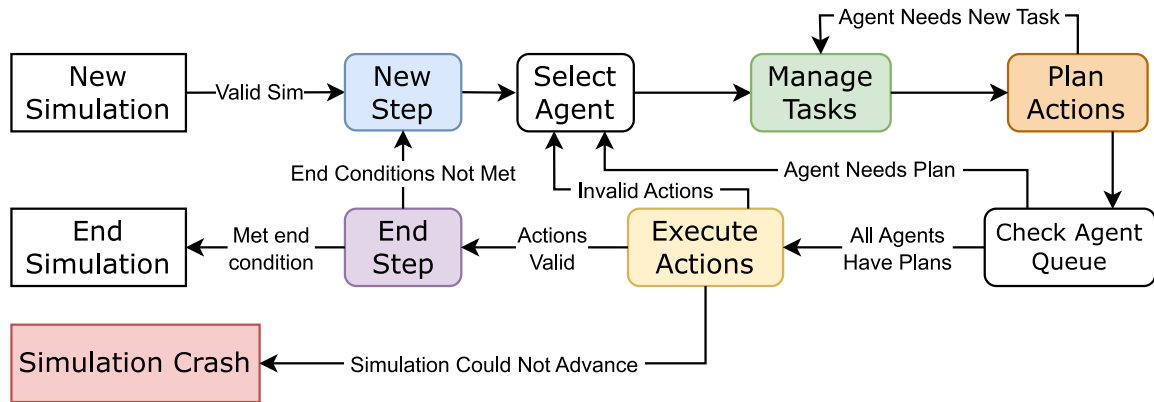


Figure 7: Graphical overview of the proposed state machine.

Routines designed to handle collision resolution, task assignment, and pathfinding are given their own states and can, with sufficient care, be implemented in a manner which meshes well with other algorithms. For instance, the application engineer may be able to implement an optimization in pathfinding heuristics into an algorithm which contains its own optimization for task assignment.

Being only a strategy for implementation, there are no imposed requirements to enable this process in terms of the programming language or the techniques employed within the individual states. This strategy does not preclude the use of any particular

algorithm which performs these functions. If the state machine is implemented with a provision for an algorithm to request a specific state, additional logical branches in the execution of the algorithm are trivial to add, further extending the functionality of the system. FleetBench is a novel application which is driven by this strategy in its execution and implementation of the algorithms described in Chapter 2; WHCA*, TPTS, and their ancestors.

The following sections provide brief overviews of the behaviors which are found in each section of the state diagram. The programmatic implementation of the state machine in Python is left to discussion in Appendix B.

Simulation Definition

Systems which can be represented in this manner are complex and dynamic, presenting many opportunities to make impactful decisions. Before the simulation begins, there are a number of choices to be made which act as defining “rules” for the simulation. Examples include:

- How many actions can an agent take per timestep?
- Does interacting with a task endpoint consume a timestep?
- Does rotation have a cost?
- Do agents experience faults during operation?
- How are tasks added to the system?
- How and when should the simulation be considered solved, if ever?

These types of restrictions apply globally to the simulation and must be respected throughout its lifetime. These rules must be configured during the simulation's setup state before other operations begin. These configurations must be available at all times in the simulation to inform the logic of the algorithms.

This state is also an appropriate time for algorithms to execute any routines which preprocess the graph, as in the case of TP and TPTS. If any such routines fail to execute, some kind of logic must be implemented. For example, the TPTS algorithm comes with notions of what a well-formed MAPD problem is. If the simulation is run on an MAPD problem which does not meet these conditions, guarantees about completeness are revoked. In such cases it may be preferable to warn the user or abort the simulation entirely.

New Timestep

At each new timestep there is an opportunity for the system to be updated with new information, informing the behaviors taken during the timestep which is being simulated. Typically this new information will be composed of new tasks, either generated on the fly or as part of a predefined schedule to be released at a particular timestep. Other events such as agent breakdowns, changes in operating strategy, or the introduction of additional agents to the system could also occur here.

In a more pragmatic sense, this state is also a good place for handling various programmatic concerns such as resetting the states of iterables or managing priority queues.

Task Management

Interactions with the task set minimally consist of two operations: task generation and task assignment. Task assignment is the act of designating a particular agent as the executor of a particular task. Task generation enters a new task to the set, whether via generating a completely new task or introducing a pre-defined task according to some task schedule supplied in the simulation definition step.

To maintain the analogy of the system to a real-world application, an external authority should manage task-defining processes. A warehouse system would require knowledge of an item's location and destination, while an air traffic control system may need to enforce timing constraints by restricting the availability of "tasks" to certain timesteps. In these cases, tasks need to be admitted to the system in an online fashion, which is left up to the engineer. For the testing process, it is likely sufficient to implement a custom generator or use a predefined list of tasks.

An overview of the proposed actions enacted during the Task Management state is provided in **Figure 8**.

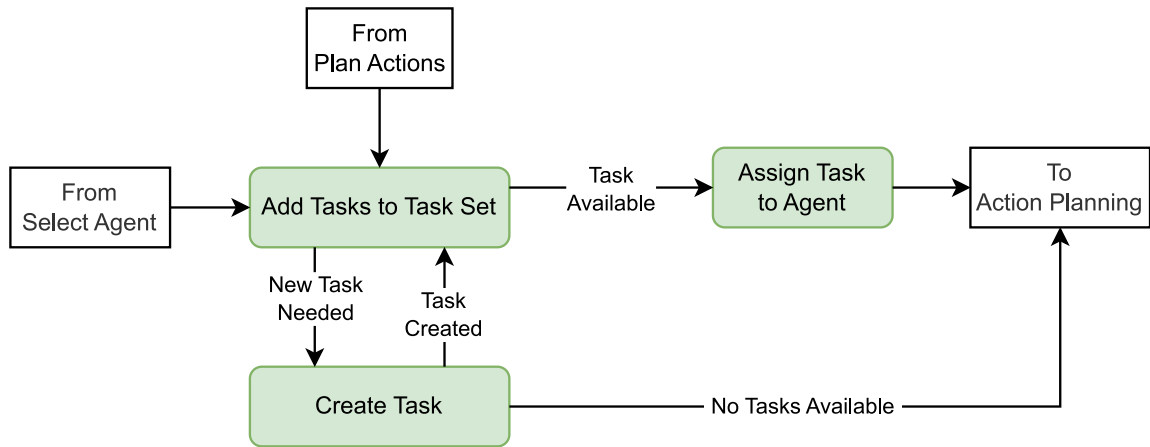


Figure 8: Graphical overview of the Task Management state.

Action Planning

During each timestep an agent must take some kind of action, even if the action is to wait in place. The act of determining which action best advances the system toward a solution state will account for the majority of an algorithm's procedure and is therefore the most involved aspect of the implementation process. Following a similar decomposition process makes it clear that an algorithm must be able to make decisions regarding certain functions: fulfilling task requirements, finding paths, and acting on planned paths. These routines each require a logical branch which guides the progression through the state machine to the correct behaviors.

Generally, an agent which is in its goal location should attempt to fulfill its objective by performing its tasked behavior. In simplistic implementations, this could be merely being in the goal node at some point in time, but the approach makes no assertions that this must be true. For example, the execution of a task could be sufficiently complicated and

time-consuming that it requires multiple timesteps. In such cases, the planning algorithm must compensate.

Agents which are not where they need to be should be driven closer to their goals while adhering to other system requirements (chiefly, no collisions along their paths). If an agent already has a plan and there are no immediate problems in continuing to execute the plan, the default case should be that it advances along its plan. This framework does not prevent the implementing engineer from altering plans in an online fashion, remaining flexible in the case where an auxiliary goal should be achieved, such as avoiding future congestion. Alternatively, if an agent has no plan at all—as may be the case immediately after the assignment of a new task—it should attempt to find a valid plan.

An implementation concern arises. Path planning operations may fail in certain cases. For example, an agent may not be able to take any action without colliding with the intent of another agent. A complete path may also not exist under current system conditions. In such cases it is possible a bounded path search could find a partial path as in the case of WHCA*. Once again, the state machine approach does not offer restrictions on how such cases should be handled, although the implementation in this work uses a collision resolution system which takes effect once all agents have declared their intents.

The agent queue should be exhausted by the end of the action planning phase, with all agents having declared some intent to take a particular action. All that remains from this point on is the validation and execution of these planned actions.

An overview of the proposed actions enacted during the Action Planning state is provided in **Figure 9**.

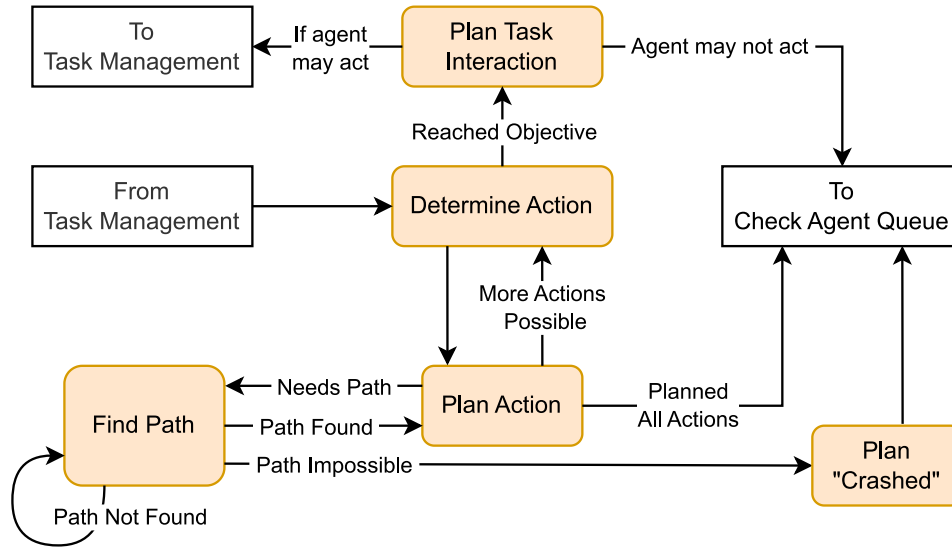


Figure 9: Graphical overview of the Action Planning state.

Action Execution

With a set of actions found for each agent in the system, the algorithm should be able to advance by executing all agent actions simultaneously. Successfully doing so represents the passage of a timestep.

As mentioned in the Action Planning section, it is possible that generated agent plans are insufficient in some way that causes collisions or other errors. As both a pragmatic and performance concern, such occurrences must be handled. Simulations which crash on the first instance of incompatible plans provide little to no data about the functionality of the algorithm in other cases. Further, not every algorithm is well-adapted to the MAPD challenges, as will be shown with the implementation of WHCA* in Chapter 5. Collecting data on the occurrence rate of failures throughout the simulation lifetime is probably useful and so the approach includes handling these cases.

If no disallowed collisions exist, then the implementation of the algorithm must be able to mutate the state of the simulation in a manner corresponding to the planned agent actions, at which point the timestep is considered completed.

An overview of the proposed actions enacted during the Action Execution state is provided in **Figure 10**. The Resolve Collisions state could attempt to find a solution internally, never leaving the Action Execution state, or re-enter the agent selection-task-plan cycle with new restrictions for the colliding agents.

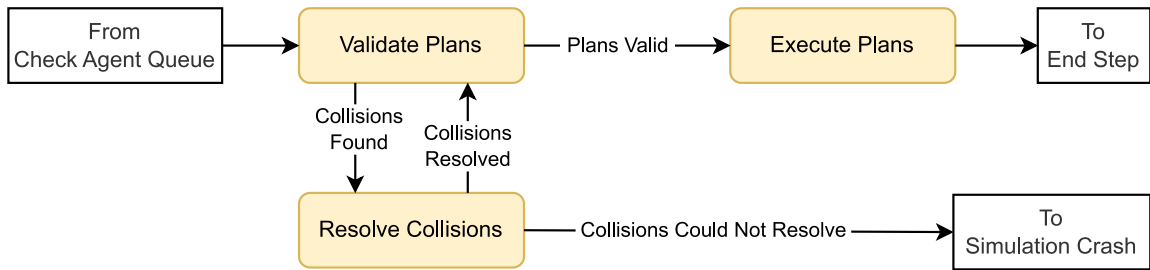


Figure 10: Graphical overview of the Action Execution state.

End of Step

To account for simulations or tests which have a defined endpoint, a branch in logic must be introduced which evaluates the current simulation state against the set of requirements for completion. Simple conditions upon which a simulation should end include elapsed timesteps, completed task counts, and unresolvable collisions. Equalizing end conditions ensures a level playing field for comparison of multiple algorithms. In the case of simulations which should run forever, it is sufficient for the ending criteria to always evaluate to false, thus keeping the system looping through new timesteps for an arbitrary amount of time.

An overview of the proposed actions enacted during the End Step state is provided in **Figure 11**.

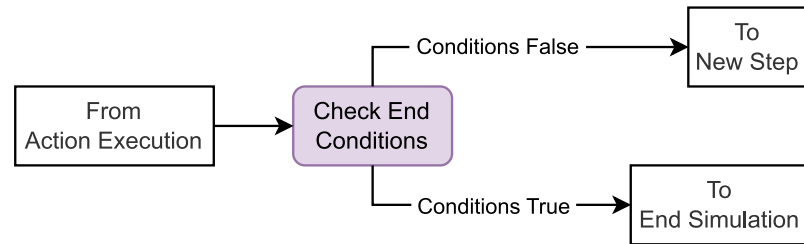


Figure 11: Graphical Overview of the End Step state.

4. Framework Application

In Chapter 2, two decoupled multi-agent problem solving algorithms were introduced and their inner workings explained. In Chapter 3, a flexible and general framework for the implementation of algorithms was presented. This chapter combines the concepts of the preceding chapters, applying the process such that the two most developed algorithms, WHCA* and TPTS, are adapted to function on equivalent playing fields through the use of the state machine approach.

To begin, the algorithms are decomposed into their underlying behaviors such that they cohere with the state machine diagrams presented in Chapter 3. With increasing granularity, routines present in the algorithms are assigned to states in the governing state machine until the algorithm is fully represented. From this position it becomes trivial to extract a strategy for practical implementation, as is done in the design of FleetBench. The use of this process, demonstrated in broad strokes in this chapter and at the implementation level in Appendix B, demonstrates the advantages of approaching MAPF and MAPD solution implementation in this manner.

This chapter assumes that the Simulation Definition state is already completed by a user configuring certain options. The actual implementation of the definition state experiences a great degree of freedom in terms of what options are available in the simulation and must be developed in an “as needed” fashion. As an example, **Table 1** lists all the configuration options for the implementation used in FleetBench at time of writing. As mentioned, these rules may be expanded significantly to cover unplanned agent

breakdowns, kinematic concerns such as rotation, or limitations on agent charge or fuel, among real-world behaviors.

Table 1. FleetBench simulation definition options.

| Category | Option Name | Choices |
|-----------------|--|-------------------------------------|
| Pathfinding | Solver Algorithm | Single-Agent A* |
| | | Multi-Agent A* (LRA*) |
| | | Cooperative A* (CA*) |
| | | Hierarchical CA* (HCA*) |
| | | Windowed HCA* (WHCA*) |
| | | Token Passing (TP) |
| | | TP with Task Swaps (TPTS) |
| | Heuristic Function | Dijkstra |
| | | Manhattan |
| | | Euclidean |
| | Heuristic Relaxation Coefficient | $1 \leq \varepsilon \leq 50$ |
| | Pathfinder Search Window Depth | $1 \leq w \leq 100$, if applicable |
| Agent Behavior | Agent Collision Handling | Respected |
| | | Ignored |
| | Task Interaction Cost | Instantaneous |
| | | Timestep |
| | Agent Count | $0 \leq i \leq \infty$ |
| Task Generation | Task Generation Technique | Scheduled |
| | | On Agent Availability |
| | Task Location Configuration | Node weights |
| | | Include/Exclude node |
| End Conditions | Simulation ends on task completions | $1 \leq O_{complete} \leq \infty$ |
| | Simulation ends on timesteps elapsed | $1 \leq t_{elapsed} \leq \infty$ |
| | Simulation ends on schedule completion | True or False |

WHCA*

Before fitting the decoupled MAPF algorithm WHCA* into the state machine model it is worth noting that WHCA* offers no particular strategy for task assignment, as

the MAPF problem assumes that all agents have a task assigned before the problem should be solved. As a result, any implementation of the WHCA* algorithm in an MAPD context will need a generic strategy for task assignment. Here, the generic routine is named ***GenerateTask***. It simply selects the first available task from the task set or creates a new task if the simulation definition allows.

Critically, WHCA* is also incomplete in the MAPD case. It fails to consistently avoid collisions during its runtime. This problem arises when agents finish their current plans, and thus have no reservations, while another agent is attempting to reach the same goal location. If the first agent to arrive finds itself trapped, it will be unable to move away while simultaneously not being able to remain in place. In order to avoid an immediate end of the simulation via the crashed state, it is necessary to develop a generic collision resolver. Even in the MAPF case, a one-agent width corridor of sufficient depth (exceeding the window size) will prevent progress from being made as neither agent will find an escape from its current position, resulting in an infinite stall. FleetBench approaches this problem using a collision resolver which prioritizes trapped agents and forces a replanning of agent motions until the problem is resolved. The resolver is presented in Appendix H, as it is not central to the work done here.

The bulk of the logic employed in WHCA* is for pathfinding, making it relatively simple to fit into the state machine model. For completeness, the algorithm is expanded to include the routines used in FleetBench for task selection and collision resolution.

