**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 while remaining intuitive and simple, further 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, including the options discussed in Chapter ?. During simulation, a state machine designed exactly as presented in Chapter ? 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. 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 in real time at the user’s request.

**Table listing all algorithms and the appendix section containing the source**

FleetBench was designed to be extensible. By placing script files in the appropriate application path, a user is able to add additional algorithms to the program. The state machine structure provides a regulated way of calling specific functions which are listed in Appendix ?. These functions must be present in the algorithm scripts and values must be returned in a certain format, but no further requirements are imposed.

**Figure showing the simulation screen of FleetBench**

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.

**Figure showing the design of a floorplan 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.

**Figure showing a set of results from FleetBench simulation**

**Test Cases**

Several test cases were produced to demonstrate FleetBench’s implementation of the WHCA\* and TPTS algorithms. Test cases were developed quickly, supporting the usability and flexibility of the applications.

**Case 1**

The first case is roughly analogous to cooperatively shelving books in a library. There is a single task starting node, acting as the source of items into the storage system managed by three agents. Several destination nodes exist to represent the many sorting locations at which items may end up. 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 ?. 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 showing the settings used for each simulation in case 1**

**Case 2**

A second test case is also developed, very similar to the warehousing situation presented in [1]. Rows of pickup nodes represent goods in storage which need to be moved to delivery locations presumably connected to fulfillment processes. Agents must again travel through corridors, only this time the spaces are less interconnected, such that “bays” are formed at each interaction point. This creates a severe bottleneck in each corridor that must be navigated without obstructing other agents.

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

**Table showing the settings used for each simulation in case 2**

**Case 3**

A test case which exposes the behavior of the particular A\* implementation in FleetBench is also created. 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 the 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 directly travel to. 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. It’s performance is compared to HCA\*, which plans full paths when possible, and against LRA\*, the least clever multi-agent algorithm.

**Table showing the settings used for each simulation of case 3**

**Results**