TABLE OF CONTENTS

Page

Chapter

[1. Background 3](#_Toc151089808)

[Motivation 7](#_Toc151089809)

[2. Problem Definitions and Solutions 12](#_Toc151089810)

[Defining Multi-Agent Problems 13](#_Toc151089811)

[Basic Approaches 18](#_Toc151089812)

[The A\* Algorithm 18](#_Toc151089813)

[LRA\* 23](#_Toc151089814)

[Multi-Agent Approaches 25](#_Toc151089815)

[Windowed Hierarchical Cooperative A\* 25](#_Toc151089816)

[Cooperative A\* 26](#_Toc151089817)

[Hierarchical Cooperative A\* 28](#_Toc151089818)

[Windowed Hierarchical Cooperative A\* 31](#_Toc151089819)

[Token Passing 33](#_Toc151089820)

[Well-formed Problems and Completeness Guarantees 33](#_Toc151089821)

[Token Passing 35](#_Toc151089822)

[Token Passing with Task Swaps 38](#_Toc151089823)

[3. Generalizable Approach 42](#_Toc151089824)

[Common Behaviors 43](#_Toc151089825)

[Simulation Definition 45](#_Toc151089826)

[New Timestep 46](#_Toc151089827)

[Task Management 47](#_Toc151089828)

[Action Planning 48](#_Toc151089829)

[Action Execution 50](#_Toc151089830)

[End of Step 51](#_Toc151089831)

[4. Framework Application 52](#_Toc151089832)

[WHCA\* 53](#_Toc151089833)

[TPTS 56](#_Toc151089834)

[5. Implementation, Testing, and Results 63](#_Toc151089835)

[Design of Experiment 67](#_Toc151089836)

[Case 1 68](#_Toc151089837)

[Case 2 69](#_Toc151089838)

[Case 3 71](#_Toc151089839)

[Results 72](#_Toc151089840)

[Case 1 75](#_Toc151089841)

[Case 2 77](#_Toc151089842)

[Case 3 79](#_Toc151089843)

[6. Conclusion 82](#_Toc151089844)

[Future Work 83](#_Toc151089845)

## 1. Background

Mobile robotics is a rapidly maturing field with a wide range of applications including industrial settings [1], space exploration [2], and medical applications [3]. Significant advancements have been made in robot capabilities, manufacturing costs, and safety to enable the integration of robots in a growing number of settings.

One setting has garnered particular attention in recent years, forming 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 up to 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 such automated systems in fulfillment centers [8].

A key problem faced in warehouse robotics is that of finding appropriate paths and trajectories of 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: videogames [9], search problems including rescue and evacuation [10], and air-traffic management [11]. This problem has received significant attention from both robotics and artificial intelligence researchers, becoming well characterized but remaining definitively unsolved [10].

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 [12]. 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 [13].

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. 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 [14]. 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.

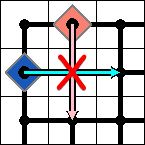


Figure 1: An instance in which two agents who have found best paths to target nodes using single-agent techniques would collide, rendered in FleetBench.

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 via the famous A\* algorithm [15]. This approach cannot be readily 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. More complex strategies are needed.

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

Centralized approaches such as CBS [16], PRIMAL [17], and Push and Swap [18] 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) [18]. However, this approach is computationally intense, even for small scenarios, and is simply not scalable to larger applications [10]. If computation time begins to exceed action time systems will begin to lag, resulting in cascading underperformance as additional objectives are introduced. Advancements in this area often involve trying to prune the search space [18], reducing search time.

Decoupled approaches such as Windowed Hierarchical Cooperative A\* [19] and Token Passing [13] 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 at large agent counts—the solution found cannot be guaranteed to be optimal, or even complete unless certain preconditions are met [13]. 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 [20]. In these cases, it is difficult to justify implementation in real-world systems as-is.

### 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. This problem is especially difficult for those not already well-versed in the field and may act as a deterrent for industry implementation, hobbyist engagement, or 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, and with no clear predictor that can identify when this 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 [12]. 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 all 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 time.

In an effort to bridge these gaps between theoretical knowledge and practical test implementation, Chapter ? 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 dynamic and flexible, while anchoring key facets present in most algorithms. With this approach, algorithms of any type may be implemented into one coherent system, ensuring ease of test and iteration. This represents the primary intent of the manuscript.

As proof of concept, Chapter ?? presents a pair of programs which implement the strategy for a selection of algorithms which exist in research. The first novel program, called GraphRendering is a utility for generating the system maps over which a fleet of agents operates. 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. Layers of abstraction are used to generalize the problem, so that with careful planning most situations may be represented. 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. An auxiliary program is also provided, which allows for rapid graphical generation of maps.

Together, these two applications create a workflow for a user to rapidly test algorithms on a variety of test cases with various 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 ??? as evidence that the approach and application are useful for analysis.

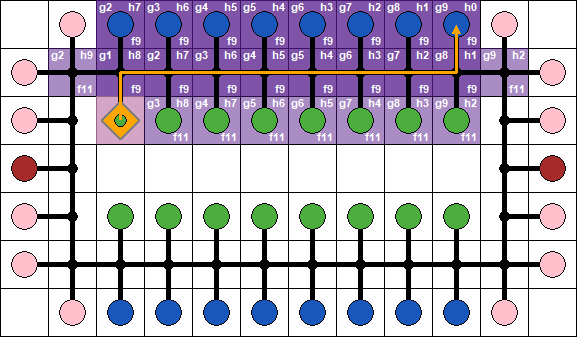


Figure 2: An agent seeks a path to a node through an abstraction of space using the A\* algorithm, rendered in FleetBench. The map represents an MAPD situation, where blue nodes are for pickup, and green are for deposit.

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

## 2. Problem Definitions and Solutions

Applications benefiting from 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 [27].

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 using the model. Finding the shortest path, among other problems of graph traversal, 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 (also 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. Otherwise, it is said to be undirected. An edge may also have a weight, or cost, associated with it, possibly representing the 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 composed of a set of vertices and a set of edges . 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:

where

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 agents contains information about how many agents are in the problem’s system and their individual properties. The task set contains 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 and are used to define the positions of all agents and tasks. In this way, a multi-agent problem is defined:

where

* ,
* .

With a formal expression of the problem, it becomes possible to formally express conflicts. Figure 1 shows the two basest cases of conflict possible in this construction of the MAPF and MAPD problems: 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 causes an undesired collision.

Further conflicts exist but in general there is no unified requirement for certain conflicts to be observed throughout the literature [12].

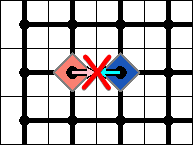
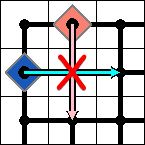


Figure 1: Two conflicts agents may experience when moving within a multi-agent problem instance, rendered in FleetBench. Left: Vertex conflict. Right: Edge conflict.

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 for any given node at any timestep is given as .

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

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:

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 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 objective or 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. 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.

#### The A\* Algorithm

One of the most useful results in the study of graph traversal is the A\* algorithm [15]. 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 [28]. 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.