Algorithm 9 shows the translated version of WHCA* in the context of the whole system loop, where lines of the original algorithm have been replaced with named routines

for ease of reference. The resulting information is directly transferrable to the state machine diagram. Extra care should be taken to ensure that the additional loops which are possible due to implementation choices. For instance, how collisions are resolved—by nullifying plans and re-entering the agent selection state or by forcing new plans within the action execution state? Implementing the former would require an additional loop which begins with agent selection, moving through task management and action planning until all agents have created valid paths. For the latter, plans are simply re-calculated during the action execution state.

| Algorithm 9. WHCA* state machine representation | State |
|--|-----------------------|
| Input is an MAPD problem $MAPD$, defined by the user | |
| 1: $M \leftarrow \text{PreProcess}(MAPD)$ | Simulation Definition |
| 2: $reserved \leftarrow \emptyset$ | |
| 3: while not endConditions(M): | End Step |
| 4: $agentQueue \leftarrow \text{Update}(agentQueue), O \leftarrow O \cup \{newTasks\}$ | New Step |
| 5: for $\forall a_i \in A$: | Select Agent |
| 6: if $MAPD$ allows task creation: | Manage Tasks |
| 7: $O \leftarrow O \cup \{\text{createTask}(MAPD)\}$ | |
| 8: if a_i has no assignment: | |
| 9: $task \leftarrow \text{choice}(\{O_{valid} \subseteq O:$ $\qquad\qquad\qquad \forall o_j \in O_{valid} \text{ not assigned or completed}\})$ | |
| 10: if $task \neq \emptyset$: | |
| 11: Assign a_i to $task$ | |
| 12: while mayAct(a_i): | Plan Actions |
| 13: if Position(a_i) \equiv Goal(a_i): | |
| 14: $plan \leftarrow plan \cup \{\text{taskInteraction}(a_i, \text{Goal}(a_i))\}$ | |
| 15: else : | |
| 16: if a_i has plannedPath: | |
| 17: $plan \leftarrow plan \cup \{\text{next}(plannedPath)\}$ | |
| 18: else : | |
| 19: $plannedPath \leftarrow$ $\qquad\qquad\qquad \text{WHCAStar}(\text{Position}(a_i), \text{Goal}(a_i))$ | |
| 20: $plan \leftarrow plan \cup \{\text{next}(plannedPath)\}$ | |
| 21: Store $plan, plannedPath$ in system memory | |

| | |
|---|------------------------|
| 22: $valid \leftarrow \text{validatePlans}(\forall plan)$ | Execute Actions |
| 23: while not $valid$: | |
| 24: $\text{resolveCollisions}(M, \forall plan)$ | |
| 25: $valid \leftarrow$ Validate all stored plans | Sim. Crash |
| 26: if $valid$ is impossible: | |
| 27: System Crashes | |
| 28: Execute all actions | Execute Actions |
| 29: System time increments | End Step |
| 30: Solution found for $MAPD$, for endConditions(M) | End Simulation |

TPTS

TPTS, being designed for the MAPD problem, more directly approaches real-world applications which use continuously active cooperative robots as part of their implementation. The algorithm has a strategy for both task selection and task exchanging, offering an improvement in efficiency by minimizing unnecessary travel time. It also offers conditional guarantees regarding completeness of the algorithm. If the system map conforms to the definitions presented in Chapter 2 for well-formedness, then it should be impossible for a collision to occur. Verifying whether those conditions are met is a task for the Simulation Definition state. In the case that a system map is not well-formed and is used for simulation anyway, a conflict resolution system must be in place to prevent the simulation from entering the crashed state.

| Algorithm 10. TPTS state machine representation | |
|---|------------------------------|
| Input is an MAPD problem $MAPD$, defined by the user | State |
| 1: $M \leftarrow \text{PreProcess}(MAPD)$ | Simulation Definition |
| 2: $isWellFormed \leftarrow \text{checkWellFormed}(M)$ | |
| 3: if not $isWellFormed$: | |
| 4: Handle badly formed problems, simulation may abort | |
| 5: $reserved \leftarrow \emptyset$ | |

| | |
|--|------------------------|
| 6: while not endConditions(M): | End Step |
| 7: $agentQueue \leftarrow \text{Update}(agentQueue), O \leftarrow O \cup \{newTasks\}$ | New Step |
| 8: for $\forall a_i \in A$: | Select Agent |
| 9: if $MAPD$ allows task creation: | Manage Tasks |
| 10: $O \leftarrow O \cup \{\text{createTask}(MAPD)\}$ | |
| 11: if a_i has no assignment: | |
| 12: GetTask(a_i) | |
| 13: while mayAct(a_i): | Plan Actions |
| 14: GetTask(a_i) | |
| 15: $valid \leftarrow \text{validatePlans}(\forall plan)$ | Execute Actions |
| 16: while not $valid$: | |
| 17: resolveCollisions($M, \forall plan$) | |
| 18: $valid \leftarrow \text{Validate all stored plans}$ | |
| 19: if $valid$ is impossible: | Sim. Crash |
| 20: System Crashes | |
| 21: Execute all actions | Execute Actions |
| 22: System time increments | End Step |
| 23: Solution found for $MAPD$, for endConditions(M) | End Simulation |

Once again, a line-by-line process of re-composing the algorithm into named routines is presented in **Algorithm 10**. However, a problem arises. Because the assignment optimization requires comparison of path lengths (line 16), the *GetTask* routine of TPTS interleaves the search for a task assignment with the planning of the path, as an optimization to avoid recalculating paths a second time. Without extra effort, this approach will not fit neatly into the state machine model, requiring the implementing engineer to jump through hoops and introduce additional logic. This issue is demonstrated in **Algorithm 11**.

| | |
|---|---------------------|
| Algorithm 11. GetTask from TPTS state machine representation Demonstrates that planning paths occurs between management of tasks, requiring a different strategy. | |
| 1: function GetTask($agent$) | Manage Tasks |
| 2: $O' \leftarrow \{o_j \in O \mid \text{no path} \in \text{reserved ends in } p_j \text{ or } d_j\}$ | |
| 3: while $O' \neq \emptyset$: | |
| 4: $task \leftarrow \text{BestTask}(\text{Position}(a_i), O')$ | |

| | |
|--|---------------------|
| 5: $O' \leftarrow O' \setminus task$ | |
| 6: if no agent assigned to <i>task</i> : | |
| 7: Assign <i>agent</i> to <i>task</i> | |
| 8: Path1(<i>agent</i> , <i>task</i>) | Plan Actions |
| 9: return <i>true</i> | |
| 10: else : | |
| 11: $oldState \leftarrow (assignments, reserved, M)$ | Manage Tasks |
| 12: $a_i' \leftarrow$ agent assigned to <i>task</i> | |
| 13: $path' \leftarrow Path(a_i')$ | |
| 14: Unassign a_i' from <i>task</i> ; Remove $path'$ from <i>reserved</i> | |
| 15: $path \leftarrow Path1(agent, task)$ | Plan Actions |
| 16: if $ path < path' $: | |
| 17: $success \leftarrow GetTask(a_i')$ | Manage Tasks |
| 18: if <i>success</i> : | |
| 19: return <i>true</i> | |
| 20: else : | |
| 21: $(assignments, M) \leftarrow oldState$ | |
| 22: if Position(<i>agent</i>) $\notin V_{endpoints}$: | |
| 23: $path \leftarrow Path2(a_i)$ | Plan Actions |
| 24: if $path \neq \emptyset$: | |
| 25: return <i>true</i> | Manage Tasks |
| 26: else : | |
| 27: if $\{o_j \in O \mid Position(a_i) \in \{p_j, d_j\}\} = \emptyset$: | |
| 28: Stay(a_i) | Plan Actions |
| 29: else : | Manage Tasks |
| 30: Path2(a_i) | Plan Actions |
| 31: return <i>true</i> | |
| 32: return <i>false</i> | Manage Tasks |

The state machine design pattern allows for two methods of fixing this problem. First, the state machine can be adjusted with a logical branch uncritically allowing this behavior, shown in **Figure 12**. This is akin to direct manipulation of the data intended to be managed within the action planning state. Because the underlying routines may be called

from anywhere, as is done in the direct implementation of TPTS, this is not really a problem.

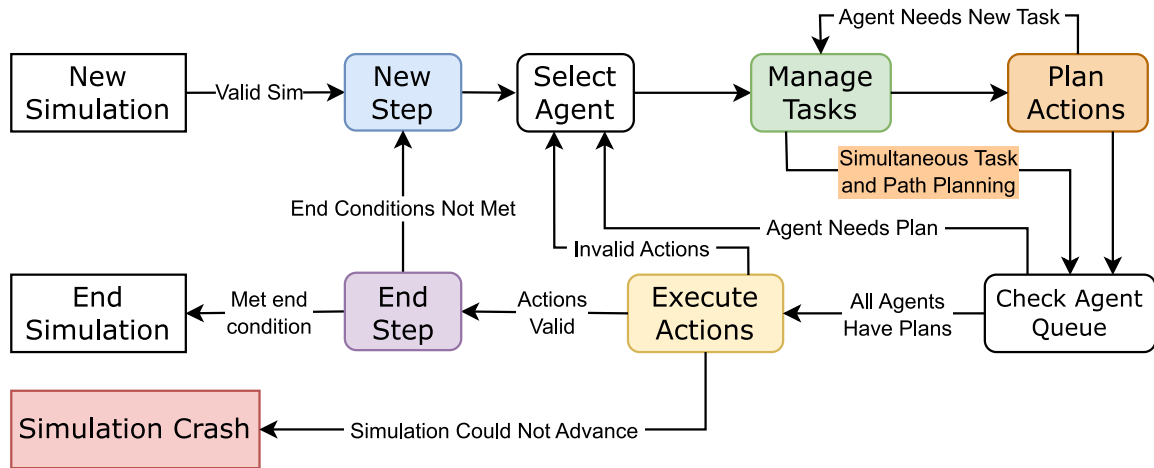


Figure 12: Modification of the state machine for simultaneous task and path planning, showing a new branch which accounts for simultaneous task and path planning during the manage tasks state.

To adhere to the state flow diagram more strictly (and enjoy its benefits) it is necessary to decouple the task swapping from path planning. This re-introduces the duplication of path searches in the case a task swap occurs, which is intuitively undesirable. However, because the intention of this work is to provide a generalizable method for testing algorithms, it is expected that a significant amount of work will be done by implementors of a particular algorithm to optimize its function in the real-world application space. This is clearly beyond the scope of this work, whose primary aim is to evaluate the *system performance* under a multitude of conditions rather than its *program runtime*, while still adhering to the logic of the algorithms in use.

To make this possible, a few pseudo-task singletons are introduced to represent an agent's intent: *IntendTask*, *IntendRest*, *IntendStay*. These will be evaluated in the Action

Planning state, shown in **Figure 9** from Chapter 3, to determine whether *Path1*, *Path2*, or *Stay* should be called, respectively. Instead, during the task management state the calls to the pathfinders *Path1_{ps}*, *Path2_{ps}*, and *Stay_{ps}* are used to find paths without requesting space in the reservation table, returning the singletons alongside the paths. Paths which remain valid in the future do not need to be recalculated, so can be stored in the agent's memory. Agents which have planned a move before a task swap renders the plan useless will have their intended action for the timestep revoked, thus being caught by the check agent queue state and made to seek alternative actions. The new procedure is given in **Algorithm 12**, showing that the task determination and the action planning are decoupled and solved independently.

| Algorithm 12. Detangled TPTS state machine representation Input is an MAPD problem <i>MAPD</i> , defined by the user | | State |
|--|--|------------------------------|
| 1: $M \leftarrow \text{PreProcess}(MAPD)$ 2: $isWellFormed \leftarrow \text{checkWellformed}(M)$ 3: if not $isWellFormed$: 4: Handle badly formed problems, simulation may abort 5: $reserved \leftarrow \emptyset$ | | Simulation Definition |
| 6: while not $\text{endConditions}(M)$: | | End Step |
| 7: $agentQueue \leftarrow \text{Update}(agentQueue), O \leftarrow O \cup \{newTasks\}$ | | New Step |
| 8: for $\forall a_i \in A$: | | Select Agent |
| 9: if $MAPD$ allows task creation: 10: $O \leftarrow O \cup \{\text{createTask}(MAPD)\}$ 11: if a_i has no assignment: 12: $O' \leftarrow \{o_j \in O \mid \text{no path} \in reserved \text{ ends in } p_j \text{ or } d_j\}$ 13: while $O' \neq \emptyset$: 14: $task \leftarrow \text{BestTask}(\text{Position}(a_i), O')$ 15: $O' \leftarrow O' \setminus task$ 16: if no agent assigned to $task$: 17: Assign a_i to $task$ 18: else : 19: Store assignments, reservations in memory | | Manage Tasks |

| | | |
|-----|--|--|
| 20: | $a_i' \leftarrow$ agent assigned to $task$ | |
| 21: | $path' \leftarrow \text{Path}(a_i')$ | |
| 22: | Unassign a_i' from $task$ | |
| 23: | Remove $path'$ from $reserved$, a_i' | |
| 24: | $path, intent \leftarrow \text{Path1}_{ps}(a_i, task)$ | |
| 25: | if $ path < path' $: | |
| 26: | $success \leftarrow \text{GetTask}(a_i')$ | |
| 27: | if $success$: | |
| 28: | break | |
| 29: | else : | |
| 30: | Restore assignments, reservations | |
| 31: | if $intent$ is not “intendTask”: | |
| 32: | if $\text{Position}(agent) \notin V_{endpoints}$: | |
| 33: | $path, intent \leftarrow \text{Path2}_{ps}(a_i)$ | |
| 34: | else : | |
| 35: | if $\{o_j \in O \mid \text{Position}(a_i) \in \{p_j, d_j\}\} = \emptyset$: | |
| 36: | $path, intent \leftarrow \text{Stay}_{ps}(a_i)$ | |
| 37: | else : | |
| 38: | $path, intent \leftarrow \text{Path2}_{ps}(a_i)$ | |
| 39: | while $\text{mayAct}(a_i)$: | |
| 40: | If $path$ is still valid use it, otherwise: | |
| 41: | if $intent$ is “intendTask”: | |
| 42: | $\text{Path1}(a_i, task)$ | |
| 43: | else if $intent$ is “intendRest”: | |
| 44: | $\text{Path2}(a_i)$ | |
| 45: | else if $intent$ is “intendStay”: | |
| 46: | $\text{Stay}(a_i)$ | |
| 47: | $valid \leftarrow \text{validatePlans}(\forall plan)$ | |
| 48: | while not $valid$: | |
| 49: | $\text{resolveCollisions}(M, \forall plan)$ | |
| 50: | $valid \leftarrow$ Validate all stored plans | |
| 51: | if $valid$ is impossible: | |
| 52: | System Crashes | |
| 53: | Execute all actions | |
| 54: | System time increments | |
| 55: | Solution found for $MAPD$, for endConditions(M) | |

This implementation could therefore be used without adjustment to the top-level state machine diagram, minimizing the amount of “hard-coding” that need be done during

implementation of a new algorithm to an existing simulator based on the principles laid out in Chapter 3. In the appendices, it will be shown that FleetBench is capable of implementing the first solution without compromising state flow, proving that this is generally not a significant concern and may be left up to the discretion and preference of the user wishing to extend FleetBench's functionality.

5. Implementation, Testing, and Results

To demonstrate the efficacy of the implementation strategy presented in Chapter 3, the state machine design pattern was used in the creation of a simulation and test program named FleetBench. FleetBench was developed to provide a guided user interface (GUI) overtop a sufficiently performant and extensible simulation of multi-agent problems and solutions while remaining intuitive and simple, all with the goal of increasing the accessibility of implementing and testing solutions to MAPF and MAPD problems. FleetBench natively allows a user to define the number, order, and positions of agents and tasks. Before simulation, the end user is able to define a number of options which determine the behavior of the simulation at runtime, as discussed in Chapter 4. During simulation, a state machine designed exactly as presented in Chapter 3 and Appendix B is used to drive the execution of all implemented algorithms. Data is collected and displayed continuously to the user during runtime, providing instant feedback about the performance of an algorithm. To aid in analysis, the state of the simulation is reconstructable from saved data at any particular timestep, providing an intuitive way to seek explanations for algorithmic failures. Developed in Python and using the native GUI library TkInter, the application exposes a customized rendering engine accessible in user-defined scripts which produces visualizations of found paths, agent motions, key object highlighting, and labeling on a per-state basis at the user's request.

FleetBench is designed to be extensible. By placing script files in the appropriate application path, a user is able to add additional algorithms to the program or modify the

behavior of existing work. The state machine structure provides a regulated way of calling the algorithm's subroutines, which are listed in Appendix B. These functions must be present in the algorithm scripts and some data must be returned in a certain format, but no further requirements regarding their function are imposed.

FleetBench has two modes. The first, available immediately upon launching the program and supplying a map file, allows the user to define the system to be used in simulation. Agents and tasks may be created, with assigned positions and names. This initial state of the system can be saved and manipulated, allowing rapid minor changes to be made between simulation runs.

The second mode, shown in **Figure 13**, runs and visualizes the progression of the system as the algorithm of choice is executed and modifies the system state. Configuration of the simulation runtime allows the user to define which states are visualized. On the right-hand panel a deck of controls are provided. Beneath the controls are representations of agent activity, position, and assignment as well as task statuses and tabulated data about the progression of the simulation.

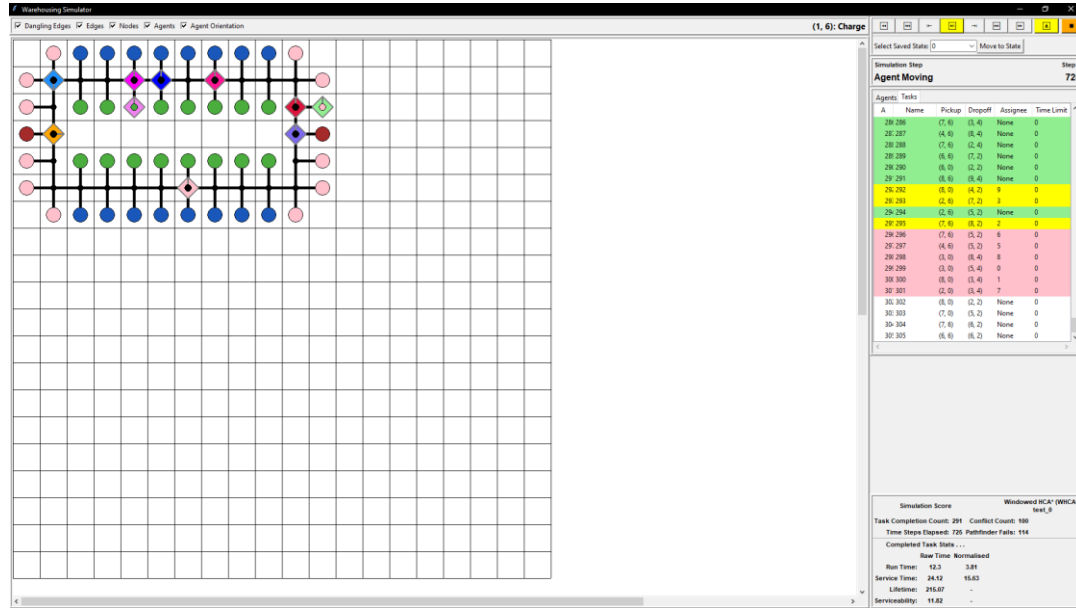


Figure 13: An image of the simulation window in FleetBench. A defined problem has been submitted and the program has simulated 725 timesteps, showing partial completion of the task set on the right, and the current state of the system on the left.

A second application called GraphRendering was developed by modifying existing code from an open-source program named TileBasic to provide a visual process for designing the system map using tiles. It currently produces 4-neighbor graphs, with the ability to set specific node roles such as pickup, delivery, and rest. This application is also built in Python, using the TkInter GUI library. The map generation interface is shown in **Figure 14**.

It is possible to generate maps externally, using the map input file format given in Appendix D.

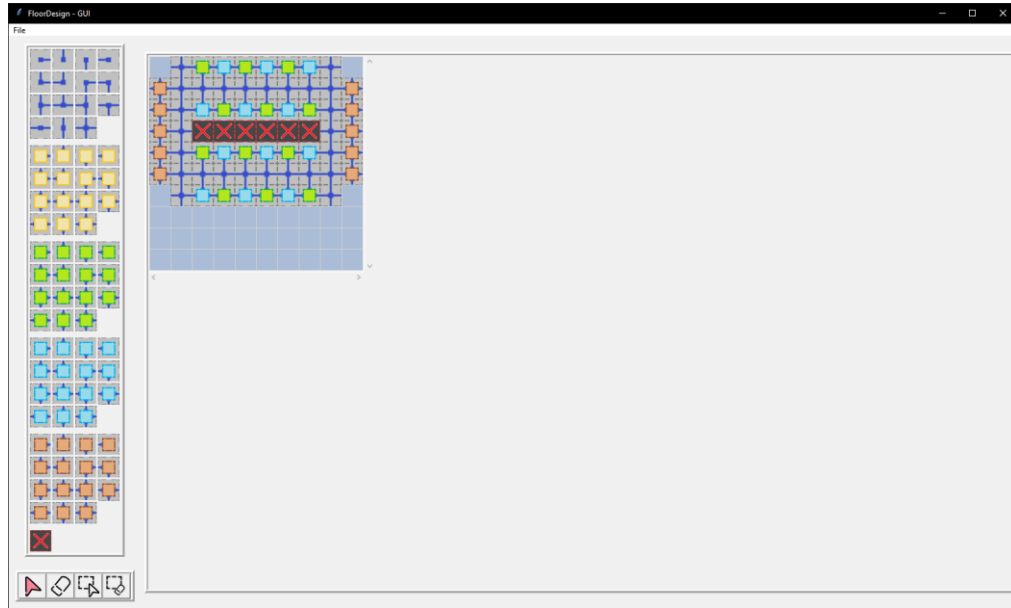


Figure 14: Screenshot of the map creation process in GraphRendering.

