Multiagent Neuro-evolution for Learning Traffic Management Policies

Jen Jen Chung

November 2018

Abstract

The following provides documentation for `multiagent_learning/testWarehouse.cpp` and its associated libraries. The top level project file is available at https://github.com/JenJenChung/multiagent_learning.git, the libraries are available at https://github.com/JenJenChung/include.git.

Contents

1 Introduction

The `testWarehouse` program was first introduced in [3] and was again used for generating the experimental results in [2]. A complete description of the domain is available in either reference; below is a summary of the main points.

1.1 Domain Description

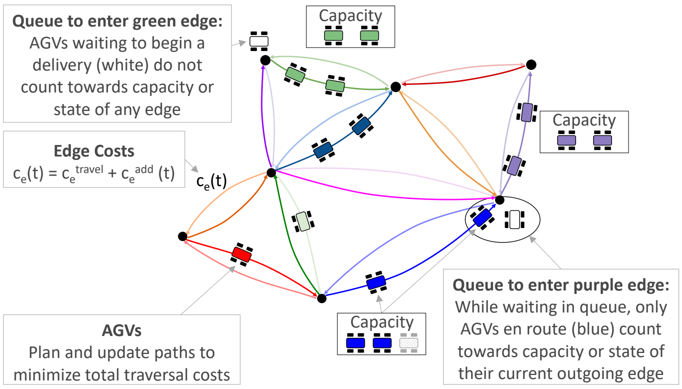

Figure 1: The environment is described by a traffic graph where each directed edge must obey strict capacity constraints. AGVs cannot enter edges that are already at capacity. Those waiting to transition between edges continue to occupy space on their current edge. The goal of the traffic management domain is to find the set of additional cost functions $c_e^{add}(t)$ that result in the maximal number of successful deliveries.

Figure 1 provides an illustration of the domain set up. In this domain, M autonomous ground vehicles (AGVs) deliver packages between various locations in a warehouse. The routes in the warehouse are represented as a high level traffic graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each edge $e \in \mathcal{E}$ defines a single direction of travel between two vertices of the graph, i.e. $e = (u, v)$ is the edge from vertex u to vertex v, where $u, v \in \mathcal{V}$. AGVs compute their paths across this graph according to some cost-based planner such as Dijkstra's or A*. The costs associated with traversing each edge are defined as the sum of a fixed and known cost of travel, $cost_e^{travel}$, and an additional time-varying cost $cost_e^{add}(t)$ assigned by the traffic management agents (described in the following subsection),

$$cost_e(t) = cost_e^{travel} + cost_e^{add}(t). \tag{1}$$

AGVs in the system have access to the instantaneous graph costs calculated using Equation (1), and are able to replan their paths according to the latest costs at any edge transition. However, once they begin traversing an edge, they are committed to continuing along that edge until they reach their next transition. In the following experiments, all AGVs greedily plan to minimize their traversal costs directly according to the costs at the time of planning.

The domain defines an AGV capacity for each directed edge in the graph cap_e, which imposes a constraint on the motion of the AGVs. During an episode, the number of AGVs on an edge, $n_e(t)$, cannot exceed the capacity of that edge. This means that an AGV planning to transition to an edge which is at capacity must wait on its current edge until there is space. While waiting, it continues to count towards the capacity of its current edge. All AGVs are initialized at the start of an episode. Once an AGV completes a delivery, it is immediately assigned a new delivery mission which begins at its current location. At this point, if the AGV must wait to enter the first edge on its new path, it does not count towards the capacity of any edge but is considered to be in a "holding zone" until it begins traversal.

1.2 Multiagent Traffic Management

The task of the traffic management system is to discover the appropriate additional costs, $c_e^{add}(t)$, to apply to each edge in the traffic graph to incentivize the AGVs to avoid congested areas but still reach their destinations in a timely manner. The interaction between the multiagent traffic management team and the AGV traffic is shown in Figure 2.