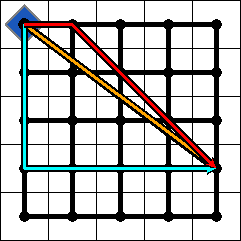


Figure 2: Three different measures of distances on a grid rendered in FleetBench: Euclidean (orange), Manhattan (cyan), and Chebyshev (red).

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. This is not always the best possible path, 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:

* returns an estimate of the distance between nodes and , determined by using an appropriate heuristic function.
* returns the set of nodes which are connected to node via an edge. This function may be augmented to include the node in the returned values if an agent may be interested in waiting in the same position.
* returns the edge cost of a movement from to , which in many cases is equivalent to the function’s result but may not always be. With the assumption that all edge costs are the same this function does not perform any calculation, but it may easily be augmented.
* returns a Boolean *True* value if the input node is the goal node, and Boolean *False* otherwise.
* is used when the goal node is explored by the algorithm. Using saved *g-Score* data, the optimal path to from 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.

|  |  |  |
| --- | --- | --- |
| **Algorithm 1** A\* | | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **if** IsGoal(: |
|  | BuildPath( |
|  | **return** |
|  | **for** : |
|  | Cost |
|  | **if** |
|  |  |
|  |  |
|  |  |
|  | **if** **not in** |
|  |  |
|  | **return** |

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 the 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 list of all nodes which share an edge with the current node, which are called neighbors.

On lines 13-20, for each neighbor a *g-Score* is calculated which combines 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 open set is empty (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 [29]. Ultimately, however, the choice of path planning algorithm is a degree of freedom for the designer and does not significantly impact the procedures outlined in Chapter ?.

#### LRA\*

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 Y.

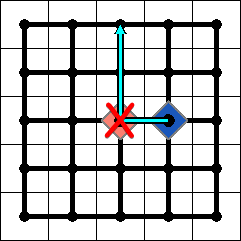
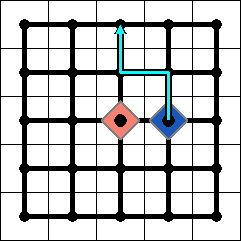
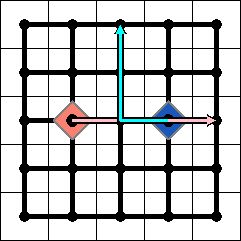


Figure 3: The LRA\* replanning procedure. On the left, two agents plan optimistic paths to their goals. A collision is detected, and the lower priority agent (blue) plans a new route, avoiding the immediate neighbor.

A simple approach, which is widely used in the videogame industry, is to allow A\* to execute until a collision is inevitable [19]. 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. To do this it needs access to the system state at the current timestep to evaluate the position of other agents in the system: . The updated function is described in Algorithm 2.

|  |  |
| --- | --- |
| **Algorithm 2** Neighbors function for LRA\* | |
|  |  |
|  |  |
|  | Neighbors(): |
|  | **if** Position |
|  |  |
|  | **return** |

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 [19]. 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 [19]. 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 [13].

These approaches will be adapted via the procedure laid out in Chapter ? as examples of the methodology so as to provide points of comparison. Their usage within the framework laid out in Chapter ? to produce experimental data in Chapter ?? will reveal each approach’s advantages, restrictions, and demonstrate the value of this work.

#### 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” [19]. 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.

See Appendix ??? for details on the algorithm’s implementation within FleetBench.

##### 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 keep track 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 which is accessible 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** Cooperative A\* (CA\*) | | |
|  | **function** |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **if** IsGoal(: |
|  | BuildPath( |
|  |  |
|  | **return** |
|  | **for** FreeNeighbors |
|  | Cost |
|  | **if** |
|  |  |
|  |  |
|  |  |
|  | **if** **not in** |
|  |  |
|  | **return** |
|  | **function** FreeNeighbors |
|  |  |
|  | **for** Neighbors: |
|  | **if** ( |
|  |  |
|  | **return** |
|  | **function** Replan |
|  |  |
|  | CAStar |
|  | **return** |

##### 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 as 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. 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 agents 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.

In combination, 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 is the goal. 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** Reverse-Resumable A\* (RRA\*) | | |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | **if** |
|  | **return** |
|  |  |
|  |  |
|  |  |
|  | **if** IsGoal(: |
|  | **return** |
|  | **for** : |
|  | Cost |
|  | **if** |
|  |  |
|  |  |
|  |  |
|  | **if** **not in** |
|  |  |
|  | **return** |

##### 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 searching time 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 [19]. 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 , 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 for WHCA\* | |
|  | Finished-WHCA |
|  | **return** |

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 there is some path to the goal. Effectively, this means that 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. The general collision resolution strategy employed in this case is described in Appendix ?.

#### 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, the agents 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 [13]. 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:

* Vendpoints, which contains all nodes satisfying the endpoint definition.
* Vtask, which contains all pickup and delivery nodes.
* Vntask, 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. Vntask must contain at least as many non-task 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 the related algorithms, presented in the next sections. Optimality of solutions is not guaranteed, as greedier algorithms may find faster solutions while sacrificing the completeness guarantee. Figure 4 presents three instances of an MAPD problem which demonstrate the second and third conditions. 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. The middle instance is not well-formed as there are fewer non-task endpoints than there are agents which violates the second requirement. The instance on the right is also not well-formed because the path to the delivery endpoint from a pickup endpoint is required to cross over an endpoint, violating the third requirement.

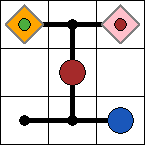
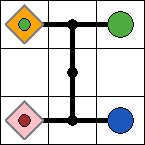
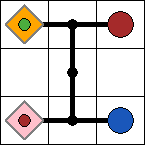


Figure 4: Three MAPD instances. Green circles indicated pickup endpoints. Blue circles indicate delivery endpoints. Brown nodes indicate non-task endpoints. Two agents, shown as diamonds, are in the problem space.

##### 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 an agent’s endpoint 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 or delivery node are the end of another agent’s path in the token may be considered a valid assignment.

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 as .

The primary benefit of this preprocessing is that it becomes trivial to select the task with the nearest pickup location, 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 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 Task | | |
|  | **function** BestTask |
|  |  |
|  | **return** |

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, the trivial path denoted ***Stay*** is found, and the agent waits in its current position. ***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 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 , 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 placeso long as its current position is not the endpoint of another task in the system to avoid future blockages (lines 12-13). Otherwise, 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** Token Passing (TP) | | |
|  | Preprocess |
|  |  |
|  | **while** |
|  | Update |
|  | **for** |
|  |  |
|  | **if** |
|  |  |
|  | Assign to |
|  |  |
|  | Path1 |
|  | **else if** |
|  | Stay |
|  | **else**: |
|  | Path2 |
|  | All agents execute plans; |
|  | **function** Path1 |
|  | **if**  reached |
|  | CAStar |
|  | **else**: |
|  | CAStar |
|  | **function** Path2 |
|  |  |
|  | CAStar |
|  | **function** Stay |
|  |  |
|  |  |

##### 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 may be better suited to complete task than , but if is not able to make its way to a free endpoint from its current position, the task swap should fail and should be allowed to continue with its previously planned actions. This process is represented by the ***GetTask*** procedure in Algorithm 8.

|  |  |  |
| --- | --- | --- |
| **Algorithm 8** Token Passing with Task Swaps (TPTS) | | |
|  | Preprocess |
|  |  |
|  | **while** |
|  | Update |
|  | **for** |
|  | GetTask |
|  | All agents execute plans; |
|  | **function** GetTask |
|  |  |
|  | **while** |
|  |  |
|  |  |
|  | **if** no agent assigned to |
|  | Assign to |
|  | Path1 |
|  | **return** |
|  | **else**: |
|  |  |
|  | agent assigned to |
|  | Path |
|  | Unassign from ; Remove from |
|  | Path1 |
|  | **if** : |
|  | GetTask |
|  | **if** : |
|  | **return** |
|  | **else**: |
|  |  |
|  | **if** Position |
|  | Path2 |
|  | **if** |
|  | **return** |
|  | **else**: |
|  | **if** |
|  | Stay |
|  | **else**: |
|  | Path2 |
|  | **return** |
|  | **return** |

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 (lines1-2). 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 but 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.