The intended workflow for a user wishing to evaluate the performance of an algorithm on their multi-agent system is as follows:

- Develop a map either via the GraphRendering application or a custom script, adhering to the map file format given in Appendix D.
- Create an implementation of the algorithm if one is not already provided, adhering to the extension documentation in Appendix F.
- In FleetBench, create a new session using the map file.
- Design the initial placement of agents in the system.
- Optionally, define an initial set of tasks.
 - If a predefined list of tasks should be used, the user will need to provide a comma-separated values file as described in Appendix E.

- Define simulation configuration options, including which algorithm should be used, how new tasks are introduced to the system and upon what conditions (if any) the simulation should end.
- Run the simulation, recording the resulting data for analysis.

Because of the effect the past has on how a simulation proceeds in the future, it is expected that variation of individual parameters will produce significant changes in the performance of an algorithm. Care should be taken to ensure that the results of different tests are treated fairly in analysis. Repeatability of results is an important factor, which FleetBench adheres to by using the same pseudorandom generator for all operations which require a “random” choice.

Design of Experiment

Several test cases were produced to demonstrate FleetBench’s implementation of the MAPF and MAPD algorithms. Test cases were developed quickly using the GUI of GraphRendering for the map design and the features in FleetBench to design agents and task schedules, supporting the usability and flexibility of the applications. They are designed to showcase certain features and behaviors of algorithms and multi-agent problems in general. Case 1 shows a bottleneck around access to a single node which is used repeatedly. Case 2 demonstrates an analogy of a real-world warehouse problem. Case 3 demonstrates the problem of agents always seeking to minimize the A* algorithm’s h -

Score without considering the real-world cost of motion, as all implementations used consider all edge weights to be equivalent.

Case 1

The system map for case 1 is shown in **Figure 15**. The system is composed of a 1-width corridor with three agents, shown in orange, pink, and red. The green, leftmost, node is the deposit location, while blue nodes are pickup locations. The agent start positions are endpoints, satisfying TP and TPTS requirements.

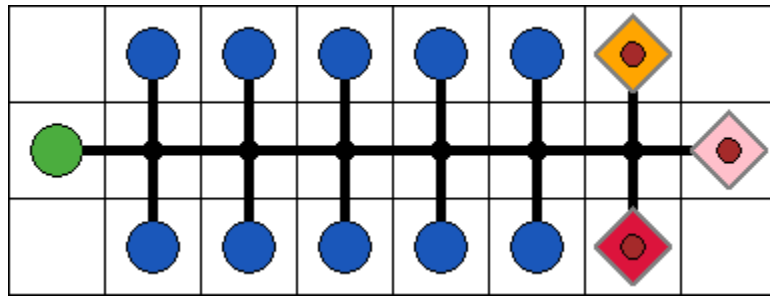


Figure 15: System map for test case 1. A single corridor of width 1 which is flanked by pickup nodes. There is a single delivery node, in green, to the left. Three agents must work together to move objects to the delivery node.

The first case could be analogized as cooperatively retrieving books in a library. There is a single task delivery node (a cart of books), acting as the ending location for items retrieved from the storage system (the library shelves) managed by three agents. Several nodes exist to represent the many sorting locations at which items may be stored. In this relatively constrained space, the motion of agents presents a challenge wherein agents must dip into destination nodes to avoid collisions moving down the corridor, which may itself impact the retrieval of task objects from their storage locations.

The simulation configuration options used to obtain the results in experimentation are given in **Table 2**. The data was collected once for HCA* and WHCA* and again for TP and TPTS to show performance differences between the “upgraded” versions of the algorithms, as well as between the two families of algorithms. Each choice for the simulation is configurable, encouraging experimentation and repeat trials.

Table 2. System configuration for case 1 experiments.

| Option | HCA* | WHCA* | TP | TPTS |
|-------------------------------------|-------------------------------|-------|----|------|
| Map Name | case_1 | | | |
| Agent Starting Positions | {(6,0), (7,1), (6,2)} | | | |
| Initial Task Set | None | | | |
| A* Heuristic Function | Manhattan Distance | | | |
| A* Heuristic Relaxation Coefficient | 1 | | | |
| Window Size | | 5 | | |
| Agent Collisions | Respected | | | |
| Task Interaction Time Cost | Instantaneous | | | |
| Task Schedule | case_1_schedule (Appendix C) | | | |
| End Condition | All Scheduled Tasks Completed | | | |

Case 2

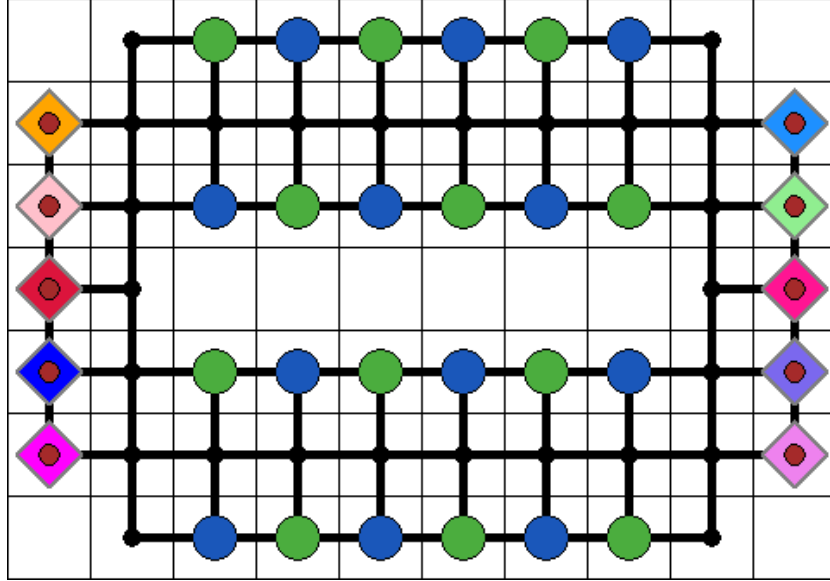


Figure 16: System map for test case 2. A warehouse reorganization problem with 10 agents, starting on the sides of the system map. The map has two horizontal corridors of width three. The delivery and pickup locations are alternating and fully interconnected with other nodes on the sides of these corridors.

A second test case is developed (shown in **Figure 16**), very similar in form to the warehousing situation presented in [12]. In this case the analogy is closer to warehouse internal reorganization, with an arbitrary pattern of pickup and delivery nodes. Agents must again travel through corridors, only this time the spaces are more interconnected, allowing agents to move over task endpoints during their journeys. This widens the bottleneck considerably, allowing a greater number of agents to be present in the system. In these trials, ten agents were used, starting on endpoints at either end of the corridors.

As before, simulation configuration options are presented in **Table 3**. For this experiment only the implementations of WHCA*, TP, and TPTS are compared. To

demonstrate changes in performance of the WHCA* algorithm as the window size is changed, the WHCA* experiments are repeated for a few window sizes.

Table 3. System configuration for case 2 experiments.

| Option | WHCA*-3 | WHCA*-5 | WHCA*-10 | TP | TPTS |
|-------------------------------------|---|---------|----------|----|------|
| Map Name | case_2 | | | | |
| Agent Starting Positions | {(0,1), (0,2), (0,3), (0,4), (0,5), (9,1), (9,2), (9,3), (9,4), (9,5),} | | | | |
| Initial Task Set | None | | | | |
| A* Heuristic Function | Manhattan Distance | | | | |
| A* Heuristic Relaxation Coefficient | 1 | | | | |
| Window Size | 3 | 5 | 10 | | |
| Agent Collisions | Respected | | | | |
| Task Interaction Time Cost | Instantaneous | | | | |
| Task Schedule | case_2_schedule (Appendix C) | | | | |
| End Condition | All Scheduled Tasks Completed | | | | |

Case 3

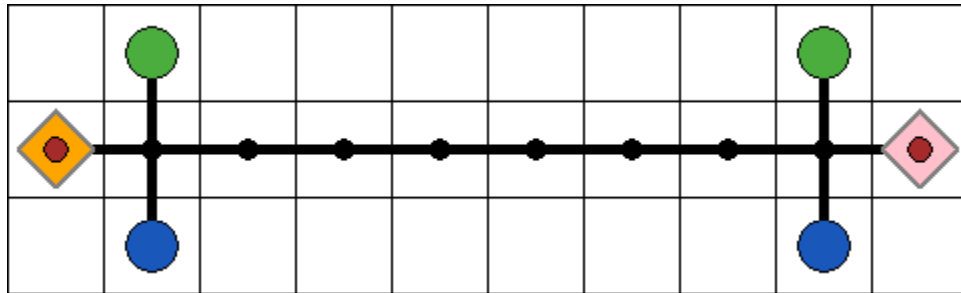


Figure 17: System map for test case 3. A 1-width bottlenecked corridor with task endpoints at each side. Two agents share the space and are forced to use the nodes at the ends of the corridor to exchange places.

This test case exposes a consequence of treating all edge weights as having the same value when finding paths using A*. The map is represented in **Figure 17**. Two agents

must share a long corridor with no ability to exchange positions except at the corridor's ends. The heuristics used encourage agents to always move in the direction of their goals as immediately as possible. This behavior can create situations in which agents perform pointless movement actions toward a goal that is impossible to travel to without backpedaling. The resulting sequence will show that they must move backward to avoid the path of the blocking agent until it is possible for the two agents to exchange positions via some rotation through nodes outside the corridor.

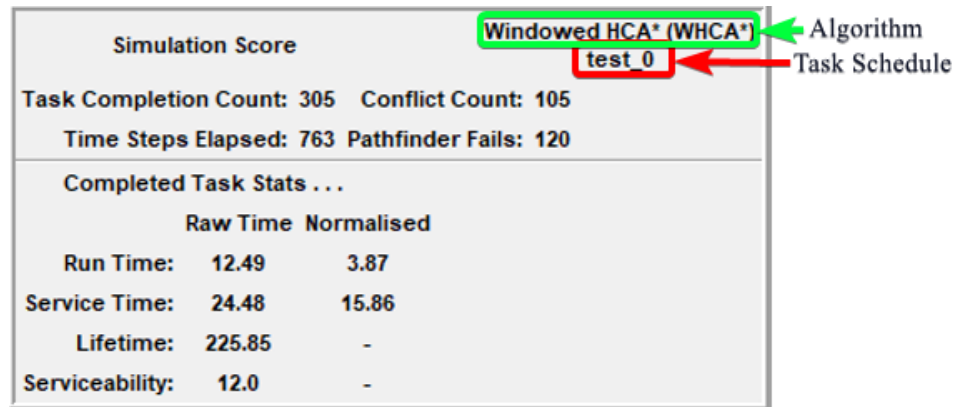
This behavior is discussed in the Results section of this chapter. Once again, the configuration options used for this simulation are summarized in **Table 4**. The windowing behavior of WHCA* is of special interest. Its performance is compared to HCA*, which plans full paths when possible, and against CA*. LRA* simply fails to solve the problem, as noted in [18].

Table 4. System configuration for case 3 experiments.

| Option | CA* | HCA* | WHCA* |
|-------------------------------------|-------------------------------|------|-------|
| Map Name | case_3 | | |
| Agent Starting Positions | {(0,2), (9,2)} | | |
| Initial Task Set | None | | |
| A* Heuristic Function | Manhattan Distance | | |
| A* Heuristic Relaxation Coefficient | 1 | | |
| Window Size | | | 5 |
| Agent Collisions | Respected | | |
| Task Interaction Time Cost | Instantaneous | | |
| Task Schedule | case_3_schedule (Appendix C) | | |
| End Condition | All Scheduled Tasks Completed | | |

Results

Experiments are run on each test case in the manner described in the previous section, with the aim of evaluating the performance of each algorithm within the system. By equalizing the playing field as discussed in previous chapters, it is possible to draw quantitative conclusions. FleetBench implements a rudimentary set of datapoints for which data is collected during the simulation lifetime. The results are constantly displayed, as shown in **Figure 18**.



| Simulation Score | | |
|--|----------|------------|
| Task Completion Count: 305 Conflict Count: 105 | | |
| Time Steps Elapsed: 763 Pathfinder Fails: 120 | | |
| Completed Task Stats ... | | |
| | Raw Time | Normalised |
| Run Time: | 12.49 | 3.87 |
| Service Time: | 24.48 | 15.86 |
| Lifetime: | 225.85 | - |
| Serviceability: | 12.0 | - |

Annotations:
 - Green box around 'Windowed HCA* (WHCA*)' with a green arrow pointing to 'Algorithm'.
 - Red box around 'test_0' with a red arrow pointing to 'Task Schedule'.

Figure 18: Data panel of a simulation in FleetBench.

Because all test cases are equalized using the same task scheduling system, an obvious performance metric to examine is the amount of time taken to complete the schedule. This is simply equivalent to the number of timesteps elapsed while using a simulation end condition which triggers when all scheduled tasks are completed.

FleetBench also records some data regarding the timing of interactions with tasks as a measure of the moment-to-moment performance of the system. FleetBench names four

different aspects of task completion, each defined by a different calculation. There are four moments in time relevant to the processing of a task:

- Task Creation: The timestep at which the task is entered into the simulation's active task list.
- Task Assignment: The first timestep at which an agent is assigned to the task.
- Task Pickup: The timestep at which an agent executes the first portion of the task by "picking up" the task for delivery.
- Task Completion: The timestep at which an agent executes the last portion of the task by "delivering" the task to the endpoint.

From these definitions four time intervals of interest are developed. Run Time captures the time spent by agents completing a task once the agent reaches and completes the "pickup" portion of the task. Service Time measures the amount of time agents are preoccupied with a task, starting from the timestep on which the task was assigned. Lifetime measures how long tasks spend in the system before being completed. As the Lifetime of tasks increases, the system is falling further and further behind the incoming task set. Serviceability approximates how long it takes agents to reach the starting node of the next task, upon assignment, from their current position. If agents travel very long distances to reach the next task (as may be the case in unoptimized assignment implementations) this number will be quite large. These definitions give rise to the following formula:

$$\text{Run Time} = T_{\text{Completion}} - T_{\text{Pickup}}$$

$$\text{Service Time} = T_{\text{Completion}} - T_{\text{Assignment}}$$

$$\text{Lifetime} = T_{\text{Completion}} - T_{\text{Creation}}$$

$$\text{Serviceability} = T_{\text{Pickup}} - T_{\text{Assignment}}$$

Two additional measures are provided as normalizations of the baseline measure. In this application, the normalization is intended to capture an idea of how much time is spent on a task compared to the optimal minimum completion time. This value is only consistently defined using the minimum travel time from pickup endpoint to delivery endpoint of the task, as an agent assigned to a task could be anywhere in the system. This minimum time interval for the task to be completed is determined using an A* search which ignores the presence of all agents and obstacles in the system, similar to the HCA* approach presented in [18]. In FleetBench, the Run Time is normalized, capturing timesteps spent avoiding collisions during an agents path from pickup to delivery endpoints. The Service Time is also normalized, providing an idea of the time lost due to inoptimal starting position of agents servicing a task. The two values are given by these formula:

$$\text{Run Time}_{\text{norm}} = \text{Run Time} - \text{Min Time}$$

$$\text{Service Time}_{\text{norm}} = \text{Service Time} - \text{Min Time}$$

The values reported by FleetBench are the simulation mean values for all above formula, calculated via summation of all values and division by the number of tasks in the system. Normalized values before the end of a simulation should be treated with caution, as the averaging of minimum times is done for completed tasks while the system may have

many tasks in progress. No attempt is made to measure an agent’s progress on a per-timestep basis.

Two kinds of failure are possible during the execution of an algorithm. The first is a collision, where two agents plan paths which result in a vertex or edge conflict. Such failures are termed “Agent Conflicts” to distinguish from the second type of failure. When an agent seeks a path and is unable to find any route to its destination node the failure is called a “Pathfinder Failure”. This is most often due to the agent searching for a path being trapped in its location, rather than the obstructions being distant. Otherwise, the search depth would simply increase, gaining larger degrees of freedom, until the path is found. As a result of “Pathfinder Failures” frequently leading to collisions which must be resolved, the failure counts tend to be highly coupled.

With metrics defined, the results of experiments on the three test cases provided can be discussed.

Case 1

Case 1 is simulated using the task schedule provided in Appendix C. The resulting data are provided in **Table 5**. All algorithms from **Table 2** successfully solve the multi-agent problem such that all tasks are completed, although the WHCA* family of algorithms notably suffers many agent conflicts and failed pathfinding operations. Each conflict results in algorithmic inefficiency, as agents must replan their routes from a new position, leading to increased time taken to reach destinations and potentially a cascade of

obstructions to other agents. However, the fundamental operating principles of TP and TPTS prevent it from being performant in this situation. Because agents are prevented both from being assigned tasks and from planning paths ending in the same location as any other agents, the bottleneck becomes very problematic. With only one destination node, it is not possible for any second agent to plan paths in the system. Therefore all tasks are completed by the same agent in this case, resulting in a massive increase in elapsed timesteps. This produces a much larger task lifetime.

However, because all tasks assigned are the best tasks for an agent to be executing given its position, the normalized task run times are very optimal. The serviceability measure in this case largely represents information about the system map, as agents mostly travel through the same space repeatedly with only small deviations to avoid imminent collisions, never having to travel inordinately far to reach an outlier task.

HCA*, compared to WHCA*, takes overall slightly fewer timesteps to finish solving the problem. This is because HCA*, by planning complete paths, avoids a small number of frivolous movements due to its greater foresight, staggered path planning, and the WHCA* window size. Smaller windows increase the frequency of situations where agents get trapped in dead-ends, which are completely avoided by fully planning the paths.

Table 5. Results of experimentation on case 1.

| Option | HCA* | WHCA* | TP | TPTS |
|---------------------------|-------------|--------------|-----------|-------------|
| Tasks Completed | 90 | 90 | 90 | 90 |
| Timesteps Elapsed | 377 | 383 | 1008 | 729 |
| Agent Conflicts | 76 | 76 | 0 | 0 |
| Pathfinder Failures | 59 | 60 | 0 | 0 |
| Run Time | 6.58 | 6.86 | 4.42 | 4.1 |
| Run Time (normalized) | 2.48 | 2.76 | 0.32 | 0 |
| Service Time | 12.38 | 12.61 | 10.87 | 8.09 |
| Service Time (normalized) | 8.28 | 8.51 | 6.77 | 3.99 |
| Lifetime | 110.88 | 117.73 | 425.87 | 231.93 |
| Serviceability | 5.8 | 5.76 | 6.44 | 3.99 |

These data support initial hypotheses about the performance of these algorithms based on their properties. According to this experiment, for a small and relatively simple situation involving object retrieval to a single deposit location, HCA* is the ideal algorithm to be used. It is possible that performance issues arise in larger and more complicated scenarios (which can also be tested in FleetBench), in which case using WHCA* (with an appropriate choice for window size) appears likely to provide a sufficient solution.

Case 2

Case 2 is simulated using the task schedule provided in Appendix C, and the resulting data are provided in **Table 6**. Being a very interconnected graph, there are many degrees of freedom for most agents attempting to find paths. This fact appears to produce comparable solutions with all algorithms, and few conflicts when using algorithms which do not assure generality. All five algorithms seem to complete the task schedule in a similar amount of time, except for TP. The very similar run time values indicate that agents' paths

were typically unobstructed after picking up the task. WHCA*-3 did experience elevated run times, likely because of its low planning depth leading to more frequent replanning to avoid collisions as agents more easily claim paths which end up disrupting other agents. Increasing the window size appears to result in more collisions, as a consequence of greater planning depth introducing more reservations in any given area over time.

TPTS experiences some deviation in timing values, each of which have several explanations. By carefully examining the playback of the simulation, it becomes obvious that in certain situations agents plan excessively large numbers of waiting moves, likely waiting for their objectives to be clear of agents which are waiting for new tasks. This results in high service times and is a known issue in TP and TPTS [32]. Additionally, agents tend to begin the simulation by moving in similar directions. This results in task selections clogging up areas with the restriction that no plan end in the same location as another plan in the system. As a result, agents are frequently forced to choose tasks which are actually further away and resolve in an entirely different area of the system, where there are not agents currently operating. This results in higher service times. Alternatively, certain tasks are ignored for a significant amount of time as they happen to not be near agents when agents are free. Some benefits are still retained, however, as agents will still attempt to complete the fastest tasks soonest, reducing the overall average lifetime of tasks. These patterns, not as visible in TP, prompt questions about the task assignment optimization.

Table 6. Results of experimentation on case 2.

| Option | WHCA*-3 | WHCA*-5 | WHCA*-10 | TP | TPTS |
|---------------------------|---------|---------|----------|-------|-------|
| Tasks Completed | 150 | 150 | 150 | 150 | 150 |
| Timesteps Elapsed | 232 | 225 | 224 | 275 | 249 |
| Agent Conflicts | 1 | 4 | 5 | 0 | 0 |
| Pathfinder Failures | 1 | 4 | 5 | 0 | 0 |
| Run Time | 7.53 | 7.09 | 7.07 | 7.89 | 7.38 |
| Run Time (normalized) | 1.53 | 0.9 | 0.88 | 1.7 | 1.19 |
| Service Time | 14.79 | 14.25 | 14.01 | 14.69 | 25.38 |
| Service Time (normalized) | 8.6 | 8.07 | 7.82 | 8.5 | 19.19 |
| Lifetime | 71.57 | 66.35 | 63.39 | 73.08 | 56.09 |
| Serviceability | 7.25 | 7.17 | 6.94 | 6.8 | 18.0 |

Once again, the results seem to support the hypothesized behavior of these algorithms, affirming the efficacy of FleetBench in testing warehousing situations with a variety of settings. FleetBench also enabled the user to investigate specific performance cases, resulting in a deeper understanding of how and why TP and TPTS may struggle to optimize certain axes of performance.

Case 3

Case 3 is simulated using the task schedule provided in Appendix C, and the resulting data are provided in **Table 7**. This test case is largely intended to be watched rather than analyzed and exposes a consequence of failing to perfectly analogize the system to real-world applications. Specifically, the current FleetBench implementation assumes that a movement from one node to another incurs the same costs as staying in place. Because the path planner always seeks to move in the direction of least distance from the

goal, an agent may pointlessly advance toward its goal even if it knows it will have to move backward in the future to avoid the path of another agent sharing the space. This sequence is represented in **Figure 19**. In a real-world application, this would result in wasted fuel costs, unnecessary wear, and a greater risk of agents experiencing crashes or operational faults.

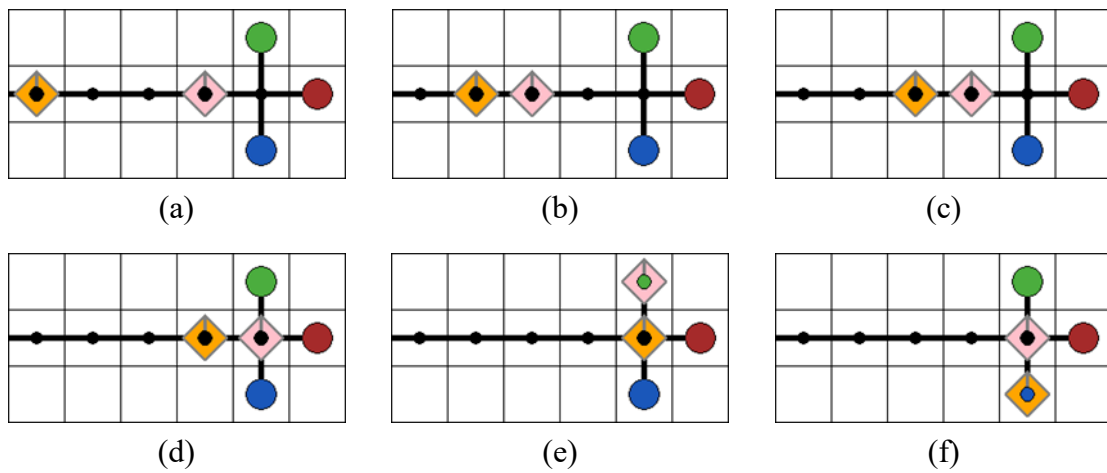


Figure 19: Sequential moves by two naïve agents. Pink is attempting to move left, while orange moves right. It can be seen in images (a) and (b) that pink makes moves in its goal direction which it must then undo in images (c) and (d) due to orange’s higher priority. In (e) and (f), the orange agent moves out of the way, allowing pink to move to its goal direction unimpeded.

WHCA* seems to slightly underperform compared to HCA*, just as in case 1, further supporting the idea that optimal paths are found when full plans are made. CA* and HCA* perform identically. As expected, the optimization of the abstract hierarchy is merely one of computation time and data reuse.

Table 7. Results of experimentation on case 3.

| Option | CA* | HCA* | WHCA* |
|---------------------------|------------|-------------|--------------|
| Tasks Completed | 26 | 26 | 26 |
| Timesteps Elapsed | 159 | 159 | 187 |
| Agent Conflicts | 5 | 5 | 5 |
| Pathfinder Failures | 5 | 5 | 5 |
| Run Time | 6.08 | 6.08 | 7.92 |
| Run Time (normalized) | 0.31 | 0.31 | 2.15 |
| Service Time | 11.92 | 11.92 | 14.08 |
| Service Time (normalized) | 6.15 | 6.15 | 8.31 |
| Lifetime | 49.96 | 46.96 | 63.5 |
| Serviceability | 5.85 | 5.85 | 6.15 |

These results demonstrate that the algorithms perform in a consistent manner, providing similar results as in case 1 but in a different environment. It also demonstrates the utility of FleetBench’s playback function as a visual tool for gaining insight into the operating principles of the underlying implementation.

6. Conclusion

This thesis presented an overview and definition of the multi-agent problem and two solutions to the MAPF and MAPD problems (Chapter 2), a generalizable approach to adapting solutions to the problems for implementation (Chapters 3, 4), and a workflow consisting of two computer applications which ensures a level testing field for multiple algorithms (Chapter 5). The stated purpose of the work was to simplify the testing process, minimize knowledge requirements when comparing algorithms for performance, provide an improvement in visualization over existing solutions, and expose additional data to the user to enable deeper analysis.

Using the implementation strategy of Chapter 3, FleetBench was developed. Its basic function as a state machine and user interface for creating test cases was extended to include execution of two families of algorithms. FleetBench and GraphRendering were used to generate several arbitrary test cases in a fast and intuitive process. Algorithms belonging to each of the two families presented in Chapter 2 were then tested against these test cases. This process demonstrated known and hypothesized behavior from research in the field within the implementation, affirming its accuracy. Features of FleetBench proved useful in visually confirming the behavior of the algorithms, enabling more intuitive analysis. Statistical data collected during the simulation was used to fuel analysis and springboard the development of explanations for the behavior of the system in simulation.

Taken together, these results represent success—the intended goals for the research and implementation work were achieved.

Future Work

The algorithms implemented in this paper are of a certain type: they operate with similar methods of storing or processing data and generally rely on the optimality of the A* algorithm for pathfinding. Other approaches exist in research which may not integrate as smoothly with the framework presented in Chapter 3 as WHCA* or TPTS. For example, BIBOX [21] and Push and Swap [17] solve multi-agent problems through topological analysis of the system state. Centralized approaches such as CBS [15] which seek optimal moves at each step were said to be difficult to scale and have not been implemented in FleetBench. New directions in the research make use of Machine Learning to apply pattern-based solutions, as in PRIMAL [16]. While the framework presented in Chapter 3 can be used as a target for adapting the function of each of these solutions to a standard testing environment, it may become simpler for the user to adjust the presented state machine to better suit these approaches rather than use the current methods.

In terms of performance, FleetBench is far from a perfect application. While it performs the stated objective with sufficient performance in relatively simple and small cases, it suffers from performance issues as the scale increases. As discussed in the background research, real-world warehousing problems span enormous spaces and field thousands of robots to aid in fulfillment of tasks. Analogizing such problems within

FleetBench would prove extremely difficult—even for powerful computers—due to a lack of performance optimization in the current implementation.

FleetBench could also be made smarter, though this is largely up to the user extending its function. If conclusions can be drawn about the excellence of an algorithm in solving a particular class of multi-agent problem, analysis could be performed before testing the algorithm to indicate an algorithm may be of particular interest to the user.

Additional statistical tracking could be implemented to deepen analysis. For example, a user may be interested in the rate of task completion throughout the simulation. A significant drop could indicate a bottleneck, allowing a user to easily identify the simulation timesteps leading up to the performance throttle to start a deeper investigation.

The extensibility of FleetBench is ultimately a subjective experience but changes could be made which make the process smoother. Of particular interest would be dynamic option generation, as currently a user extending the program is required to implement their own UI code using the techniques from Appendix F, executed at FleetBench's runtime. Currently, the component scripts are broken out for modularity. It is possible that a user would prefer to instead supply a library in any format they prefer to develop, which is an approach currently not supported by FleetBench.

Most of all, every additional algorithm which is correctly added to the baseline functionality of FleetBench increases its value as a testing tool, saving effort and time for the end user.

Finally, as a pair of prototype-level applications, FleetBench and GraphRendering may contain bugs and lack quality of life features present in more matured programs. They could also be merged into a single application that provides end-to-end development and testing.

APPENDIX

APPENDIX A:

FLEETBENCH CODE OVERVIEW

The underlying implementation of FleetBench as a computer application is an extensive work of programming. The source code is too large to be practically included in this manuscript (at time of writing, over 8500 lines across 72 files). The appendices are intended to direct readers to the repositories containing the version of the application used in this work. An exception to this is contained in Appendix B: the state machine implementation. This is because the state machine is central to the topic of the paper, and its code is non-repetitive and critical.

In the future, FleetBench may be further developed and changed. If so, the repository will be updated with notice of this development, but this original version shall be preserved.

The list of files in **Table 8** only contains the locations of code which should be of particular interest, but all code used in the project may be obtained in this repository: [\[repository link goes here\]](#)

The scripts which implement the functionality of the MAPF and MAPD solutions discussed in this paper are split across several modules, each named in accordance with their intended function (though additional functionality is permitted). There are four types of modules in use: *Managers*, *Taskers*, *Pathfinders*, and *Movers*. Not every algorithm implementation makes use of all script types. As a result, a default implementation is supplied and there is no requirement that a solving algorithm need have a unique version of each module type. Of course, any shared implementations must function properly for all algorithms which depend on them.

Manager scripts are used as a central storage of shared information. The shared reservation table presented in Cooperative A* implementations would be found here. In the simulation code, the object containing these scripts is named `self.infoShareManager`.

Tasker scripts are responsible for obtaining and assigning tasks to agents. These scripts may or may not generate tasks but are required to at least assign tasks from the available set in the simulation space. In the simulation code, the object containing these scripts is named `self.simulationTasker`.

Pathfinder scripts are the implementation of the path search methods used by each algorithm. For all algorithms in this work, some variation on the A* search is used and found here. In the simulation code, the object containing these scripts is named `self.agentActionAlgorithm`.

Mover scripts are responsible for the execution of agent plans, updating the system state with the new information required to advance the simulation. In the simulation code, the object containing these scripts is named `self.agentMovementManager`.

The following table indicates where each script may be found for each algorithm in the repository.

Table 8: Locations of algorithm scripts in FleetBench.

| Algorithm | Manager Script | Tasker Script | Pathfinder Script | Mover Script |
|-----------|----------------|---------------|-------------------|--------------|
| A* | | | | |
| LRA* | | | | |
| CA* | | | | |
| HCA* | | | | |
| WHCA* | | | | |
| TP | | | | |
| TPTS | | | | |

In order to use FleetBench without modification, an installer has been provided: [\[link to installer goes here\]](#)

To modify FleetBench, it is best to use a virtual environment and package manager. For convenience and ease of reference, all libraries in use have been collected into a file named `requirements.txt`. The contents of the file are reproduced here:

[\[list of libraries goes here\]](#)

APPENDIX B:

STATE MACHINE IMPLEMENTATION

The central idea presented in this paper is the use of a state machine to drive a general simulation strategy which promotes modularity, maintainability, and configurability. In this appendix, the underlying Python implementation of the state machine is presented and explained. For clarity, code pertaining to other application features such as visualization, debugging, and statistical tracking is omitted to focus on the primary components which create a change in the system state.

State Machine Definition and Execution

The implementation begins with a pragmatic observation about the way in which a user is expected to interact with the state machine. Using the control panel provided in FleetBench, the user may advance the simulation state continuously through each state, or by a single state per click. It is also possible to jump directly to the start of a selected timestep for review. The expected result is that some form of user interface update occurs alongside the execution of the behaviors in the state the simulation advances to.

The first code excerpts represent some form of boilerplate that should be executed on each state change, while the latter is the bulk of desired action in the system, which changes on each state change. As a result, it proved best to embed the code which mutates the simulation state in a general function which refers to some value of requested state.

A function is defined to make the cyclic process of state advancement easy to repeat. Recall that code not directly related to the execution of state behaviors is omitted:

```

1:  def simulateStep(self, stateID):
2:      try:
3:          self.simulationStateMachineMap[stateID]["exec"]()
4:      except:
5:          self.requestedStateID = "simulationErrorState"
6:          # Call the next state
7:          if self.doNextStep:
8:              self.simulationStateMachineNextStep(self.simulationStateID)

```

This function calls some other function stored in a state machine map using a passed state ID (line 3). If the state fails to execute for any reason, then the simulation prepares to enter an error state (line 5). Otherwise, if the system is continuously advancing through states (line 7), another function is called which uses a timer to call `simulateStep` again after some amount of time (configured by the user) has passed. This delay exists to allow users to view the visualization of a state.

The simulation is driven forward by an action of setting the value of `self.requestedStateID` to some string known to exist in the state map and returning the state execution function.

The state map is designed to package all required information for what should happen during the execution of a state using a simple pattern, shown by example for the new step state:

```

9:   self.simulationStateMachineMap = {
10:     "newSimStep": {
11:       "exec": self.newSimStep,
12:       "stateLabel": "Preparing Next Step",
13:       "renderStateBool": simulationSettings["renderNewSimStep"],
14:       "stateRenderDuration": simulationSettings["renderNewSimStepTime"]
15:     },

```

Lines 10-15 represent a completely defined state in this implementation. The "exec" key of the dictionary provides access to a function handle that should be called when a state is to be executed on line 3. The "stateLabel" field identifies the string which should be displayed when the simulation enters the state. The two rendering values indicate whether the state should be rendered and for how long before advancing to the next state.

The process is repeated for every state, resulting in a state map covering all states in the diagram. The code is repetitive and omitted for brevity of the discussion.

New Sim Step

This state is representative of a new timestep in the simulation. In the current iteration, only two things occur here.

```

16: def newSimStep(self):
17:     # Prepare for a new step
18:     # First, save the current state if it is the right time
19:     if stepID % self.SIMULATION_STATE_SAVE_INCREMENT == 0
20:         self.stateHistoryManager.copyCurrentState(stepID)
21:     self.requestedStateID = "selectAgent"
22:     return

```

First, a copy of the current simulation state is made on every *n*th timestep. In this case, using modulo of $n=1$, every timestep is saved in memory for review (line 19). The copy compiles data regarding agent positions, task status, and agent searches via specific functions within the algorithm scripts. This takes place in a separate object which manages the timetable of saved states (line 20).

Second, the state plans to advance to the "selectAgent" state and returns (lines 21-22). The simulation now waits until the next state is triggered, via delay or user interaction.

Select Agent Step

This state is used to select an agent for which actions in the following states should be executed. The agent is saved using its internal ID, which is internally generated and managed by FleetBench.

```

23: def selectAgent(self):
24:     # Select an agent, keeping in mind there may be a queue of agents

```

```

25:     # Agent must only be selected according to simulation definition rules
26:     if self.algorithmType == "sapf":
27:         # Single-agent pathfinding methods
28:         # Only use the first agent in the list, user error if multiple agents
29:         self.currentAgent = self.simAgentManagerRef.agentList[0]
30:         self.agentQueue = []
31:     elif self.algorithmType == "mapf":
32:         try:
33:             # Search the queue until an agent is found that has not acted
34:             for agent in self.agentQueue:
35:                 if self.simAgentManagerRef.agentList[agent].actionTaken is False:
36:                     newAgent = agent
37:                     break
38:             self.currentAgent = self.simAgentManagerRef.agentList[newAgent]
39:         except StopIteration:
40:             # The queue is empty, so the simulation step can end
41:             self.requestedStateID = "agentCollisionCheck"
42:             return
43:         # There was an agent in the queue, so it should act
44:         self.requestedStateID = "taskAssignment"
45:         return

```

There are two relevant cases pertaining to the choice of algorithm for this state. For completeness, FleetBench allows the use of raw A* for pathfinding which is only usable by a single agent. To account for this, the algorithms have been typed with "sapf" or "mapf" to indicate single-agent pathfinding or multi-agent pathfinding, respectively.

If an algorithm is implemented for single-agent operations, then the first agent in the list (typically an agent "0") is used exclusively for the entirety of the simulation (lines 26-30).

Otherwise, for multi-agent applications, the list of agents is searched for the first agent which has not already indicated a planned action, which is marked by the object property ".actionTaken" (lines 31-37). If no such agent exists, the simulation assumes it has implemented plans for all agents, and the plans can now be evaluated for potential failures (lines 39-42). If an agent is found, then the corresponding agent object is saved as the "current" agent in the simulation (line 38). The simulation then advances to checking the task state of the agent via state request and returns (lines 43-45).