So long as the requirements for well-formedness are met, the authors of TP and TPTS guarantee that each algorithm solves all MAPD instances [13].

## 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 assure that results from an algorithm are generalizable [10]. For instance, the WHCA\* algorithm presented in [19] 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 [13]. Some standard test cases used in MAPF problems are offered as possible reference benchmarks, but there is little agreement on the use of and limited applicability of results from these cases [12].

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 testing time requirement, this chapter presents a general approach to the implementation of algorithms. The approach presented here is a procedure which keeps the overall system coherent and easy to configure for testing. Results of the implementation of algorithms from Chapter ?? using this strategy are presented in Chapter ??? to demonstrate its feasibility.

### 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.
* Agents must move efficiently toward their goals.

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 implement 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 in an algorithm’s progression. It promotes the desired modularity and extensibility while presenting a simplified programming interface.

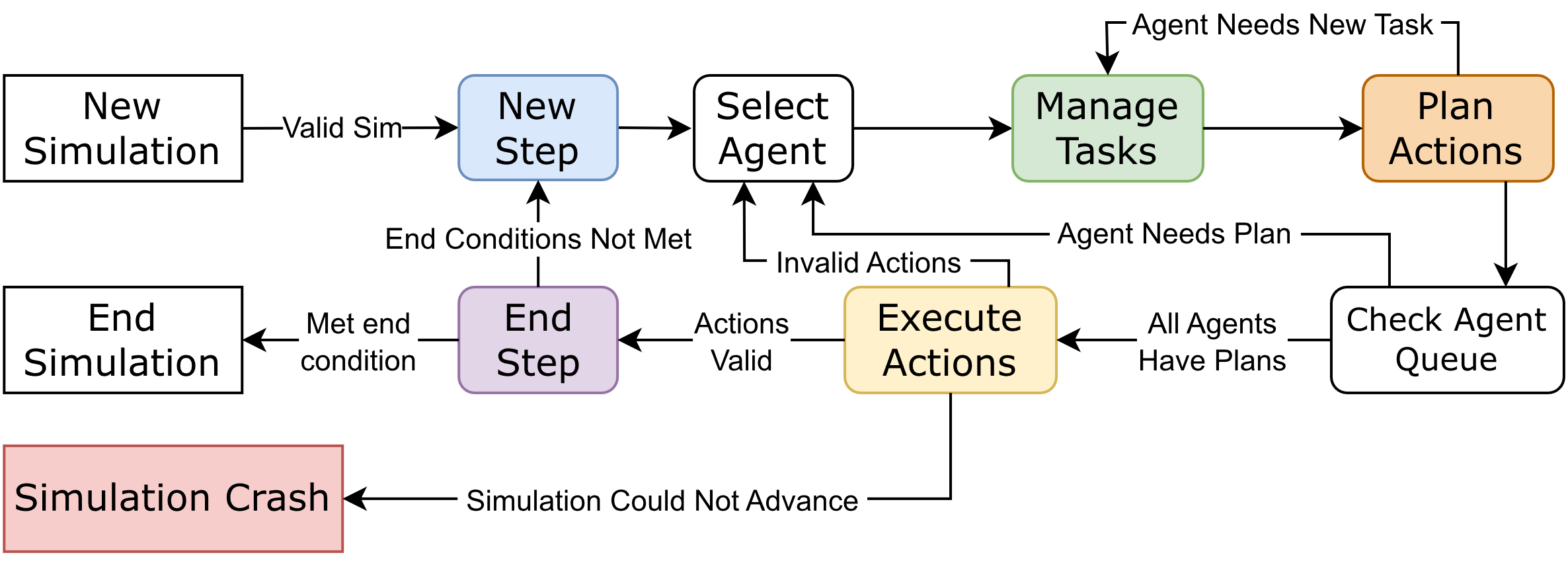


Figure 1: Graphical overview of the proposed state machine.

Actions handling 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 an application described in Chapter ??? which is driven by this approach in its implementation of the algorithms described in Chapter ??: 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 programming implementation of the state machine is left to discussion in the Appendix.

#### 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 point consume a timestep?
* Does rotation have a cost?
* Do agents experience faults during operation?
* How are tasks added to the system?

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. Perhaps the most important consideration is what constitutes the end of the simulation if the solver is not intended to run for an indeterminate amount of time. 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.