Task Assignment

This state is analogous to the task management state. It has the ability to assign a task to the "current" agent, execute calls for generating new tasks, and pull new tasks into the simulation from the schedule.

```

46:     def taskAssignment(self):
47:         if self.simulationSettings["tasksAreScheduled"]:
48:             tasksToPop = []
49:             for i, task in enumerate(self.taskSchedule):
50:                 if eval(task[3]) <= self.stepCompleted:
51:                     tasksToPop.append(i)
52:                     # The task can be brought in
53:                     pickupNode = task[0]

```

```

54:         dropoffNode = task[1]
55:         timeLimit = eval(task[2])
56:         creationTime = eval(task[3])
57:         name = task[4]
58:         self.simTaskManagerRef.createNewTask(pickupNode=pickupNode,
59:             dropoffNode=dropoffNode, timeLimit=timeLimit, taskName=name,
60:             timeStamp=creationTime)
61:     # Pop the tasks
62:     for index in sorted(tasksToPop, reverse=True):
63:         self.taskSchedule.pop(index)
64:     # Assign a new task if the task status is "unassigned"
65:     # Or generate a new task if there are not, and generation on demand
66:     # is enabled
67:     # Or if there are no tasks, and generation is disabled, the agent
68:     # is aimless
69:     if self.currentAgent.taskStatus == "unassigned" and
70:         self.currentAgent.currentTask is None and
71:         self.currentAgent.pathfinder.returnNextMove() is None:
72:         taskSelect = self.simulationTasker.selectTaskForAgent(
73:             self.currentAgent, timeStamp=self.stepCompleted)
74:         if type(taskSelect) is not int and taskSelect is not True:
75:             if self.simulationSettings["taskGenerationAsAvailableTrigger"]
76:                 == "ondemand":
77:                 self.generateTask()
78:                 self.requestedStateID = "taskAssignment"
79:                 return
80:         else:
81:             # If task cannot be generated yet, then the agent
82:             # is free and directionless and does nothing
83:             self.requestedStateID = "selectAction"
84:             return
85:     else:
86:         if self.simulationSettings["taskGenerationAsAvailableTrigger"] ==
87:             "onassignment":
88:             self.generateTask()
89:             # Task generated successfully
90:             self.requestedStateID = "selectAction"
91:             return
92:     else:
93:         # Agent does not need a task, so continue to the next state
94:         self.requestedStateID = "selectAction"
95:     Return

```

As a matter of priority, tasks which should be entered into the simulation state because of a task schedule are handled first, otherwise they could be skipped. If tasks should be added, a list of the new tasks is generated. When the simulation was defined, a file containing the schedule (of format described in Appendix E) was read in and processed. The resulting list of tasks was saved as "self.taskSchedule".

This list of tasks is processed at each step. A task is created using the information in the list when the corresponding list entry indicates that a task should be released on a timestep which is less than or equal to the current simulation timestep (lines 49-60). When this happens, to prevent duplication, the tasks are also popped from the task schedule and will never be added to the simulation again (lines 61-63).

From this point, it is decided whether an agent will be assigned a new task.

If an agent is not currently assigned to a task and has reached the end of its planned movement sequence, then it can be assigned a task (lines 69-71). The tasking script is called to enact a required function "selectTaskForAgent", which must return an internal task ID if an agent was assigned a task (lines 72-73).

If an agent was not assigned a task as a result of calling the "selectTaskForAgent" function, then the simulation checks to see if the user-defined rules allow for a task to be generated. In this case, the situation is encoded as a string value: "ondemand". If the case is a match for any rules then a task is generated using the relevant tasker script's "generateTask" function and the simulation re-enters the state to try assigning a task to the agent again (lines 77-79). Otherwise, the simulation was unable to generate a task, leaving the agent without a directive. The simulation should still advance to the action planning state where aimless agents will be handled (lines 80-84).

If the agent was successfully assigned a task, then the simulation checks to see if the user-defined rules allow for a task to be generated. In this case, the situation is encoded with the string value "onassignment". If this case is a match for any rules, then a task is generated (lines 85-89). The simulation then progresses to the action planning state (lines 90-91).

If an agent was already assigned a task, then the state is simply skipped and the simulation advances directly to the action planning state (lines 92-95).

Select Action

This state exists to decide whether an agent ought to try and move toward its goal or, if it is at its goal, perform some action pertaining to the execution of its task.

```

96:  def selectAction(self):
97:      # Agent needs to determine its action for this step
98:      # If the agent has already acted, then its "turn" is over
99:      if self.currentAgent.actionTaken == True:
100:          self.requestedStateID = "selectAgent"
101:          return
102:      # If the agent is already on its task-given target tile,
103:          it should act immediately
104:      if self.currentAgent.currentNode == self.currentAgent.returnTargetNode()
105:          and self.currentAgent.currentTask is not None:
106:          # An action is about to be taken, but it could be free
107:          self.requestedStateID = "taskInteraction"
108:          if self.simulationSettings["agentMiscOptionTaskInteractCostValue"] ==
109:              "No cost for pickup/dropoff":
110:              self.currentAgent.actionTaken = False
111:              self.requestedStateID = "taskInteraction"
112:          elif self.simulationSettings["agentMiscOptionTaskInteractCostValue"] ==
113:              "Pickup/dropoff require step":
114:              self.currentAgent.actionTaken = True
115:              self.requestedStateID = "taskInteraction"
116:          return
117:      else:

```

```

118:         # If the agent is not at its target tile, it should be trying
119:         to move there
120:         self.requestedStateID = "agentPlanMove"
121:         return

```

Immediately, an agent is checked for having already planned some action. If it has, then the simulation advances directly to choosing another agent (lines 99-101).

Otherwise, via priority, if the agent is in the target location for its currently assigned task and status, the agent should attempt to resolve its task state (lines 104-105). Two possibilities arise based on user-defined rules currently available in FleetBench: the interaction consumes the agents turn, or the interaction is “free” and resolves instantly.

If the interaction is free, then the simulation simply advances directly to the interaction state (lines 108-111).

If the interaction consumes the agents turn on this timestep, then the agent is marked as having taken an action, and the simulation advances to the task interaction state where necessary updates to task and agent status are made (lines 112-114).

If the agent still needs to reach its goal node, then the simulation advances to the movement planning state, where it will either advance toward or seek a path to the goal location (lines 117-121).

Task Interaction

This state triggers an update to task and agent statuses. Preconditions for whether this should be possible will have been satisfied before this state is entered, meaning no guarding is necessary. Exiting the state is done with a dependence on whether the agent can pursue further actions.

```

122: def taskInteraction(self):
123:     # Agent should be able to interact with the task
124:     interactionResult = self.currentAgent.taskInteraction(
125:         self.currentAgent.currentNode, self.stepCompleted)
126:     if interactionResult == "completed":
127:         # If the task was finished, count the completion
128:         self.tasksCompleted = self.tasksCompleted + 1
129:         if self.simulationSettings["taskGenerationAsAvailableTrigger"] ==
130:             "completed":
131:             self.generateTask()
132:     elif interactionResult == "pickedUp":
133:         if self.simulationSettings["taskGenerationAsAvailableTrigger"] ==
134:             "onpickup":
135:             self.generateTask()
136:     # If this counted as an action, then the agent cannot move
137:     if self.currentAgent.actionTaken == True:
138:         self.requestedStateID = "checkAgentQueue"
139:     elif self.currentAgent.actionTaken == False and
140:         self.currentAgent.currentTask is None:
141:         # If not, then it can also move
142:         self.requestedStateID = "taskAssignment"

```

```

143:     elif self.currentAgent.actionTaken == False and
144:         self.currentAgent.currentTask is not None:
145:         self.requestedStateID = "selectAction"
146:     return

```

As mentioned, the first action taken is to have the agent execute its interaction routine, supplying its current location and the current simulation timestep for logging purposes (lines 124-125).

The next action taken depends on whether the task was completed or started as a result of the interaction (remember that the tasks in FleetBench consist of pickup and delivery only).

If the task was completed, then its completion should be logged for data (line 128). Additionally, if the user-defined rules for the simulation state a new task should be generated every time a task is completed, then a task should be generated (lines 129-131).

If the task was only started, then a check is done for a user-defined rule that causes a task to be generated when a task is started (lines 132-135).

With tabulation of task interactions completed, a decision must be made regarding which state is to be advanced to. If the user-defined rules state that an agent interacting with a task results in a timestep being used up, the agent should be marked as having a plan and the simulation should advance to selecting a new agent (lines 137-138). If the agent could continue to move but does not have a task, it will need a new task (lines 139-142). If it does have a task, then it is sufficient to move immediately toward planning a new action (lines 143-145).

Plan Move

This state produces a large number of outcomes. The objective is to seek and reserve the optimal action for an individual agent in a particular timestep and system state.

```

147: def agentPlanMove(self):
148:     # Determine whether the agent can move or needs to find a path first
149:     agentTargetNode = self.currentAgent.returnTargetNode()
150:     if agentTargetNode is None:
151:         # Agent doesn't currently have a target
152:         agentTargetNode = self.simulationTasker.handleAimlessAgent(
153:             self.currentAgent)
154:     if self.currentAgent.pathfinder is None or
155:        self.currentAgent.pathfinder.invalid == True:
156:         self.currentAgent.pathfinder = self.agentActionAlgorithm(
157:             self.currentAgent.numID, self.simCanvasRef, self.simGraph,
158:             self.currentAgent.currentNode, agentTargetNode,
159:             self.agentActionConfig, self.infoShareManager, self.currentAgent,
160:             self.simulationSettings)
161:         self.requestedStateID = "agentPathfind"
162:         return
163:     nextNodeInPath = self.currentAgent.pathfinder.returnNextMove()
164:     if nextNodeInPath is not None:
165:         validMove = self.agentMovementManager.submitAgentAction(
166:             self.currentAgent, (self.currentAgent.currentNode,

```

```

167:         nextNodeInPath))
168:     if validMove is True or validMove is None:
169:         self.currentAgent.actionTaken = True
170:         self.requestedStateID = "checkAgentQueue"
171:     else:
172:         self.conflicts = self.conflicts + 1
173:         self.requestedStateID = "agentPlanMove"
174:     return
175: else:
176:     # If the agent does not have a complete path, it needs to find one
177:     self.currentAgent.pathfinder = self.agentActionAlgorithm(
178:         self.currentAgent.numID, self.simCanvasRef, self.simGraph,
179:         self.currentAgent.currentNode, agentTargetNode,
180:         self.agentActionConfig, self.infoShareManager, self.currentAgent,
181:         self.simulationSettings)
182:     self.requestedStateID = "agentPathfind"
183:     return

```

The first determination to be made is what an agent is attempting to do with its movement (line 149). If the agent lacks an objective, then a call is made to the tasker script method "handleAimlessAgent", which should return some kind of objective for the agent to seek (lines 150-153).

Next, the simulation checks to see whether the instance of the pathfinder belonging to the current agent is still valid, supplying a new instance if not. The pathfinder will obviously not have found a path at this point, so the simulation state is advanced to the pathfinding state (lines 154-162).

If instead the agent does have a valid pathfinder, the simulation checks to see if it also has a planned path by requesting the next move in the path (line 163). If there is no planned path, then the pathfinder instance is fed updated information about the objectives and the simulation enters the pathfinding loop (lines 177-183).

If the agent does have a planned path, then the next move to be taken is extracted and passed to the movement manager script via a "submitAgentAction" call. The move is checked for its validity and the value is returned (lines 164-167). If the returned value indicates that the move is acceptable, then the agent is marked as having planned an action and the simulation seeks a new agent by advancing the simulation to check the agent queue (lines 168-170).

If the action results in a problem of some sort, a conflict is logged and the simulation moves to try planning again (lines 171-174). In these cases it is up to the script developer to ensure something about the system changes between the action being submitted the first and second time, else an endless loop could be created which does not trip any recursion limits.

Pathfinding

This state is intended to be called cyclically until a complete path to the goal is found or proven to be impossible by the pathfinding script. This is only done to enable hooking of visualization functions which update on each call to the pathfinder, demonstrating the

progress of the search. This is not a requirement, and the script designer could immediately return the complete path in a single execution of the state if desired.

```

184: def agentPathfind(self):
185:     # For speed, only use the rendered version of the pathfinder if the
186:         state is being rendered
187:     if self.simulationStateMachineMap["agentPathfind"]["renderStateBool"]:
188:         pathStatus = self.currentAgent.pathfinder.searchStepRender()
189:     else:
190:         pathStatus = self.currentAgent.pathfinder.searchStep()
191:
192:     if pathStatus == False:
193:         self.requestedStateID = "agentPathfind"
194:         return
195:     elif pathStatus == True:
196:         self.requestedStateID = "agentPlanMove"
197:         return
198:     elif pathStatus == "wait":
199:         self.pathfindFailures = self.pathfindFailures + 1
200:         self.agentMovementManager.submitAgentAction(self.currentAgent, "crash")
201:         self.currentAgent.pathfinder.__reset__()
202:         self.currentAgent.actionTaken = True
203:         self.requestedStateID = "checkAgentQueue"

```

Two separate function calls exist to separate rendered search steps from unrendered search steps: "searchStepRender" and "searchStep". As TkInter is not a highly performant GUI library and Python is not a highly performant language, not requiring any draw actions to the simulation canvas is highly preferred when possible (lines 187-190).

Each call returns a status indicator for whether the path has been found.

If the returned value is `False`, then the pathfinder did not find a complete path in this loop. This is distinct from proving that a path is impossible, and leads to re-entering the pathfinding state to search again (lines 192-194).

If the returned value is `True`, then the pathfinder has completed its search for a path and the simulation can advance to movement execution states (lines 195-197).

If the path is proven to be impossible, then the expected return string is "wait". When this happens, the agent is experiencing a confused state where it cannot determine what its best action is. This is treated like a collision: the pathfinder is reset, but the action for this timestep is logged as a collision and the system advances to the movement execution state where the collisions are handled (lines 198-203).

No other returned values are acceptable.

Movement Execution

This state is responsible for making function calls that mutate the state of the system by executing agent plans for the timestep (if no conflicts arise). If there are conflicts, then the agents must form new plans, or the simulation would crash or break user-defined rules.

```

204: def agentMove(self):
205:     # Asks the movement manager to verify there are no collisions
206:     on this step of the simulation
207:     if self.agentCollisionBehavior == "Respected":
208:         conflicts = self.agentMovementManager.checkAgentCollisions()
209:         if isinstance(conflicts, dict):
210:             # Did not resolve, do new planning
211:             self.conflicts = self.conflicts + 1
212:             self.agentQueue = conflicts["agents"]
213:             for agent in self.agentQueue:
214:                 self.simAgentManagerRef.agentList[agent].actionTaken = False
215:                 self.simAgentManagerRef.agentList[agent].pathfinder.__reset__()
216:                 self.requestedStateID = "selectAgent"
217:             return
218:         elif isinstance(conflicts, int):
219:             # Just report the number of conflicts that needed to be resolved
220:             self.conflicts = self.conflicts + conflicts
221:             self.requestedStateID = "endSimStep"

```

The check for conflicts only needs to be done at this point for cases in which agents existing on the same node or edge is problematic behavior (line 207). If agents are allowed to move through each other by a user-defined rule, then there is no need to do any checks and movement is assumed to have been executed at the same time as planning occurred.

Checking for conflicts is done by calling another function in the movement manager script, "checkAgentCollisions". This function returns nothing, an integer count of the number of collisions which were resolved by the manager, or a dictionary of agents who are involved in a collision which needs to be resolved by re-starting the state machine's cycle for those agents (lines 209-217). The movement manager is able to determine the order in which these colliding agents act by mutating the priority queue of agents in the system.

If able, the system will naturally advance to the end of step state (line 221).

Checking the Agent Queue

Checking the agent queue amounts to seeking the next agent for which a plan needs to be made in the current timestep. The agent queue is treated as a priority list which can be changed over the lifetime of the simulation to reflect a change in priorities. This is most useful in resolving collisions which use a reservation table to change the possible actions for each agent involved in the crash.

```

222: def checkAgentQueue(self):
223:     for agent in self.agentQueue:
224:         if self.simAgentManagerRef.agentList[agent].actionTaken is False:
225:             # If at least one agent is found that has not acted, select it
226:             self.requestedStateID = "selectAgent"
227:             return
228:     # Otherwise, all agents have acted and the simulation should advance
229:     self.requestedStateID = "agentMove"
230:     return

```

By looping over the agent queue in an ordered fashion and evaluating each candidate for whether it has planned an action in this timestep, the first located agent is guaranteed to be the highest priority agent which still needs to commit to an action. Once this agent has been found, the state can immediately advance to the agent selection state (lines 223-227).

If there are no such agents, then a complete plan for this timestep has already been formed and the simulation should execute the planned actions (lines 229-230).

End Simulation Step

The end of a simulation step should be reached only when all agents have performed some action for the timestep. At this point it becomes important to update the simulation display with the new tabulated data about the simulation performance, as the new state has been set in stone. The state either allows procession to a new timestep, or ends the simulation if the relevant conditions are met.

```

231: def endSimStep(self):
232:     self.requestedStateID = "newSimStep"
233:     # Check if simulation objectives have been met
234:     if self.simulationSettings["tasksAreScheduled"]:
235:         # If using a task schedule
236:         if len(self.taskSchedule) == 0:
237:             # If there are no more tasks to be released
238:             if any(task.taskStatus != "completed" for
239:                   task in self.simTaskManagerRef.taskList.values()):
240:                 # And there are tasks left to complete
241:                 self.scheduleCompleted = False
242:             else:
243:                 # And there are no tasks left to complete
244:                 self.scheduleCompleted = True
245:         self.updateSimulationStats()
246:         for conditionType, evaluateCondition in self.simEndTriggerSet.items():
247:             if evaluateCondition[0] is True:
248:                 conditionTuple = evaluateCondition[1]
249:                 triggerValue = conditionTuple[0]
250:                 currentValue = getattr(self, conditionTuple[1])
251:                 if currentValue >= triggerValue:
252:                     self.requestedStateID = "endSimulation"
253:                     return
254:             else:
255:                 pass
256:         if self.infoShareManager is not None:
257:             self.infoShareManager.updateSimulationDepth(self.stepCompleted+1)
258:         # Reset all agents action taken states
259:         self.agentQueue = self.agentMovementManager.getCurrentPriorityOrder()
260:         for agent in self.agentQueue:
261:             self.simAgentManagerRef.agentList[agent].actionTaken = False
262:         self.agentMovementManager.setCurrentPriorityOrder([])

```

By default, the next state to be advanced to is the new step state (line 232). Simulation statistics are also calculated and their display values updated (line 245).

Three conditions are currently implemented in FleetBench. If any of the criteria are fulfilled during the evaluation process in the end step state, the simulation terminates successfully.

The first evaluated condition is completion of the supplied task schedule. This is represented by all tasks having been removed from the task schedule field and all tasks in the system having the "completed" state. If this is the case, the condition is marked as True. If not, then the condition is False (lines 234-244).

The second and third conditions are evaluated via direct numerical comparison; whether the number of timesteps or number of tasks has reached some user-determined value set at the beginning of the simulation.

Comparison of all three conditions to the goal values is regulated by a single data structure called the "simEndTriggerSet" which is currently defined as:

```

263: self.simEndTriggerSet = {
264:     "endOnTaskCount": [endOnTaskCount, (endTaskCount, "tasksCompleted")],
265:     "endOnStepCount": [endOnStepCount, (endStepCount, "stepCompleted")],
266:     "endOnSchedule": [endOnSchedule, (True, "scheduleCompleted")]
267: }
```

The first element in the lists is a Boolean value representing whether the condition should be evaluated at all. The first element in the tuple is the target value at which a simulation should be caused to terminate. The second element of the tuple is the string representation of a variable used to supply a "getattr" call to fetch the current value for comparison against the target value when the conditions are evaluated.

In this way, a loop can be used to evaluate all conditions. If any condition returns true, the simulation will terminate instead of advance to the new simulation step (lines 246-255).

If no end conditions are met, the simulation needs to prepare to advance to the next timestep.

First, the shared information manager (if it exists) is updated to expand any reservation information for an additional timestep (line 256-257).

Next, the agent priority queue is updated by fetching the movement manager's version of the priority queue (259). This is done so that the order in which agents plan actions can be varied by the algorithm scripts, rather than hiding this functionality away from users who wish to extend FleetBench. Every agent in the queue needs to be reset in order to plan a new action (lines 260-261).

Finally, the priority queue in the movement manager is reset to an empty list, to be populated as new actions are submitted and collisions are resolved by other systems (line 262).

End Simulation State

This state is mostly barren, simply acting as a terminal state for the simulation to reach. It updates the UI and prevents any further state changes from occurring without closing the window (lines 269-270).

```
268: def endSimulation(self):
269:     self.doNextStep = False
270:     self.simulationStopTicking()
```

There is no need to do anything else, as the try-except block surrounding the execution of all state behaviors already leads to the error state in case of problems.

It is encouraged that solving algorithm scripts define and raise their own errors, which will be reproduced in the terminal of the program, with traceback information.

Mandatory Script Functions

The state machine has been thoroughly discussed in the previous sections. In this overview a number of functions which must exist in the algorithm script objects were referenced. For any extension of FleetBench with new solving algorithms, the state machine therefore enforces a set of required functions.

The required functions and the arguments passed by the state machine are summarized in the following tables:

Table 9: Pathfinder script required functions. Call lines relative to this appendix.

| Function Name | Call lines | Arguments | Return Values |
|-------------------|------------|-----------|------------------|
| .returnNextMove | 71, 163 | None | Node ID, or None |
| .searchStepRender | 188 | None | Status Code |
| .searchStep | 190 | None | Status Code |
| .reset | 201, 216 | None | None |

Table 10: Tasker script required functions. Call lines relative to this appendix.

| Function Name | Call lines | Arguments | Return Values |
|---------------------|---------------------|---|------------------|
| .generateTask | 77, 88, 131, 135 | Current Timestep | Internal task ID |
| .selectTaskForAgent | 72 | Current Agent Object, Current Timestep | Internal task ID |
| .handleAimlessAgent | 152 | Current Agent Object | Node ID |

Table 11: Mover script required functions. Call lines relative to this appendix.

| Function Name | Call lines | Arguments | Return Values |
|--------------------------|-------------------|---|-----------------------------|
| .submitAgentAction | 165, 200 | Current Agent Object Tuple composed of (fromNode, toNode) | Validity Code |
| .checkAgentCollisions | 208 | None | Conflict count or dict |
| .getCurrentPriorityOrder | 259 | None | Sorted list of agent IDs |
| .setCurrentPriorityOrder | 262 | List | None |

Table 12: Manager script required functions. Call lines relative to this appendix.

| Function Name | Call lines | Arguments | Return Values |
|------------------------|-------------------|------------------|----------------------|
| .updateSimulationDepth | 257 | Next timestep | None |

APPENDIX C:

TEST DATA RECREATION

The operation of FleetBench is a highly deterministic process. Following the simulation setting information provided in Chapter 5 for each of the three test cases, the data may reproduced using provided files. However, the task of generating these files has not been thoroughly explained and will be here.

Map Files

Fundamentally, the map file generated by GraphRendering for use in FleetBench is a .json file which stores information about the selected tiles. Information about this file's format and how it is used by FleetBench can be found in Appendix D.

The map files used in the experiments of Chapter 5 are provided as part of the code repository for this work. Direct access is provided in **Table 13**.

Table 13: Test case map file access locations.

| | |
|--------|--------------------------------------|
| Case 1 | [link to map file 1] |
| Case 2 | [link to map file 2] |
| Case 3 | [link to map file 3] |

Task Schedule Files

Task schedules are generated by and submitted to FleetBench as comma-separated value files. The first row is a set of header data which is only used to verify the format of the file and must be present. Each row of data represents enough information to define a task. Information about this file's format and how it is used by FleetBench can be found in Appendix E.

The task schedule files used in the experiments of Chapter 5 are provided as part of the code repository for this work. Direct access is provided in **Table 14**.

Table 14: Test case task schedule file access locations.

| | |
|--------|---|
| Case 1 | [link to task schedule 1] |
| Case 2 | [link to task schedule 2] |
| Case 3 | [link to task schedule 3] |

FleetBench Session Files

Taken together with the configuration information in Chapter 5, the map files are enough to reconstruct the test conditions. Task schedules are selected at the simulation runtime. However, the placement of agents in the grid is also a potentially tedious task.

This information is saved by FleetBench whenever a session file is saved to disk. The session file is a binary file which cannot be directly read generated by the Python `pickle` library functions. This is done to simplify the act of saving the state of the system, but means the files are not replicable externally.

For convenience, the session files used when testing are provided with the source code in the repository for this work, and linked to in **Table 15**.

Table 15: FleetBench test case session file access locations.

| | |
|--------|--------------------------|
| Case 1 | [link to session file 1] |
| Case 2 | [link to session file 2] |
| Case 3 | [link to session file 3] |

APPENDIX D:

MAP FILE CREATION AND FORMAT

The creation of map files is done using the second application, GraphRendering. GraphRendering is a modified version of the software Tile Basic, which is distributed as part of the GNU General Public License. To comply with the license requirements, a copy of the license is provided in the code repository, and the modified application is distributed under the same license.

The original application, Tile Basic, was altered to remove features which allow inserting custom tile images. The options were replaced with specific images which can be used in the creation of a graph using the GraphRendering/Tile Basic UI. The underlying data structure and file output was modified in order to expose the relevant data about node/tile types as well as edge connections which are specific to each tile image.

This application is not meant to be extended but source code is provided as part of the manuscript:

An installer is also provided for convenience:

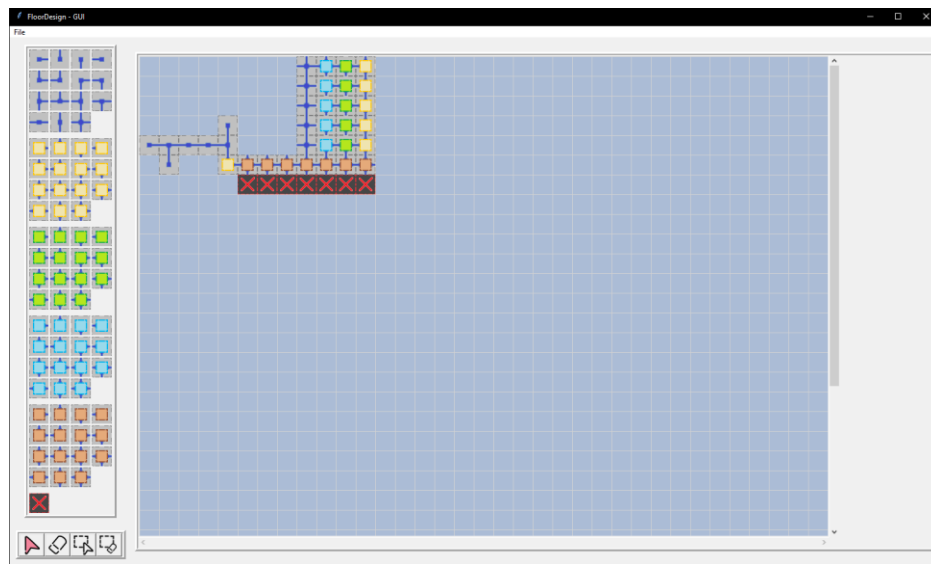


Figure 20: Map creation process in GraphRendering.

The process of creating a map in GraphRendering is simple and intuitive, if a bit tedious. First, define the map boundaries by creating a new map. To begin designing, select the desired painting tool from the bottom left menu. From left to right, shown in **Figure 20**, the options are paint single tile, erase single tile, paint boxed selection, and erase boxed selection. Next, select the desired tile from the left-hand panel. Begin drawing the tile map,

minding connections between nodes shown by the tile images. Edges which connect only in one direction will be ignored by FleetBench, so they are safe to leave in place.

Saving a map is done via File->Save Map.

Primarily, these map files are generated by GraphRendering, but for a large map the process can be very manual and tedious. Providing the file format is intended to allow a developer to process their map files into a known acceptable format in an automated fashion. When done this way, if the map is not intended to be opened in GraphRendering but rather only in FleetBench, it is acceptable to omit the image path and style ID as these are ignored values in FleetBench and really only local files for GraphRendering.

As an example, the map file for a 2-by-2 map is provided, composed of four nodes each with four edges exiting the node. The top left is an open node, top right is a rest node, bottom left is a delivery node, and bottom right is a pickup node. This configuration is shown in **Figure 21**.

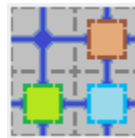


Figure 21: An example of a small map created in GraphRendering.

The saved file which represents this data is readable in plain text:

```

1: [
2:   {
3:     "mapDimensions": {
4:       "Xdim": 2,
5:       "Ydim": 2
6:     }
7:   },
8:   {
9:     "nodePosition": {
10:      "X": 0,
11:      "Y": 0
12:    },
13:    "nodeStylePath":
14:      "x:\\Github\\GraphRendering\\GUI_items\\tileSelectorDefaultImages\\04_edgeNSWE.png",
15:    "nodeStyleID": 14,
16:    "nodeType": "edge",
17:    "nodeEdges": {
18:      "N": 1,
19:      "W": 1,
20:      "S": 1,
21:      "E": 1
22:    }
23:   },
24:   {
25:     "nodePosition": {
26:      "X": 0,
27:      "Y": 1

```

```

28:   "nodeStylePath":
    "x:\\GitHub\\GraphRendering\\GUI_items\\tileSelectorDefaultImages\\24_depositNSWE.png",
29:   "nodeStyleID": 44,
30:   "nodeType": "deposit",
31:   "nodeEdges": {
32:     "N": 1,
33:     "W": 1,
34:     "S": 1,
35:     "E": 1
36:   }
37: },
38: {
39:   "nodePosition": {
40:     "X": 1,
41:     "Y": 0
42:   },
43:   "nodeStylePath":
    "x:\\GitHub\\GraphRendering\\GUI_items\\tileSelectorDefaultImages\\44_restNSWE.png",
44:   "nodeStyleID": 74,
45:   "nodeType": "rest",
46:   "nodeEdges": {
47:     "N": 1,
48:     "W": 1,
49:     "S": 1,
50:     "E": 1
51:   }
52: },
53: {
54:   "nodePosition": {
55:     "X": 1,
56:     "Y": 1
57:   },
58:   "nodeStylePath":
    "x:\\GitHub\\GraphRendering\\GUI_items\\tileSelectorDefaultImages\\34_pickupNSWE.png",
59:   "nodeStyleID": 59,
60:   "nodeType": "pickup",
61:   "nodeEdges": {
62:     "N": 1,
63:     "W": 1,
64:     "S": 1,
65:     "E": 1
66:   }
67: }
68:]

```

The first section provides general information about the map file, namely its overall dimension (lines 2-7).

Afterwards, each node is assigned data including its position in the grid, the image displayed in GraphRendering when the file is opened, the internal ID of the specific node type, the descriptive node type string, and which edges (out of four possible cardinal directions) are connected.

Loading to FleetBench

The critical information which FleetBench uses when replicating the information stored in the map file in its own data structures are the nodePosition (zero-indexed integer values), nodeType (edge, pickup, deposit, rest, charge), and nodeEdges (True or False, per

direction) fields. Properly processed, these data yield a description of a graph which FleetBench is able to load into the NetworkX library graph structure.

The NetworkX graph implementation is a fast, lightweight solution for storing graphs which also contain arbitrary data. More information on the NetworkX library may be found in its documentation [33].

The json library is used to load in all the data for processing.

FleetBench first uses the overall map dimension data to define the TkInter canvas space which will be rendered for user interaction in the relevant windows.

Then, one at a time, each node is processed to create new nodes in the NetworkX graph object. The nodes are represented with a string-ified tuple of the (X, Y) position coordinates. Use of a string avoids reference collisions in the future. Data regarding type, position (as integer values) and edge connections is stored directly in the graph for future reference.

Once all nodes have been added, FleetBench iterates over each node to validate and insert edges for nodes which bidirectionally connect to each other. For example, a node at (0, 0) is connected to a node at (0, 1) only if the first node has an edge leading south and the second node has an edge leading north according to the map file.

When all nodes and edges have been added, the structure is complete and can be rendered.

APPENDIX E:

TASK SCHEDULE CREATION AND FORMAT

Creation of task schedules is currently limited but very simple. The generation is done within FleetBench’s simulation configuration menu, under the Task Generation configuration options tab. By choosing “Create a Task Schedule” the screen shown in **Figure 22** is brought up.

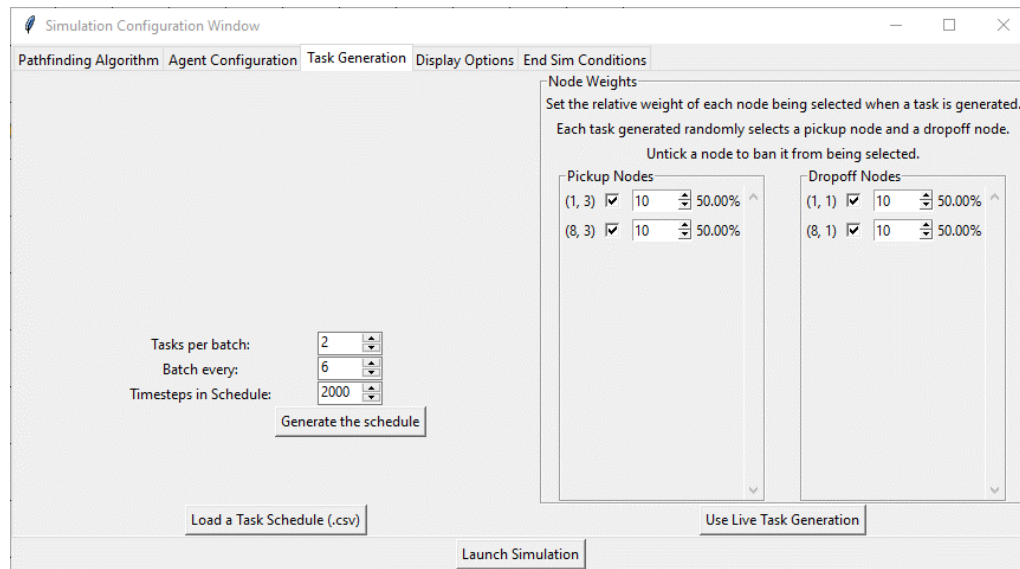


Figure 22: Task schedule generation window in FleetBench.

Tasks are generated in batches to be released into the simulation system at a regular intervals defined by the “Batch every” option (default value is the number of timesteps required in the average possible task). The user is able to customize the number of tasks per batch (default value is the number of agents in the system) and the time period over which task batches should be inserted into the system.

In the panel on the right, the user can untick nodes which should not be considered as possible pickup or delivery nodes, as well as adjust the relative weighting values of each node being selected during generation. The percentage value represents the calculated odds of a particular node being chosen per task generation.

The resulting comma-separated value file produced when the schedule is generated is composed of a header row followed by one row per task defined. As an example, the first five tasks in the schedule for test case 1 of Chapter 5 in this text are provided, transposed into a table with the same header row:

Table 16: Example task schedule file.

| PickupNode | DropoffNode | TimeLimit | ReleaseTime | Name |
|------------|-------------|-----------|-------------|------|
| (4, 0) | (0, 1) | 0 | 0 | 0 |
| (5, 2) | (0, 1) | 0 | 0 | 1 |
| (5, 0) | (0, 1) | 0 | 0 | 2 |
| (5, 2) | (0, 1) | 0 | 5 | 3 |
| (2, 2) | (0, 1) | 0 | 5 | 4 |

All tasks are defined with an infinite time limit as the solving mode for FleetBench multi-agent problems with schedules currently assumes the objective is to complete all tasks eventually. Statistical evaluation of tardiness is left to the data collected by FleetBench, discussed in Chapter 5. User extensions of FleetBench could use this value with custom schedules to determine a priority order for task assignment and completion.

The ReleaseTime value determines the timestep at which the task should be inserted into the simulation. This delay is used to smooth out the insertion of tasks to better approximate a warehouse scenario where orders trickle in. More advanced schedule design could be added as a feature to FleetBench, but the design is simple enough to emulate with external scripting.

The Name field is used to provide a human-readable ID. Currently there is no option to configure this through FleetBench, but if task schedules are generated externally FleetBench will use this information in its UI.

APPENDIX F:

EXTENDING FLEETBENCH

FleetBench uses a relatively simple to implement system (the state machine presented in Chapter 3 and Appendix B) to drive a highly dynamic and complicated system toward a solution state. There are many emergent complexities which must be accounted for during the design of an algorithm. Handling these occurrences is the job of the developer. This appendix aims to provide context on the current required steps to add an algorithm to the program from scratch. This section assumes that the implementation makes use of all four available script modules, which are named *Managers*, *Taskers*, *Pathfinders*, and *Movers*.

Adding New Algorithms

Each package of algorithms is currently held together by two dictionaries declared when the simulation process initializes: `algorithmDict` and `algorithmConfigDict`. The former is statically defined while the latter contains instructions to retrieve set values from the configuration menus.

At time of writing, this process is very far from automatic. In the future FleetBench could be greatly improved by handling the insertion of a collection of scripts into a specific filepath. It is recommended to refer to the most up to date documentation which can be found for FleetBench to do so.

An example entry into the algorithm dictionary looks like:

```
1: "Windowed HCA* (WHCA*)": (WHCAstarPathfinder,
                             WHCAstarReserver, WHCAstarMover, WHCAstarTasker)
```

Where each element of the tuple is a class which has been imported at the top of the `simulationProcessor.py` file:

```
1: from pathfindScripts.WHCAstarPathfinder import WHCAstarPathfinder
2: from pathfindManagerScripts.WHCAstarReserver import WHCAstarReserver
3: from pathfindMoverScripts.WHCAstarMover import WHCAstarMover
4: from pathfindTaskerScripts.WHCAstarTasker import WHCAstarTasker
```

Each script must therefore be contained within a class to be imported. When the classes are initialized, many arguments are passed. It is possible to increase the amount of information made available to the objects by passing in additional references, but these updates must be made for all algorithms or caught with default handling for additional arguments.

The objects passed into the `init` function of each script module are collected in **Table 17** for reference, with a descriptive name in place of the declarative code.

Table 17: Script objects and input arguments.

| | |
|---|--|
| <p>Manager object—manages the shared information state. Mostly passed as a reference to other objects for use by developer of algorithm scripts. This object is: self.infoShareManager</p> | Reference to the map graph |
| <p>Tasker object—seeks and assigns tasks for agents during simulation runtime This object is: self.simulationTasker</p> | List of pickup nodes |
| | List of delivery nodes |
| | List of node weights |
| | List of node availabilities |
| | Reference to the simulation display canvas |
| | Reference to the underlying simulation graph |
| | Reference to the shared information manager object |
| | Reference to FleetBench’s internal agent manager |
| | Reference to FleetBench’s internal task manager |
| <p>Pathfinder object—used to find paths from start to goal during the pathfinding state of the simulation. Not immediately instantiated. This object is: self.agentActionAlgorithm</p> | Reference to the set of simulation config. settings |
| | The owning agent ID |
| | Reference to the simulation display canvas |
| | Reference to the underlying simulation graph |
| | The agent’s current node |
| | The agent’s target node |
| | A set of configuration options with user-determined values, specific to the particular algorithm in use. Also contains the collision behavior setting. |
| | Reference to the shared information manager object |
| | Reference to the agent object |
| <p>Agent Mover object—the final authority on whether a set of plans is valid. Also determines the action order by modifying the agent priority queue. This object is: .agentMovementManager</p> | Reference to the set of simulation config. settings |
| | Reference to the simulation display canvas |
| | Reference to the underlying simulation graph |
| | Reference to the shared information manager object |
| | Reference to FleetBench’s internal agent manager |
| | Reference to FleetBench’s internal task manager |
| | Reference to the simulation display canvas |

In combination with the mandatory function lists found in **Table 9**, **Table 10**, **Table 11**, and **Table 12**, the scripts are able to be inserted without issue. Updates will need to be made to the UI and configuration menu in a compliant fashion with the existing work in FleetBench.