#### 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 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.

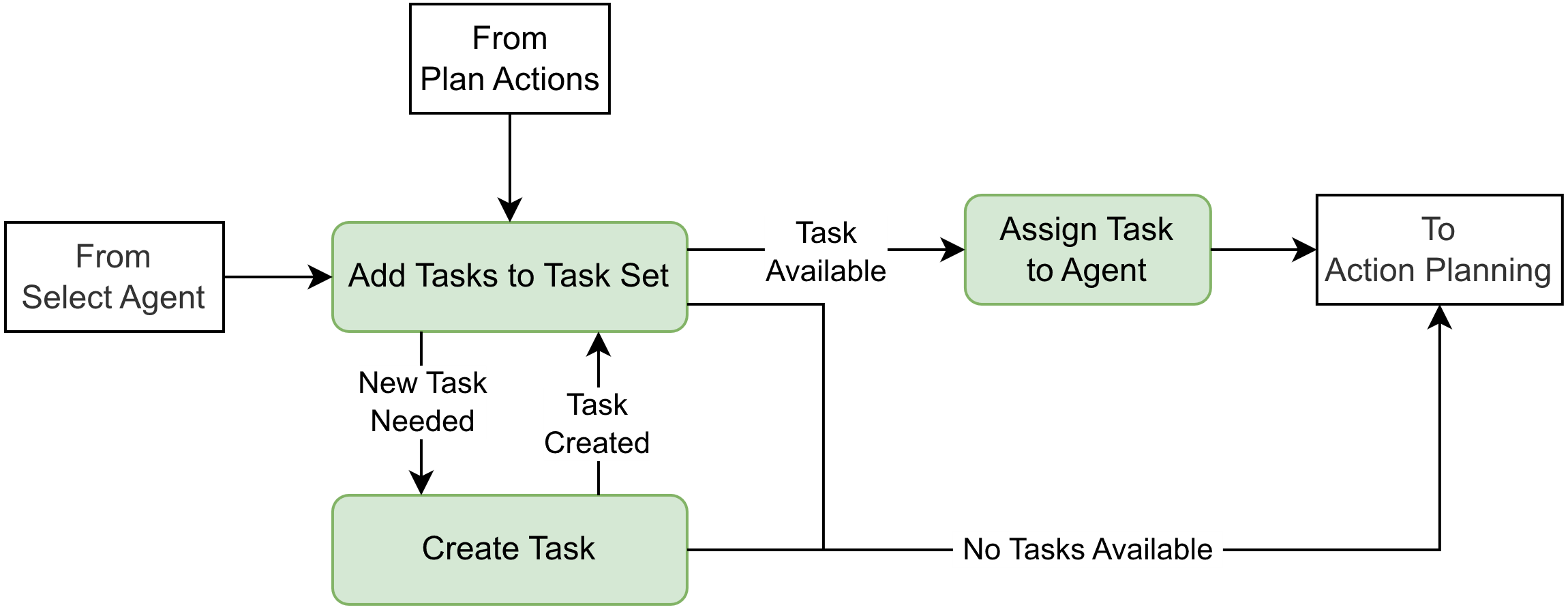


Figure 2: Graphical overview of the process undergone in the Task Management state.

#### 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 are grouped together as logical branches of the action planning state.

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.

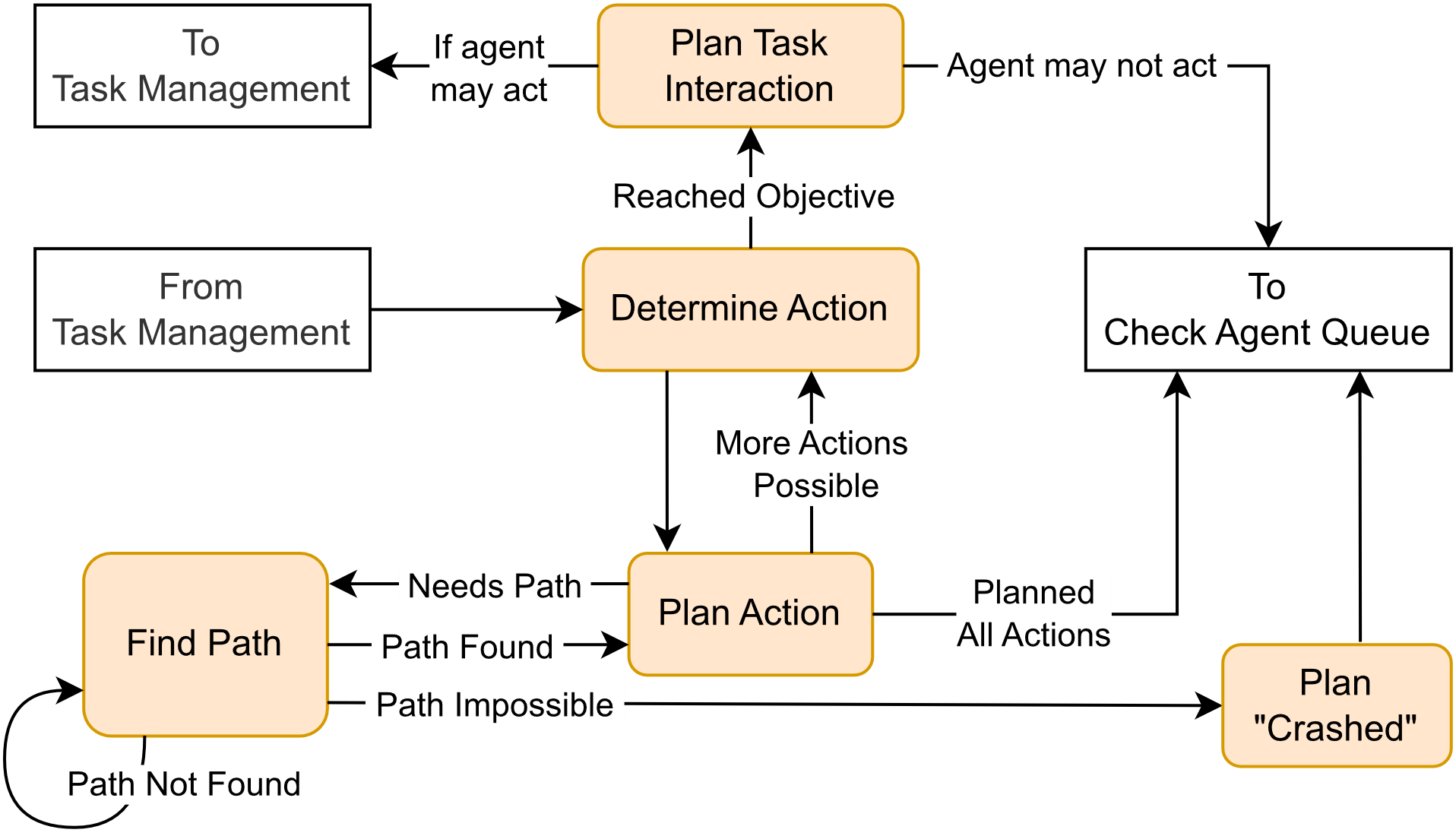


Figure 3: Graphical overview of the process undergone during the Action Planning state.

#### 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 ???. 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.

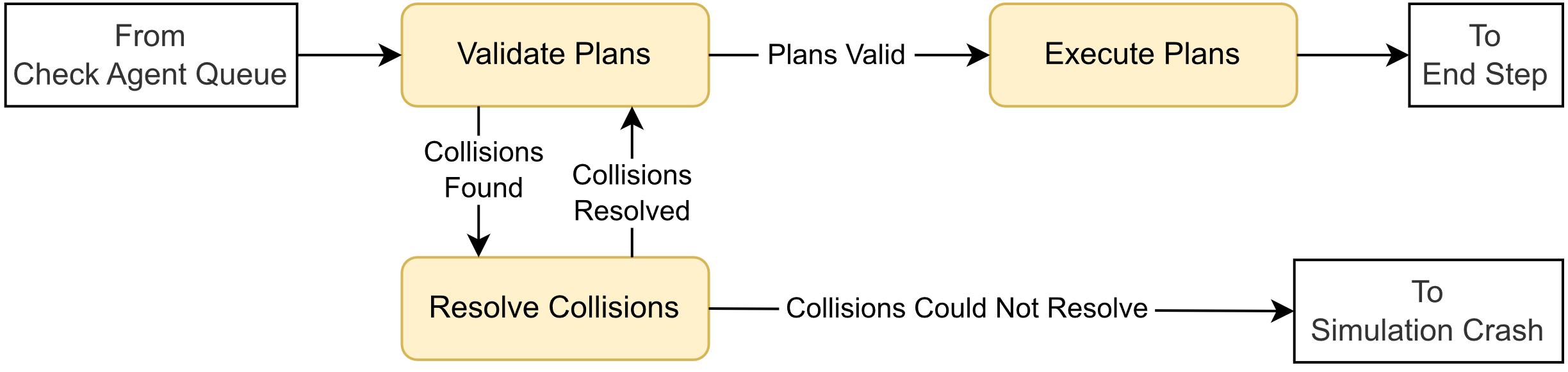


Figure 4: Graphical overview of the process undergone in the Action Execution state.

#### 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 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.

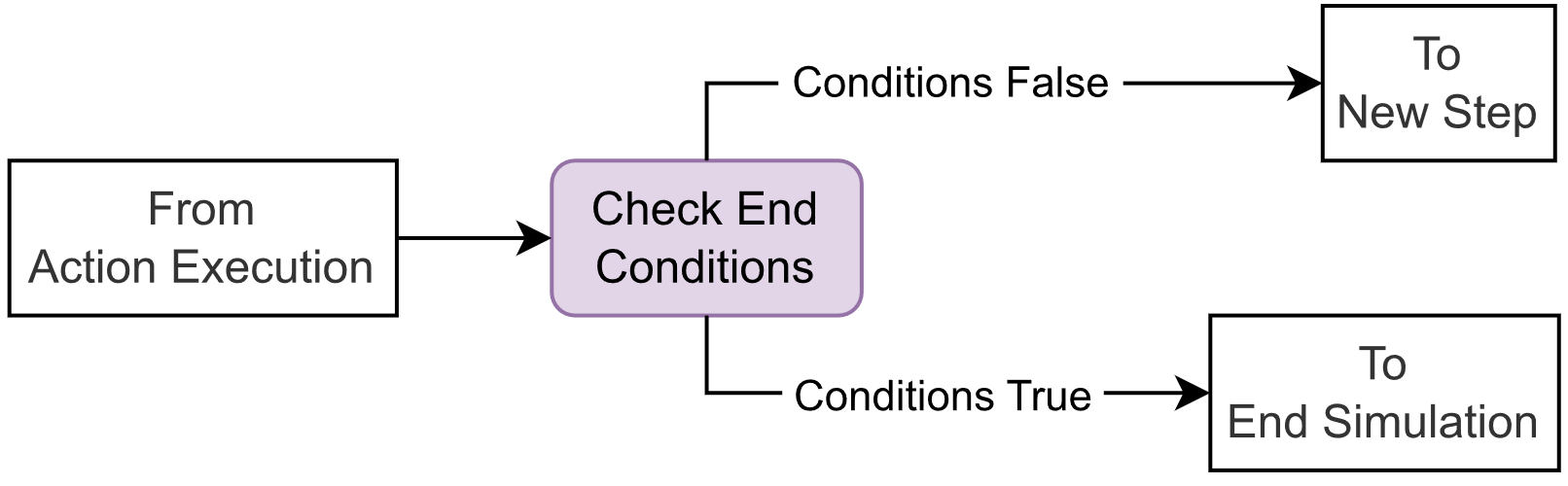


Figure 5: Graphical Overview of the process undergone during the End Step

## 4. Framework Application

In Chapter ?, two decoupled multi-agent problem algorithms were introduced and their inner workings explained. In Chapter ??, 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 two different algorithms 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 ?. 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 the Appendix, 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 as needed. As an example, Table ? lists all the configuration options for the implementation used in FleetBench at time. 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.

|  |  |  |
| --- | --- | --- |
| **Relevance** | **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 |  |
| Pathfinder Search Window Depth | , if applicable |
| Agent Behavior | Agent Collision Handling | Respected |
| Ignored |
| Task Interaction Cost | Instantaneous |
| Timestep |
| Agent Count |  |
| Task Generation | Task Generation Technique | Scheduled |
| On Agent Availability |
| Task Location Configuration | Node weights |
| Include/Exclude node |
| End Conditions | Simulation ends on task completions |  |
| Simulation ends on timesteps elapsed |  |
| Simulation ends on schedule completion | True or False |

### WHCA\*

To begin 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, as in the implementation given in Appendix ?, 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 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 the Appendix, as it is not central to the work done here.

The bulk of the logic employed in WHCA\* is for pathfinding which makes it relatively trivial 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, explained in Appendix ?.

Algorithm ? 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-cemented during the action execution state.

|  |  |  |
| --- | --- | --- |
| **Algorithm 1** Solving MAPD problems using WHCA\* | | |
|  | Input is an MAPD problem , defined by the user | **State** |
|  | PreProcess | **New Simulation** |
|  |  |
|  | **while** **not** endConditions | **End Step** |
|  | Update | **New Step** |
|  | **for** | **Select Agent** |
|  | **if** allows task creation: | **Manage Tasks** |
|  |  |
|  | **if** has no assignment: |
|  |  |
|  | **if** |
|  | Assign to |
|  | **while** mayAct: | **Plan Actions** |
|  | **if** Position Goal |
|  |  |
|  | **else**: |
|  | **if** has : |
|  |  |
|  | **else**: |
|  | WHCAStar(PositionGoal |
|  |  |
|  | Store in system memory |
|  | validatePlans | **Execute Actions** |
|  | **while not** |
|  | resolveCollisions |
|  | Validate all stored plans |
|  | **if**  is impossible: | **Sim. Crash** |
|  | System Crashes |
|  | Execute all actions | **Execute Actions** |
|  | System time increments | **End Step** |
|  | Solution found for , for endConditions | **End Simulation** |

### TPTS

TPTS, being designed for the MAPD problem, better analogizes real-world applications in continuously active cooperative robotics applications and is more involved. 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 [13], 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 simulated anyway a conflict resolution system must be in place to prevent the simulation from entering the crashed state.

Once again, a line-by-line process of re-composing the algorithm into named routines is presented in Algorithm 2. 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 3.

|  |  |  |
| --- | --- | --- |
| **Algorithm 2** Solving MAPD problems using TPTS | | |
|  | Input is an MAPD problem , defined by the user | **State** |
|  | PreProcess | **New Simulation** |
|  | checkWellformed |
|  | **if not** |
|  | Handle badly formed problems, simulation may abort |
|  |  |
|  | **while** **not** endConditions | **End Step** |
|  | Update | **New Step** |
|  | **for** : | **Select Agent** |
|  | **if** allows task creation: | **Manage Tasks** |
|  |  |
|  | **if** has no assignment: |
|  | GetTask |  |
|  | **while** mayAct: | **Plan Actions** |
|  | GetTask |  |
|  | validatePlans | **Execute Actions** |
|  | **while not** |
|  | resolveCollisions |
|  | Validate all stored plans |
|  | **if**  is impossible: | **Sim. Crash** |
|  | System Crashes |
|  | Execute all actions | **Execute Actions** |
|  | System time increments | **End Step** |
|  | Solution found for , for endConditions | **End Simulation** |

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 ?. 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.