The `algorithmConfigDict` object is a dictionary whose keys are the names of algorithms selectable from within the simulation configuration options. The corresponding value is a dictionary of all relevant configuration option settings which should be passed into the script modules. Currently, the pathfinder is the only module programmed to accept the dictionary, but the modification required to provide the data to other modules is trivial.

Adding Configuration Options

The relevant file for updating the configuration menu is `simulationConfigManager.py`. As a stopgap measure to prevent prospective developers from having to learn how to use TkInter's involved UI element generation, variable tracing, and entry validation techniques, a recursive function is used to generate the UI by reading a plaintext file containing a valid Python dictionary of UI element configurations.

There are currently three available input methods in FleetBench: *numericSpinbox*, *optionMenu*, and *checkButton*. Each will be generated in a nested fashion as a sub-option of the parent level UI element. In this way, the set of options will dynamically grow without needing adjustment at the TkInter level, leaving concerns about rendering to the window irrelevant to those extending the configuration options.

Each object type is packaged as a single dictionary nested inside the list in the plaintext file. As a template and example:

```

1: {
2:     "labelText": "Render Simulation?",
3:     "elementType": "checkButton",
4:     "elementDefault": True,
5:     "optionValue": self.renderPlaybackValue,
6:     "gridLoc": "auto",
7:     "elementData": <dependent on elementType>
8: }
```

The above snippet generates a *checkButton* UI widget at the top level (that is, not as a sub-option). The `labelText` refers to the printed label next to the option, informing the user what the element is for. The `elementType` must be chosen from the three available input methods. The `elementDefault` and `elementData` values are dependent on the element being generated. The `optionValue` refers to the variable name that will contain the value set by the user when interacting with the element. The `gridLoc` should be left to "auto" unless the user understands what is being done to place objects within TkInter's grid manager.

When defining a parent-level UI element, the `elementData` value must be a dictionary whose keys are irrelevant, but whose values are a list of all sub-option UI elements. For example:

```

1: {
2:     "labelText": "Render Simulation?",
3:     "elementType": "checkButton",
4:     "elementDefault": True,
5:     "optionValue": self.renderPlaybackValue,
6:     "gridLoc": "auto",
```

```

7: "elementData": {
8:   "subOpt1": [
9:     {
10:       "labelText": "Render New Sim Prep Step",
11:       "elementType": "checkButton",
12:       "elementDefault": False,
13:       "optionValue": self.renderNewSimStep,
14:       "gridLoc": "auto",
15:       "elementData": {
16:         "subOpt1": [
17:           {
18:             "labelText": "Display Time (ms):",
19:             "elementType": "numericSpinbox",
20:             "elementDefault": 300,
21:             "optionValue": self.renderNewSimStepTime,
22:             "gridLoc": "auto",
23:             "elementData": SPINBOX_DEFAULT_RANGE
24:           },
25:         ]
26:       }
27:     },
28:     ...

```

This snippet defines a `checkButton` which is labeled "RenderSimulation?" and is checked on by default. There is at least one child element, "subOpt1", which is also a `checkButton`. This child `checkButton` is labeled "Render New Sim Prep Step" and defaults to being checked off. Because the parent is on by default, the child element can be interacted with.

The child element also has its own child, referred to here as the grandchild element. The grandchild element is not interactable at first because of the child element being default checked off. It is a `spinbox` which validates all entries such that only numeric values are accepted. The `numericSpinbox` is labeled "Display Time (ms):" and has a default value of 300, with a range defined by `SPINBOX_DEFAULT_RANGE`. This code produces the elements boxed in red in **Figure 23**.

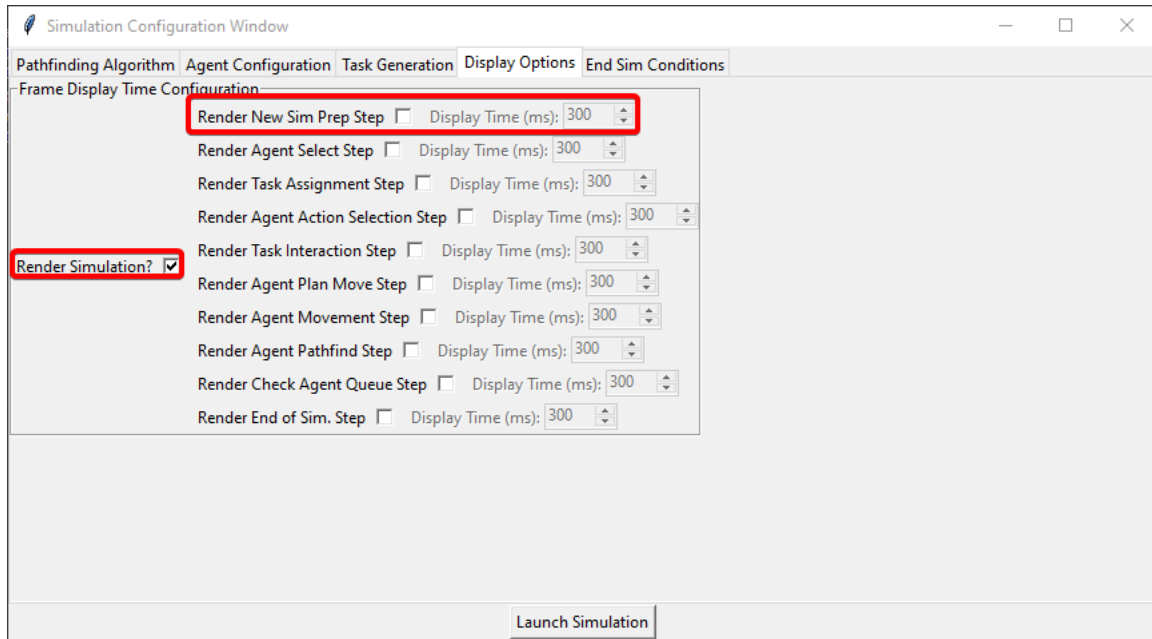


Figure 23: Screenshot of the labeled UI elements.

The optionMenu UI element is defined similarly to other parent UI elements, except that the child element name strings are used as selectable entries in the UI element. This means that now a user may select a value from an option menu and the irrelevant child elements are destroyed and the relevant options are created in their place. The child elements are again provided in a list:

```

1: {
2:   "labelText": "Pathfinding Algorithm",
3:   "elementType": "optionMenu",
4:   "elementDefault": "Single-agent A*",
5:   "optionValue": self.algorithmChoice,
6:   "elementData": {
7:     "Single-agent A*": [
8:       {
9:         "labelText": "Heuristic",
10:        "elementType": "optionMenu",
11:        "elementDefault": "Dijkstra",
12:        "optionValue": self.SAPFAstarHeuristic,
13:        "elementData": {
14:          "Dijkstra": None,
15:          "Manhattan": None,
16:          "Euclidean": None,
17:          "Approx. Euclidean": None
18:        }
19:      },
20:      {
21:        "labelText": "Heuristic Relaxation Coefficient",
22:        "elementType": "numericSpinbox",
23:        "elementDefault": 1,

```

```

24:             "optionValue": self.SAPFAstarHeuristicCoefficient,
25:             "elementData": (1, 50, 1)
26:         }
27:     ],

```

In this code snippet, an optionMenu is created with the label "Pathfinding Algorithm". The default value is identical to the first child option set in the elementData, and the selected option set is saved as a string value to a variable `self.algorithmChoice`.

The first choice, "Single-Agent A*", produces an option set comprised of another optionMenu and a numericSpinbox. The child optionMenu is not a parent UI element, and it is sufficient to define its children with the `None` singleton keyword. The resulting UI elements are shown in **Figure 24**.

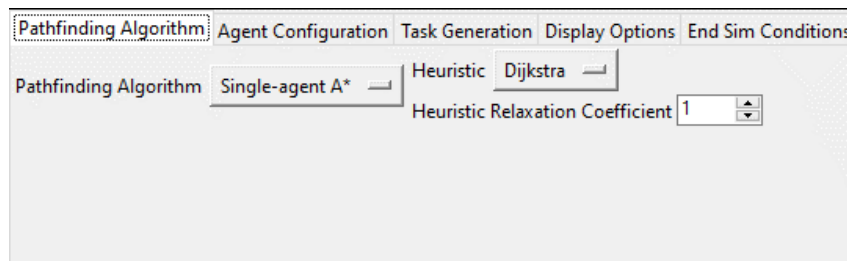


Figure 24: Screenshot of the generated optionMenu elements.

The final step in adding configuration options is to ensure that the named variables for each option are declared and passed into the simulation on launch. For each configuration tab there is a builder function, listed in **Table 18**.

Table 18: Configuration tabs and respective builder code.

| | | |
|-----------------------|---------------------------------|----------------------------|
| Pathfinding Algorithm | buildPathfindingAlgorithmPage() | simalgorithmmenuconfig.txt |
| Agent Configuration | buildAgentConfigurationPage() | simagentmenuconfig.txt |
| Display Options | buildDisplayOptionsPage() | simdisplaymenuconfig.txt |

In each case, the function reads in the corresponding plaintext files and executes the code. Therefore, the definition of variables must occur above the reading in of the file. For simplicity this can be done at the top of the function block. TkInter offers three variable types which may be used for this purpose, although the choice is often unimportant (the tcl interpreter underpinning TkInter stores all values as strings regardless). They are listed in **Table 19**.

Table 19: TkInter variable types.

| | |
|---------|---------------|
| Boolean | tk.BooleanVar |
| Integer | tk.IntVar |
| String | tk.StringVar |

In order for configuration variables to be passed into the simulation when the user launches with their configuration choices, the `packageSimulationConfiguration()` function is called. This function copies all values into a dictionary called `dataPackage` which is then returned as a single unit of data.

Here, a developer should group algorithm specific configuration options into the `datapackage` as a dictionary nested in the value of some key which identifies their algorithm. For example, the A* algorithm accepts configuration choices for its heuristic function and relaxation coefficients:

```

1: dataPackage["aStarPathfinderConfig"] = {}
2: dataPackage["aStarPathfinderConfig"]["algorithmSAPFAStarHeuristic"] =
    self.SAPFAStarHeuristic.get()
3: dataPackage["aStarPathfinderConfig"]["algorithmSAPFAStarHeuristicCoefficient"] =
    self.SAPFAStarHeuristicCoefficient.get()

```

Calling the `.get()` function on any TkInter value fetches the stored value from the Tk engine.

These configuration options will need to be referenced in the relevant declarations in the `simulationProcessor.py` file, which includes the `algorithmConfigDict`.

actually a customized extension of the TkInter native canvas object—is necessary to access the visualization functions and rendering engine.

This facet of the engine has received optimization. It was found early in testing that the thousands of objects being rendered to the canvas via requests to the tcl interpreter underpinning the TkInter implementation results in a permanent increase in memory consumption and objects which are tracked. Destroying the objects did not release memory or improve the performance of searching for objects in the space.

Currently, a list of objects is maintained which are automatically hidden instead of being destroyed. Objects which are called to be deleted are thus marked as available for use in some other manner. FleetBench handles the process of changing the appearance of old objects to suit the new purpose automatically, imposing no requirements on the user to consider performance except in the case that arbitrarily large amounts of objects are being created in a single render call.

The intended design pattern is that a script submits requests for objects to be rendered to the canvas via submission of information to a render queue in the canvas. This queue is then parsed if and only if the step which generated the requests was ticked to be rendered in the simulation configuration. If it is not, the render queue is cleared at the end of the step to minimize performance loss and prevent unintentional rendering.

The call structure for requesting a visual element always begins with a call to the canvas' `requestRender` method which must be passed three arguments:

- The first is the type of object being requested: *agent*, *highlight*, *canvasLine*, or *text*.
- The second is the action to be taken. Available actions depend on the object.
- The third is a dictionary of data describing what the object should look like, and other special information. The keys are predefined and must be adhered to in order to avoid failed executions of the rendering request.

An exception is recommended to be made for pathfinding. While identical functions could be used for both rendered and non-rendered pathfinding operations, pathfinding operations are called very often and produce many objects. A noticeable hit to performance may be experienced if sufficiently many requests are formatted and submitted to the render queue, only to be deleted.

Using tags, all objects are treated as layers such that sorting operations always yield a certain stacking order of objects. This provides a consistent visualization, but is an increasingly expensive operation as the number of objects increases.

The stacking order (lowest first) at time of writing is:

1. Highlight objects
2. Edge objects
3. Dangling edge objects
4. Node objects
5. Agent objects
6. CanvasLine objects
7. Text objects
8. InfoTile objects

Of these, only the *agent*, *highlight*, *canvasLine*, and *text* objects are available for user control. The others are expected to remain static throughout the session lifetime.

InfoTile objects are an invisible layer of graphics which use event bindings to capture mouse position. The relevant data under the cursor is then displayed in the top right of the canvas header, as shown in **Figure 26**.

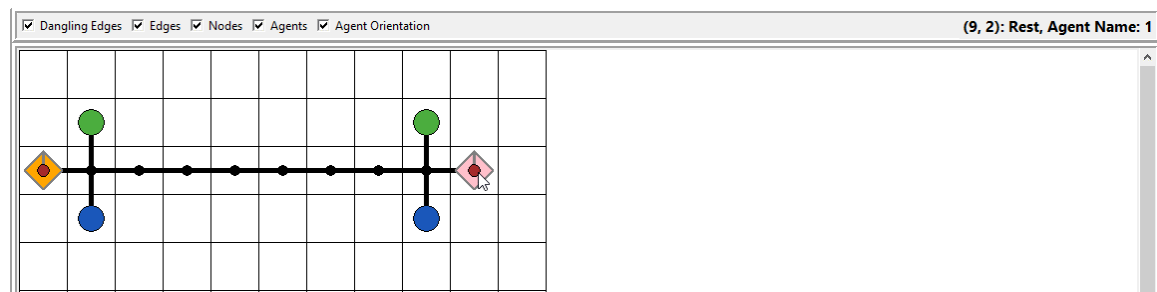


Figure 26: Screenshot of the FleetBench canvas and information bar.

The same layering system allows the user to toggle the visibility of objects using the controls found in the same bar as the mouse context text (**Figure 26**, left side).

Agent Objects

An agent “object” is actually a collection of shapes which make up the representation of an agent in FleetBench. There are three main objects which are treated as one single group: the body, the orientation indicator, and the window.

The body is drawn as a diamond with a fill color. The orientation indicator is a line extending from the center of the agent toward the direction it is “facing”. Currently this does not have any meaning in FleetBench for simulation purposes, but the groundwork for implementations which do manage rotation is in place. The window is a circle which reproduces the color of the node upon which the agent is situated, so that a user can identify the node despite an agent existing on it. **Figure 27** shows a rendered agent.

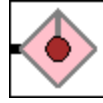


Figure 27: An agent rendered in FleetBench. The body fill color is pink, and the direction indicator points north. The node beneath the agent is brown. The node is edge-connected to some other node to the west.

When an agent object is created, it is automatically assigned identifying tags which make use of the agent's name and ID. In this way, submitting a reference to an agent ID will always access the correct set of objects.

There are five possible rendering actions to request pertaining to agent objects: *new*, *delete*, *clear*, *move*, and *rotate*. Each requires different data and will be presented independently.

New

The call format to request a new (or reuse an old) agent object is as follows:

```
1: .requestRender("agent", "new", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "position"
 - The string representation of a particular node at which the agent object should be drawn. This can be retrieved from the agent data's via the `currentPosition` property.
- "agentNumID"
 - The numeric ID of the agent this object is meant to represent. This is the internal number assigned by FleetBench, accessed via the agent data's `numID` property.

Optional key-value pairs may also be passed:

- "color"
 - The fill color which should be used when drawing the agent body. TkInter accepts a set of named colors as well as hex code colors. Default value is the agent's assigned fill color from a predefined list in the agent manager.
- "orientation"
 - The orientation which the orientation marker should be drawn to represent. Valid options are "N", "E", "W", "S". Default value is "N". This can be retrieved from the agent data's `orientation` property.

This method checks to see if there are unused agent objects in the system. If so, it calls `shiftAgentObjects` with the passed arguments to update the object and bring it back into use. If not, then `createAgent` is called with the same arguments.

Delete

The call format to request the deletion of (mark as unused and hide) an agent object is as follows:

```
1: .requestRender("agent", "delete", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "agentNumID"
 - The numeric ID of the agent this object is meant to represent. This is the internal number assigned by FleetBench, accessed via the agent data's `numID` property.

There are no valid optional arguments.

The method configures the agent object to be hidden and records it as unused, to be picked up and modified instead of creating a new agent object.

Clear

Clear is the more powerful version of delete, operating on all agents. The call format to request a clear (mark as unused and hide) of all agent objects is as follows:

```
1: .requestRender("agent", "clear")
```

No `renderData` object needs to be passed.

The method configures all agent objects to be hidden and recorded as unused, to be picked up and modified instead of creating a new agent object.

Move

The call format to request a movement of an agent object is as follows:

```
1: .requestRender("agent", "move", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "agentNumID"
 - The numeric ID of the agent this object is meant to represent. This is the internal number assigned by FleetBench, accessed via the agent data's `numID` property.
- "sourceNodeID"
 - The string representation of the node at which the agent object should be initially positioned. This can be retrieved from the agent data using the `currentPosition` property.
- "targetNodeID"
 - The string representation of the node to which the agent object should be moved.

There are no valid optional arguments.

The method executes a translation operation by subtracting the pixel coordinates of the target location from the source location, both calculated using the canvas grid aspect ratio. It also updates the color of the window object.

Rotate

The call format to request a rotation of an agent object is as follows:

```
1: .requestRender("agent", "rotate", renderData)
```

The renderData object is a dictionary which requires certain key-value pairs:

- "agentNumID"
 - The numeric ID of the agent this object is meant to represent. This is the internal number assigned by FleetBench, accessed via the agent data's numID property.
- "position"
 - The string representation of the node at which the agent object should be initially positioned. This can be retrieved from the agent data using the currentPosition property.
- "orientation"
 - The orientation which the orientation marker should be drawn to represent. Valid options are "N", "E", "W", "S". This can be retrieved from the agent data's orientation property.

There are no valid optional arguments.

The method executes a resizing of the orientation indicator about the center of the supplied position.

Highlight Objects

A highlight object refers to any translucent colored object which overlaps an entire tile on the gridded canvas. It appears behind all other objects in the display so as not to distort the color of other objects. The intended use is to indicate key nodes such as an agent's destination, path, or search activity. A sequence of arbitrary highlights using TkInter color codes "red", "orange", "yellow", "green", "blue", "indigo", and "violet" is shown in **Figure 28**.



Figure 28: A sequence of highlight objects in FleetBench.

There are four possible rendering actions to request pertaining to agent objects: *new*, *delete*, *clear*, and *move*. Each requires different data and will be presented independently.

New

The call format to request a new (or reuse an old) highlight object is as follows:

```
1: .requestRender("highlight", "new", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "targetNodeID"
 - The string representation of the node on which the highlight object should be drawn.
- "highlightType"
 - Primarily used for tagging active and inactive highlight objects. Valid options are "agentHighlight", "pickupHighlight", "depositHighlight", "pathfindHighlight", "miscHighlight", and "openSet".
- "multi"
 - Accepts a Boolean value. If True, the highlight is simply drawn. If false, all other highlights of the same type are deleted before this highlight is drawn.