|  |  |  |
| --- | --- | --- |
| **Algorithm 3** GetTask from TPTS | | |
|  | Demonstrates that planning paths occurs between management of tasks, requiring a different strategy. | **State** |
|  | **function** GetTask | **Manage Tasks** |
|  |  |
|  | **while** |
|  |  |
|  |  |
|  | **if** no agent assigned to |
|  | Assign to |
|  | Path1 | **Plan Actions** |
|  | **return** | **Manage Tasks** |
|  | **else**: |
|  |  |
|  | agent assigned to |
|  | Path |
|  | Unassign from ; Remove from |
|  | Path1 | **Plan Actions** |
|  | **if** : | **Manage Tasks** |
|  | GetTask |
|  | **if** : |
|  | **return** |
|  | **else**: |
|  |  |
|  | **if** Position |
|  | Path2 | **Plan Actions** |
|  | **if** | **Manage Tasks** |
|  | **return** |
|  | **else**: |
|  | **if** |
|  | Stay | **Plan Actions** |
|  | **else**: | **Manage Tasks** |
|  | Path2 | **Plan Actions** |
|  | **return** | **Manage Tasks** |
|  | **return** |

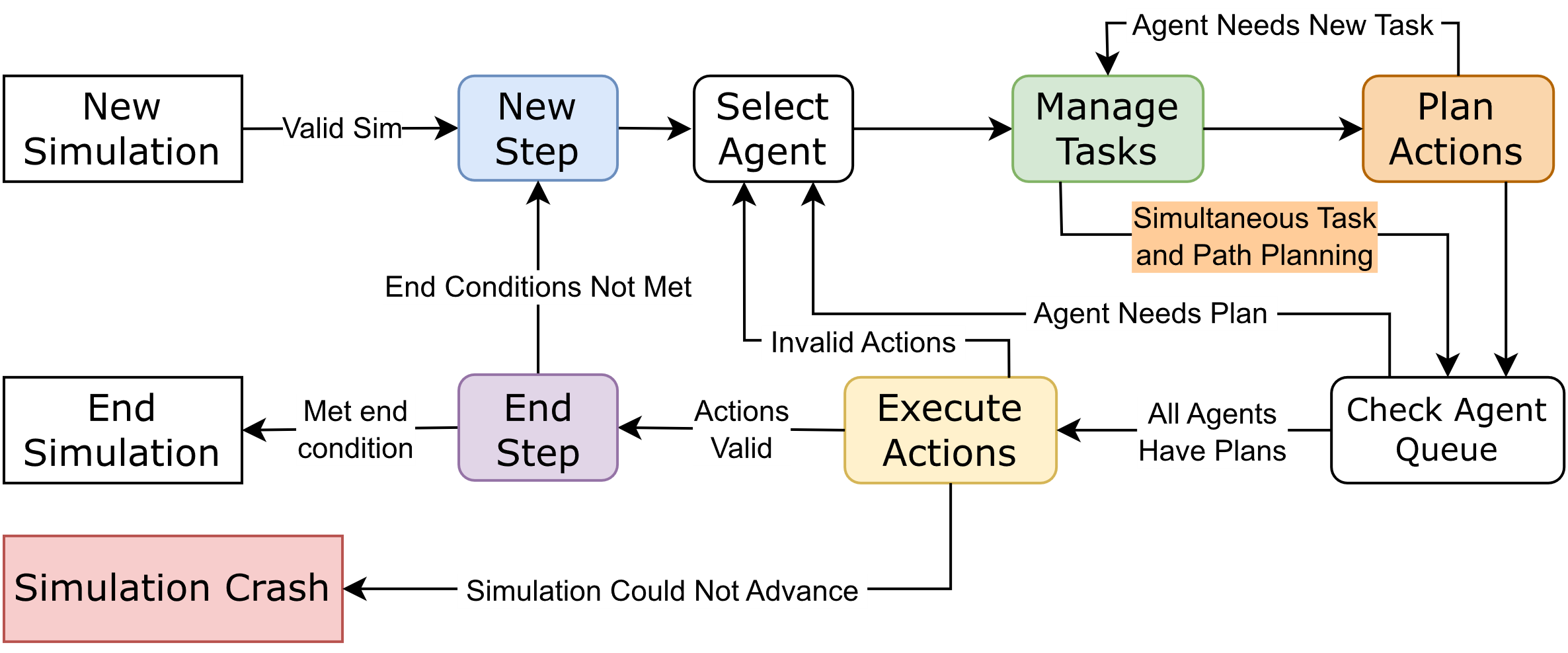


Figure 1: Modified state machine diagram, showing a new branch which accounts for simultaneous task and path planning during the manage tasks state.

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 determine action state, shown in Figure ?? from Chapter ??, to determine whether ***Path1***, ***Path2***, or ***Stay*** should be called, respectively. Instead, during the task management state the calls to the pathfinders ***Path1ps***, ***Path2ps***, and ***Stayps*** 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 4, showing that the task determination and the action planning are decoupled and solved independently.

|  |  |  |
| --- | --- | --- |
| **Algorithm 4** Solving MAPD problems using TPTS | | |
|  | Input is an MAPD problem , defined by the user | **State** |
|  | PreProcess | **New Simulation** |
|  | checkWellformed |
|  | **if not** |
|  | Handle badly formed problems, simulation may abort |
|  |  |
|  | **while** **not** endConditions | **End Step** |
|  | Update | **New Step** |
|  | **for** : | **Select Agent** |
|  | **if** allows task creation: | **Manage Tasks** |
|  |  |
|  | **if** has no assignment: |
|  |  |
|  | **while** |
|  | t |
|  |  |
|  | **if** no agent assigned to |
|  | Assign to t |
|  | **else**: |
|  | Store assignments, reservations in memory |
|  | agent assigned to |
|  | Path |
|  | Unassign from |
|  | Remove from , |
|  | Path1ps |
|  | **if** : |
|  | GetTask |
|  | **if** : |
|  | **break** |
|  | **else**: |
|  | Restore assignments, reservations |
|  | **if** is not “intendTask”: |
|  | **if** Position |
|  | Path2ps |
|  | **else**: |
|  | **if** |
|  | Stayps |
|  | **else**: |
|  | Path2ps |
|  | **while** mayAct: | **Plan Actions** |
|  | If is still valid use it, otherwise: |
|  | **if**  is “intendTask”: |
|  | Path1 |
|  | **else if** is “intendRest”: |
|  | Path2 |
|  | **else if** is “intendStay”: |
|  | Stay |
|  | validatePlans | **Execute Actions** |
|  | **while not** |
|  | resolveCollisions |
|  | Validate all stored plans |
|  | **if**  is impossible: | **Sim. Crash** |
|  | System Crashes |
|  | Execute all actions | **Execute Actions** |
|  | System time increments | **End Step** |
|  | Solution found for , for endConditions | **End Simulation** |

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 ?. 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.

## 5. Implementation, Testing, and Results

To demonstrate the efficacy of the implementation strategy presented in Chapter ?, the 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 ?. During simulation, a state machine designed exactly as presented in Chapter ? 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. For convenience, Table ? lists all algorithms which are included in the application at time of writing. 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 point highlighting, and labeling on a per-state basis at the user’s request.

|  |  |  |
| --- | --- | --- |
| **Table 1** Listing FleetBench algorithms and appendix locations. | | |
| **Problem Type** | **Algorithm** | **Appendix** |
| Single-Agent Pathfinding | Single-Agent A\* | ? |
| MAPF | Multi-Agent A\* (LRA\*) | ? |
| Cooperative A\* (CA\*) | ? |
| Hierarchical CA\* (HCA\*) | ? |
| Windowed HCA\* (WHCA\*) | ? |
| MAPD | Token Passing (TP) | ? |
| TP with Task Swaps (TPTS) | ? |

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 general functions which are listed in Appendix ?. These functions must be present in the algorithm scripts and some data must be returned in a certain format, but no further requirements are imposed.

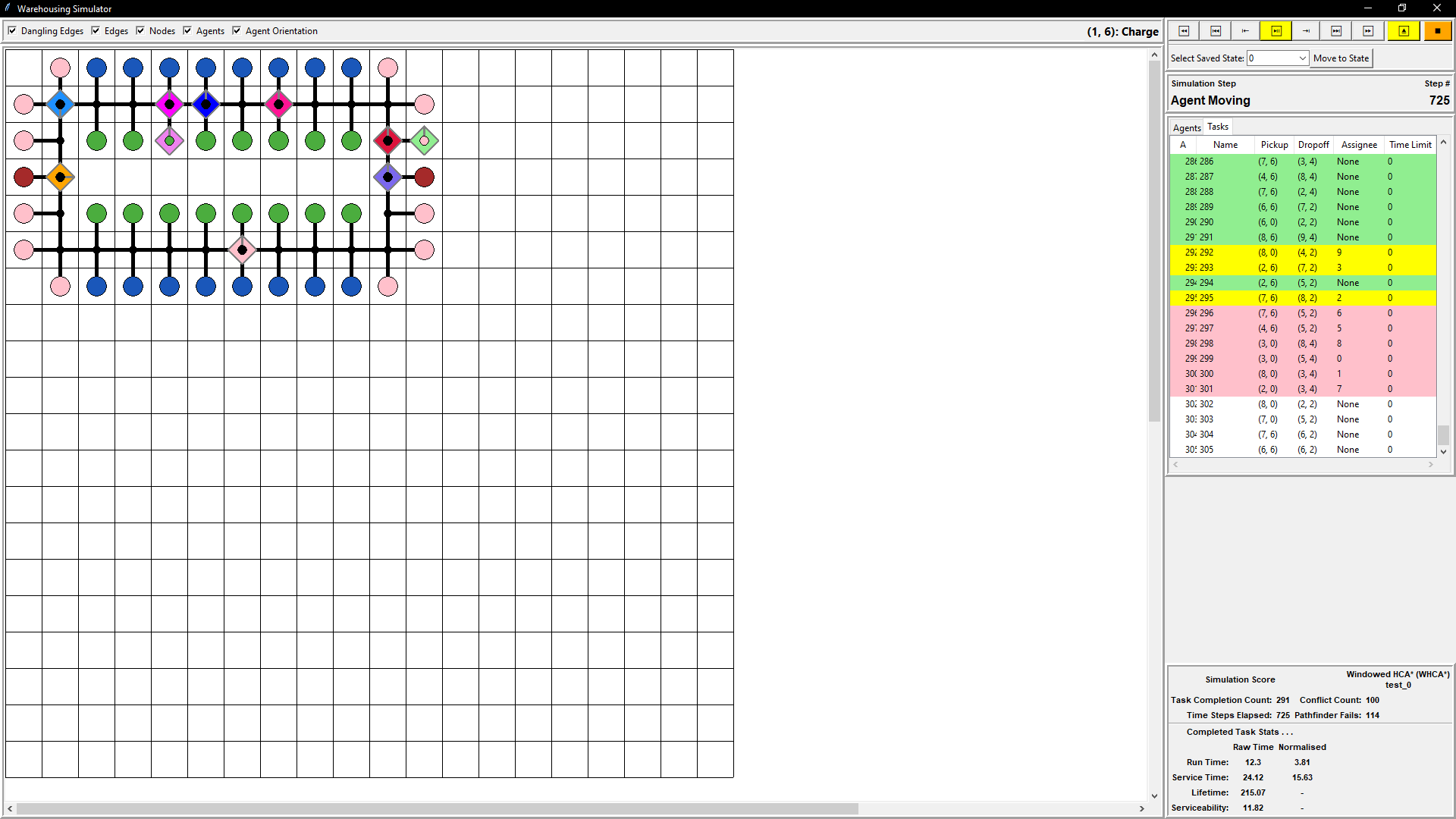


Figure 1: An image of the simulation window in FleetBench. A defined problem has been submitted and the program waits for a user to start simulating.

A second application called GraphRendering was developed to provide a visual process for designing the system tile map. 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. It is possible to generate maps externally, using the map input file format given in Appendix ?.