Optional key-value pairs may also be passed:

- "color"
 - The fill color which should be used when drawing the highlight. TkInter accepts a set of named colors as well as hex code colors. Default value is determined by highlight type.
- "alpha"
 - The degree of opacity the highlight should have. The value must lie between zero and one. Larger means less transparent. Default value is 0.5.

This method checks to see if there are unused highlight objects in the system. If so, it calls `shiftHighlightObjects` with the passed arguments to update the object and bring it back into use. If not, then `requestHighlight` is called with the same arguments.

Delete

The call format to request deletion of all highlight objects of a certain type is as follows:

```
1: .requestRender("highlight", "delete", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "highlightType"
 - Primarily used for tagging active and inactive highlight objects. Valid options are "agentHighlight", "pickupHighlight", "depositHighlight", "pathfindHighlight", "miscHighlight", and "openSet".

There are no valid optional arguments.

All highlights of a particular type will be hidden from view and marked for reuse. There is currently no way to remove a highlight on a specific node only.

Clear

The call format to request deletion of all highlight objects is as follows:

```
1: .requestRender("highlight", "clear")
```

No renderData object needs to be passed.

The method configures all highlight objects to be hidden and recorded as unused, to be picked up and modified instead of creating a new highlight object.

Move

The call format to request a movement of a highlight object is as follows:

```
1: .requestRender("highlight", "move", renderData)
```

The renderData object is a dictionary which requires certain key-value pairs:

- "position"
 - The string representation of the node to which the highlight object should be moved.
- "highlightTag"
 - The string tag which should be searched for before moving the object. Not safe against selection of multiple objects

There are no valid optional arguments.

This method is not recommended to be used. Favor deletion and renewal.

Canvas Line Objects

A canvas line object is the term used to refer to an arrow drawn through some set of points on the canvas. Typically used to represent the path an agent takes through space, it accepts an unlimited number of nodes, automatically plotting turns and applying an arrow at the head of the line to indicate directionality.

A canvas line drawn in cyan with the point sequence $\{(0,0), (2,0), (2,1), (0,1)\}$ is shown in **Figure 29**.

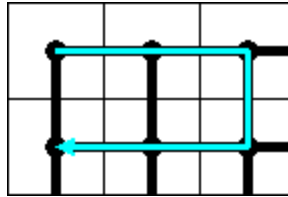


Figure 29: A canvas line object rendered in FleetBench.

When a `canvasLine` object is created, it is automatically assigned identifying tags which make use of the node path. In this way, submitting a reference to a path will always access the correct object, if it exists.

There are three possible rendering actions to request pertaining to `canvasLine` objects: *new*, *delete*, and *clear*. Each requires different data and will be presented independently.

New

The call format to request a new (or reuse an old) `canvasLine` object is as follows:

```
1: .requestRender("canvasLine", "new", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "nodePath"
 - The sequence of nodes through which the line should path. Accepts non-integer values, resulting in locations which are not the center of node representations. Minimum length of two nodes.
- "lineType"
 - Primarily used for tagging active and inactive highlight objects. Accepts any string value.

Optional key-value pairs may also be passed:

- "color"
 - The fill color which should be used when drawing the `canvasLine`. TkInter accepts a set of named colors as well as hex code colors. Default value is "white".
- "width"
 - The pixel width of the drawn arrow.

This method checks to see if there are unused `canvasLine` objects in the system. If so, it calls `shiftCanvasLineObjects` with the passed arguments to update the object and bring it back into use. If not, then `renderDirectionArrow` is called with the same arguments.

Delete

The call format to request deletion of a `canvasLine` object is as follows:

```
1: .requestRender("canvasLine", "delete", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "oldNodePath"
 - The path of the desired `canvasLine`. Accepted as a string or an iterable containing tuples of integers defining positions.

There are no valid optional arguments.

The method configures the `canvasLine` object to be hidden and records it as unused, to be picked up and modified instead of creating a new `canvasLine`.

Clear

The call format to request deletion of all `canvasLine` objects is as follows:

```
1: .requestRender("canvasLine", "clear")
```

No `renderData` object needs to be passed.

The method configures all `canvasLine` objects to be hidden and recorded as unused, to be picked up and modified instead of creating a new `canvasLine` object.

Text Objects

A text object is a piece of text rendered in a relative position within a node. It is not length safe or padded to avoid falling outside the rendered grid line of a node, so its use may be limited in certain cases. In existing algorithms, this is mostly used when performing searches through space to find paths to goals, displaying relevant scoring metrics and time depth data.

An example of a node with drawn text is given in **Figure 30**.

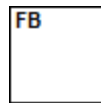


Figure 30: A piece of text rendered in the northwest corner of a tile in FleetBench.

When a piece of text is rendered in FleetBench it is automatically assigned tags corresponding to supplied type and position so that management can be done on type and positional bases.

The `font` argument is a special TkInter font wrapper. Generally this is not necessary, but if a user wishes to modify the appearance of their text at the font level, they will need to understand the TkFont specifications.

There are three possible rendering actions to request pertaining to text objects: *new*, *delete*, and *clear*. Each requires different data and will be presented independently.

New

The call format to request a new (or reuse an old) text object is as follows:

```
1: .requestRender("text", "new", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "position"
 - The string representation of the node at which the text object should be positioned.
- "text"
 - The string which should be rendered.
- "textType"
 - Primarily used for tagging text objects for future reference. Accepts any string value.

Optional key-value pairs may also be passed:

- "anchor"
 - String value defining the relative position to the center of the node's graphical representation at which the text should be rendered. Accepted values are "N", "E", "W", "S", "NW", "NE", "SW", "SE", and "CENTER". Default value is "NE".
- "textColor"
 - The fill color which should be used when drawing the text. TkInter accepts a set of named colors as well as hex code colors. Default value is "white".
- "textFont"
 - The TkFont specification to be used when rendering the text. Default is dependent on system, but is given bold weight.

This method checks to see if there are unused text objects in the system. If so, it calls `shiftTextObject` with the passed arguments to update the object and bring it back into use. If not, then `renderTileText` is called with the same arguments.

Delete

The call format to request deletion of all text objects of a certain type is as follows:

```
2: .requestRender("text", "delete", renderData)
```

The `renderData` object is a dictionary which requires certain key-value pairs:

- "highlightType"
 - The target text tag for deletion.

There are no valid optional arguments.

All text with a particular supplied tag will be hidden from view and marked for reuse. There is currently no way to remove text on a specific node only.

Clear

The call format to request deletion of all highlight objects is as follows:

```
1: .requestRender("text", "clear")
```

No renderData object needs to be passed.

The method configures all text objects to be hidden and recorded as unused, to be picked up and modified instead of creating a new text object.

APPENDIX H:

CONFLICT RESOLVER

This conflict resolver is applicable to any algorithm which uses a reservation table to check the availability of nodes in the simulation map. It is a simplistic implementation which is restricted to manipulations of the priority queue and requests for replanning of agent motions when collisions are found.

No attempt has been made to prove the algorithm is complete, but it has not yet failed to resolve a collision given sufficient time to find viable paths for agents. It is possible that in a sufficiently dense graph the routine may fail. In this case the simulation should crash, and other methods will need to be investigated. For now, the conflict resolver is used to adapt Multi-Agent Pathfinding algorithms to the Multi-Agent Pickup and Delivery problem, per the difficulties discussed in Chapter 2. Further research in this area is needed.

Conflict resolution is done immediately upon the detection of a collision, which occurs during the action execution state. Collision detection is done by parsing submitted action plans for the timestep into two dictionaries. The first is a dictionary of node keys where values are lists of the IDs of agents which intend to move into the node over the course of the timestep. Similarly, the second dictionary is composed of edge keys (a tuple of the two involved nodes sorted alphanumerically for consistency) where the values are lists of agent IDs planning to occupy those edges during the timestep.

If the length of any list in these dictionaries is greater than one, a conflict is going to occur when the actions are executed. When this occurs, the agents involved in the collision are conveniently already in the list, ready for resolution.

In the current implementation, the highest priority an agent can have (even superseding the current priority queue) is when it is in the “crashed” state. When the movements are checked at the end of the planning process for all agents, the agent is moved from its current position in the priority queue to the very top and its preferred move is planned. The preferred move is determined using the heuristic distance of the pathfinder.

The idea is that an agent trapped in a corner with one way out may be blocked in by the same (higher priority, else the agent would be planning to avoid the trapped agent) agent which is attempting to move into the trapped agent’s current node. In this case, the trapped agent will have crashed and planned to try and move away as it is the only option. This creates the swapping conflict described in Chapter 2, **Figure 3**. Because the only way to resolve this situation is to ensure that the trapped agent is able to make its way out of the location, it must be prioritized by the solver. Inconvenienced agents which can take other actions to advance are inefficient, but not entering a stalling or crashing state.

With this result, edge conflicts are given priority over vertex conflicts. The higher priority agent of the agents in the conflict maintains its priority. The lower priority agent is forced

to replan, either by resetting the agents pathfinder and activity state and re-entering the action planning loop of the state machine or by directly seeking the optimal viable move. The agent should be allowed to contest plans of agents which are of lower priority, skipping the need to enter a crashed state which would cause a possible infinite cycle of crashed agents attempting to move into crashed agents.

If there is a chain of collisions waiting to happen (for instance, a convoy of agents in a narrow hallway), the newly planned move may also result in a collision. At this point, the initially trapped agent's plan has already been cemented, preventing the new collision from resulting in a reflected collision back to the original agent. As a result, the final plans created defer to the most trapped agent's needs.

In the case of a vertex conflict, the situation is handled differently. Because agents experiencing node conflicts are not necessarily going to be forced to take a certain move, the priority of agents in these cases needs to be swapped to avoid cycling behaviors.

Deference is given to the highest priority agent involved in the collision for the current timestep. Immediately after this action is re-entered into the plan for the timestep however, the colliding agents have their priorities swapped. Frequently these agents will desire to remain in place, simply waiting for the other agent to move away. By swapping the agent priorities, the agent moving into the contested node will be prevented from trying to plan a path into the node of the deferring agent on the next timestep.

This could produce cycling behavior. However, in open spaces conflicts should not happen to begin with. In constrained spaces the situation should eventually resolve into a position where one agent is entirely trapped. This then shifts from a vertex conflict situation to an edge conflict situation, which resolves as previously discussed.

This algorithm for resolving the collisions is implemented in the mover scripts of algorithms in the WHCA* family of MAPF solutions using the functions `comprehendAgentMotions` (builds out the dictionaries of motion which are evaluated for conflicts), `checkForConflicts` (evaluates the motion dictionaries), `resolveEdgeConflicts` (implementing the described edge conflict resolution strategy), and `resolveNodeConflicts` (implementing the described node conflict resolution strategy). The current implementation directly mutates the state of the agents and immediately seeks paths to the goal which avoid further collisions, but there is no reason the behavior cannot be adjusted to make use of the state machine for the replanning procedure.

This strategy heavily utilizes the reservation table, which is adapted slightly from the baseline Cooperative A* implementation to mark off spacetime regions using the agents ID rather than simply declaring that a particular motion plan is possible or not. This allows agents to determine if they may overwrite other agent's plans by comparing their IDs and placements in the current priority list.

Further, the reservation table needs to be managed in a manner which allows path reservations to be removed from the table as agents alter their plans.

REFERENCES

- [1] “Industrial robots,” KUKA AG. Accessed: Nov. 05, 2023. [Online]. Available: <https://www.kuka.com/en-us/products/robotics-systems/industrial-robots>
- [2] mars.nasa.gov, “Summary | Rover,” NASA Mars Exploration. Accessed: Nov. 05, 2023. [Online]. Available: <https://mars.nasa.gov/msl/spacecraft/rover/summary>
- [3] O. Salzman and R. Stern, “Research Challenges and Opportunities in Multi-Agent Path Finding and Multi-Agent Pickup and Delivery Problems,” *N. Z.*, 2020.
- [4] B. van der List, “Boston Dynamics wants to change the world with its state-of-the-art robots,” *Strategy+business*. Accessed: Nov. 05, 2023. [Online]. Available: <https://www.strategy-business.com/article/Boston-Dynamics-wants-to-change-the-world-with-its-state-of-the-art-robots>
- [5] J. Dorrier, “Agility’s New Factory Can Crank Out 10,000 Humanoid Robots a Year,” *Singularity Hub*. Accessed: Nov. 05, 2023. [Online]. Available: <https://singularityhub.com/2023/09/20/agilitys-new-factory-can-crank-out-10000-humanoid-robots-a-year/>
- [6] “How Amazon deploys robots in its operations facilities.” Accessed: Nov. 05, 2023. [Online]. Available: <https://www.aboutamazon.com/news/operations/how-amazon-deploys-robots-in-its-operations-facilities>
- [7] “Automation, robotics, and the factory of the future | McKinsey.” Accessed: Nov. 05, 2023. [Online]. Available: <https://www.mckinsey.com/capabilities/operations/our-insights/automation-robotics-and-the-factory-of-the-future>
- [8] J. Prisco, “Why online supermarket Ocado wants to take the human touch out of groceries,” *CNN*. Accessed: Nov. 05, 2023. [Online]. Available: <https://www.cnn.com/2021/04/26/world/ocado-supermarket-robot-warehouse-spc-intl/index.html>
- [9] O. Majerech, “Solving Algorithms for Multi-agent Path Planning with Dynamic Obstacles”.
- [10] D. Šišlák, P. Volf, and M. Pěchouček, “Agent-Based Cooperative Decentralized Airplane-Collision Avoidance,” *IEEE Trans. Intell. Transp. Syst.*, vol. 12, no. 1, pp. 36–46, Mar. 2011, doi: 10.1109/TITS.2010.2057246.
- [11] R. Stern *et al.*, “Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks,” *Proc. Int. Symp. Comb. Search*, vol. 10, no. 1, pp. 151–158, Sep. 2021, doi: 10.1609/socs.v10i1.18510.
- [12] H. Ma, J. Li, T. K. S. Kumar, and S. Koenig, “Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks.” *arXiv*, May 30, 2017. Accessed: Nov. 05, 2023. [Online]. Available: <http://arxiv.org/abs/1705.10868>
- [13] A. Prorok, M. Malencia, L. Carlone, G. S. Sukhatme, B. M. Sadler, and V. Kumar, “Beyond Robustness: A Taxonomy of Approaches towards Resilient Multi-Robot Systems.” *arXiv*, Sep. 25, 2021. Accessed: Nov. 05, 2023. [Online]. Available: <http://arxiv.org/abs/2109.12343>
- [14] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths,” *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968, doi: 10.1109/TSSC.1968.300136.

- [15] G. Sharon, R. Stern, A. Felner, and N. Sturtevant, "Conflict-Based Search For Optimal Multi-Agent Path Finding," *Proc. AAAI Conf. Artif. Intell.*, vol. 26, no. 1, Art. no. 1, 2012, doi: 10.1609/aaai.v26i1.8140.
- [16] G. Sartoretti *et al.*, "PRIMAL: Pathfinding via Reinforcement and Imitation Multi-Agent Learning," *IEEE Robot. Autom. Lett.*, vol. 4, no. 3, pp. 2378–2385, Jul. 2019, doi: 10.1109/LRA.2019.2903261.
- [17] R. Luna and K. E. Bekris, "Efficient and complete centralized multi-robot path planning," in *2011 IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Francisco, CA: IEEE, Sep. 2011, pp. 3268–3275. doi: 10.1109/IROS.2011.6095085.
- [18] D. Silver, "Cooperative Pathfinding," *Proc. AAAI Conf. Artif. Intell. Interact. Digit. Entertain.*, vol. 1, no. 1, pp. 117–122, Sep. 2021, doi: 10.1609/aiide.v1i1.18726.
- [19] K.-H. C. Wang and A. Botea, "MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees".
- [20] J. Kottinger, S. Almagor, and M. Lahijanian, "Conflict-Based Search for Explainable Multi-Agent Path Finding," *Proc. Int. Conf. Autom. Plan. Sched.*, vol. 32, pp. 692–700, Jun. 2022, doi: 10.1609/icaps.v32i1.19859.
- [21] P. Surynek, "A novel approach to path planning for multiple robots in bi-connected graphs," in *2009 IEEE International Conference on Robotics and Automation*, Kobe: IEEE, May 2009, pp. 3613–3619. doi: 10.1109/ROBOT.2009.5152326.
- [22] N. Franke, "gdmappf." Sep. 14, 2023. Accessed: Nov. 05, 2023. [Online]. Available: <https://github.com/nathanfranke/gdmappf>
- [23] K. Okumura, "mapf-IR." Nov. 01, 2023. Accessed: Nov. 05, 2023. [Online]. Available: <https://github.com/Kei18/mapf-IR>
- [24] E. Lam, "BCP-MAPF." Oct. 13, 2023. Accessed: Nov. 05, 2023. [Online]. Available: <https://github.com/ed-lam/bcp-mapf>
- [25] A. Bose, "Multi-Agent path planning in Python." Nov. 04, 2023. Accessed: Nov. 05, 2023. [Online]. Available: https://github.com/atb033/multi_agent_path_planning
- [26] H. Peng, "Multi-Agent Path Finding." Nov. 05, 2023. Accessed: Nov. 05, 2023. [Online]. Available: <https://github.com/GavinPHR/Multi-Agent-Path-Finding>
- [27] "Tile Basic - About." Accessed: Nov. 19, 2023. [Online]. Available: <https://multilingual-coder.github.io/tilebasic/about.html>
- [28] W. Hoenig *et al.*, "Multi-Agent Path Finding with Kinematic Constraints," *Proc. Int. Conf. Autom. Plan. Sched.*, vol. 26, pp. 477–485, Mar. 2016, doi: 10.1609/icaps.v26i1.13796.
- [29] R. Dechter and J. Pearl, "Generalized best-first search strategies and the optimality of A*," *J. ACM*, vol. 32, no. 3, pp. 505–536, Jul. 1985, doi: 10.1145/3828.3830.
- [30] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Trans. Robot.*, vol. 21, no. 3, pp. 354–363, Jun. 2005, doi: 10.1109/TRO.2004.838026.
- [31] S. Koenig and M. Likhachev, "D* Lite." [Online]. Available: <https://cdn.aaai.org/AAAI/2002/AAAI02-072.pdf>

- [32] F. Grenouilleau, W.-J. V. Hoeve, and J. N. Hooker, “A Multi-Label A* Algorithm for Multi-Agent Pathfinding,” *Proc. Int. Conf. Autom. Plan. Sched.*, vol. 29, pp. 181–185, May 2021, doi: 10.1609/icaps.v29i1.3474.
- [33] “Reference — NetworkX 3.2.1 documentation.” Accessed: Nov. 25, 2023. [Online]. Available: <https://networkx.org/documentation/stable/reference/index.html>