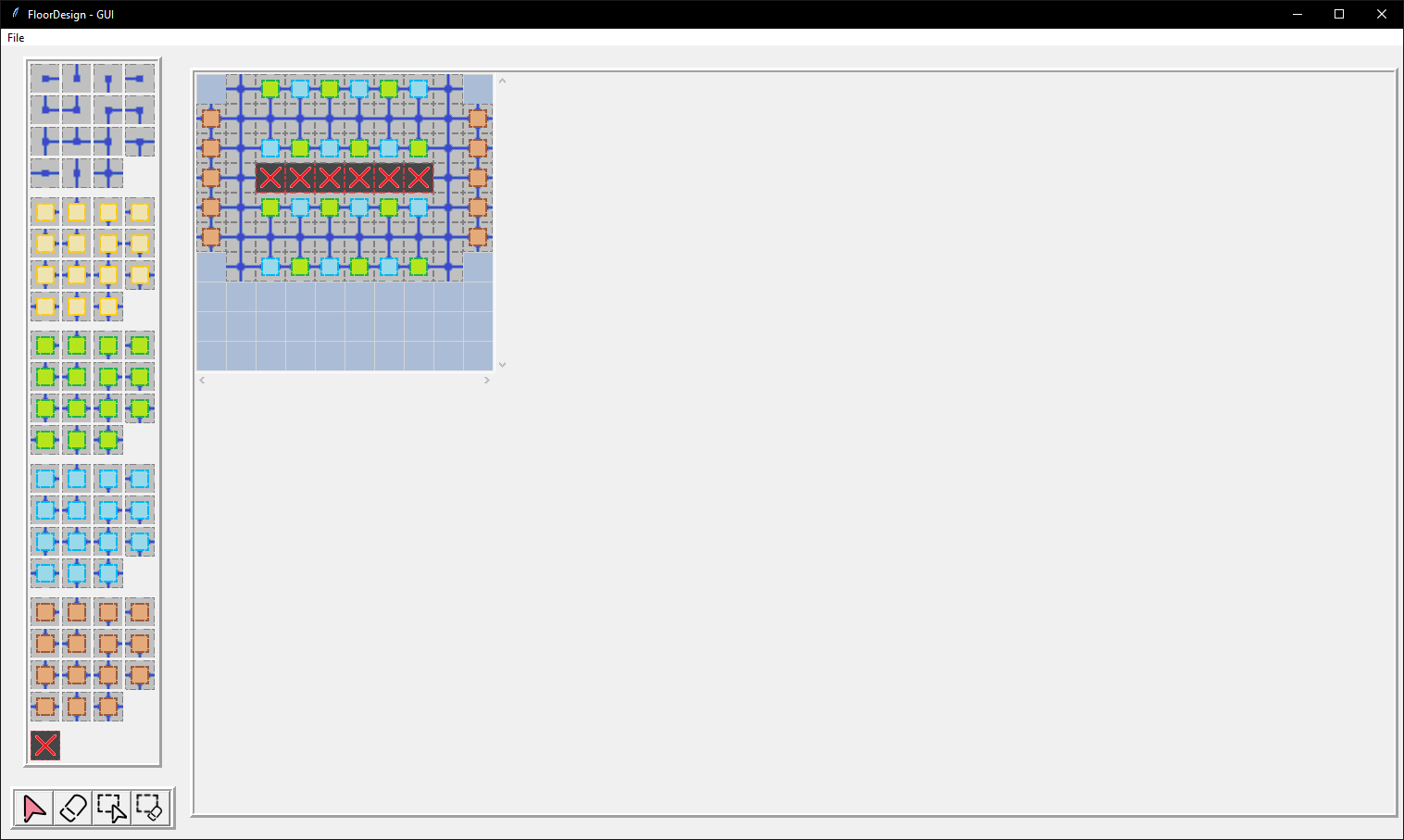


Figure 2: Screenshot of the map creation process in GraphRendering.

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 ?.
* Create an implementation of the algorithm if one is not already provided, adhering to the extension documentation in Appendix ?.
* 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.
* 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.
  + If a predefined list of tasks should be used, the user will need to provide a comma-separated values file as described in Appendix ?.
* 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.

A screenshot of a computer

Description automatically generated

Figure 3: Data panel of a simulation in FleetBench.

### 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 show 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

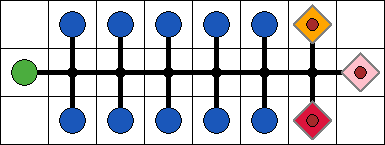


Figure 4: Test case 1. A 1-width corridor with three agents, shown in orange, pink, and red. The green, leftmost, node is the deposit location, while blue nodes are pickup locations. The agent start positions are endpoints satisfying TP and TPTS requirements.

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 spaces to avoid collisions moving down the corridor, which may itself impact the delivery of task objects to their destinations.

The simulation configuration options used to obtain the results in the next section are given in Table 1. 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 1** 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 ?) | | | |
| End Condition | All Scheduled Tasks Completed | | | |

#### Case 2

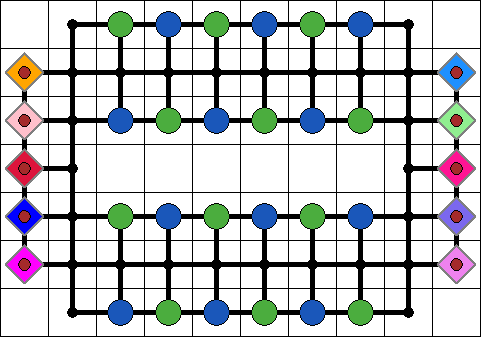


Figure 5: Test case 2. A warehouse reorganization problem with 10 agents, starting on the sides of the system map. The agents start on non-task endpoints, satisfying the TP and TPTS well-formed requirements.

A second test case is developed, very similar in form to the warehousing situation presented in [13]. 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.

As before, simulation configuration options are presented in Table ?. 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 2** 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 ?) | | | | | | | | |
| End Condition | All Scheduled Tasks Completed | | | | | | | | |

#### Case 3

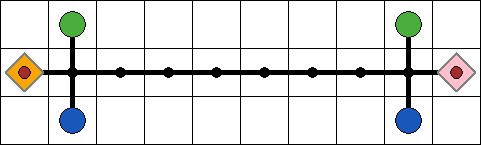


Figure 6: Test case 3. A 1-width bottlenecked corridor with task endpoints at each side. Two agents share the space and are forced to use the nodes at the ends of the corridor to exchange places.

This test case exposes a consequence of treating all edge weights as having the same value when finding paths using A\*. Two agents must share a long corridor with no ability to exchange positions except at the corridor’s ends. The heuristic defined by the code in Appendix ? encourages 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 presented in Table ?. 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 [19].

|  |  |  |  |
| --- | --- | --- | --- |
| **Table 3** 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 ?) | | |
| 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, 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 3.

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 moving toward or executing a task. 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 is an approximation of how long it takes agents to reach 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:

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 time needed for any given interval. 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 [19]. 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 combined time loss due to congestion (Run Time) and time lost due to inoptimal starting position of agents servicing a task. The two values are given by these formula:

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 frequently due to the agent searching for a path being trapped in its location, rather than the obstructions being distant as the search depth would simply increase until the path is found. Typically when this occurs the agent is about to experience a conflict anyway, so the failure counts tend to be 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 ?, and the resulting data are provided in Table 4. All algorithms from Table 1 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. 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 further translates to a longer 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 in all algorithms tested, 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 staggered path planning and WHCA\* window size.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Table 4** 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 ?, and the resulting data are provided in Table 5. 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 [30]. 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 5** 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. It 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 ?, and the resulting data are provided in Table 6. 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 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. 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. This sequence is represented in Figure ?.

|  |  |  |
| --- | --- | --- |
| A grid with a grid and a line with dots and circles  Description automatically generated with medium confidence | A colorful lines and dots on a grid  Description automatically generated | A colorful squares and dots  Description automatically generated with medium confidence |
| (a) | (b) | (c) |
| A colorful lines and dots  Description automatically generated with medium confidence | A grid with a line and dots  Description automatically generated with medium confidence | A line with dots and circles  Description automatically generated with medium confidence |
| (d) | (e) | (f) |

Figure 7: Sequential moves by two 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 6** 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 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 (Chapter ?), two solutions to the MAPF and MAPD problems (Chapter ??), a generalizable approach to adapting solutions to the problems for implementation (Chapter ???), and a workflow consisting of two computer applications which ensures a level testing field for multiple algorithms (Chapter ????). 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 ???, 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 ?? were then tested against these test cases. This process demonstrated known and hypothesized behavior from research in the field within the implementation, affirming the accuracy of the implementation. 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 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 ??? as WHCA\* or TPTS. For example, BIBOX [1] and Push and Swap [2] solve multi-agent problems through topological analysis of the system state. Centralized approaches such as CBS [3] 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 [4]. While the framework presented in this work can be used as a target for adapting the function of each of these solutions, it may become simpler for the user to adjust the presented state machine to better suit these approaches.

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.

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, executed at 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.

[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] “5 Medical Robots Making a Difference in Healthcare,” CWRU Online Engineering (Live). Accessed: Nov. 05, 2023. [Online]. Available: https://online-engineering.case.edu/blog/medical-robots-making-a-difference

[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] O. Salzman and R. Stern, “Research Challenges and Opportunities in Multi-Agent Path Finding and Multi-Agent Pickup and Delivery Problems,” *N. Z.*, 2020.

[11] 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.

[12] 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.

[13] 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

[14] 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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] K.-H. C. Wang and A. Botea, “MAPP: a Scalable Multi-Agent Path Planning Algorithm with Tractability and Completeness Guarantees”.

[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, “gdmapf.” Sep. 14, 2023. Accessed: Nov. 05, 2023. [Online]. Available: https://github.com/nathanfranke/gdmapf

[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] 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.

[28] 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.

[29] 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.

[30] 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.