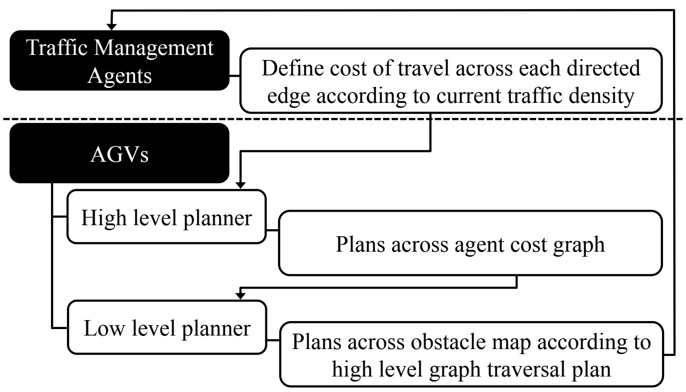

Figure 2: Hierarchical traffic management formulation, noting the separation between the multiagent traffic management system and the AGVs. The travel space is first decomposed into a high level graph representing the connectivity of different regions in the map. The multiagent system defines the cost of travel across this traffic graph, and the AGVs use these costs to determine their sequence of edge traversals. A lower level planner is then assumed to handle the local collision avoidance procedures through the obstacle map. Figure from [3].

Given N agents in a traffic management team, the goal is to concurrently learn the local costing strategies that result in the joint policy $\Pi^* = \{\pi_i\} \forall i \in \{0, \cdots, N\text{-}1\}$, which globally produces the highest number of successful deliveries. Agents are defined based on their scope, that is, the component of the joint state that they can observe, and the subset of the joint actions that they can control. In the traffic management domain, this is represented as the set of edges that an agent manages, as well as the resolution of the information available to each agent regarding the traffic on those edges. In general, given the state of each edge to be $\mathbf{s}_e(t)$, then the state for agent i is defined as,

$$\mathbf{s}_i(t) = [\mathbf{s}_e(t)], \quad \forall e \in \mathcal{E}_i, \tag{2}$$

where $\mathcal{E}_i$ is the set of edges managed by agent i. Thus, the output actions of agent i are,

$$\mathbf{a}_i(t) = \pi_i(\mathbf{s}_i(t)) = \left[cost_e^{add}(t)\right], \quad \forall e \in \mathcal{E}_i, \tag{3}$$

and the objective of the multiagent team is,

$$\max_{\Pi} \quad G(\Pi) = \textit{total deliveries}, \tag{4}$$

$$\text{s.t.} \quad n_e(t) < cap_e \quad \forall t, e \in \mathcal{E}. \tag{5}$$

Six agent definition variants are currently implemented, $\{$*Link*; *Link, time*; *Intersection*; *Intersection, time*; *Centralised*; *Centralised, time*$\}$. The *centralised* agent is a straightforward implementation of a single agent learner over the global state-action space. The next three subsections provide a description of the *link* and *intersection* agents, as well as the *time* inclusion variation.

1.2.1 Link Agents

Each *link* agent is assigned to a single directed edge, thus the team consists of $N = |\mathcal{E}|$ agents. The link agent state is simply defined as the total number of AGVs currently traversing the edge, while the link agent output is the additional cost of travel for that edge. That is,

$$s_i^{link}(t) = n_{e_i}(t), \tag{6}$$

$$a_i^{link}(t) = c_{e_i}^{add}(t), \tag{7}$$

where e_i is the edge assigned to link agent i.

1.2.2 Intersection Agents

Each *intersection* agent is assigned to manage AGV traffic on the set of *incoming* edges of a particular vertex. Thus, the team consists of $N = |\mathcal{V}|$ intersection agents, whose states and actions are,

$$\mathbf{s}_i^{int.}(t) = [n_e(t)], \quad \forall e \in \mathcal{E}_i, \tag{8}$$

$$\mathbf{a}_i^{int.}(t) = [c_e^{add}(t)], \quad \forall e \in \mathcal{E}_i, \tag{9}$$

where $\mathcal{E}_i$ is the set of incoming edges assigned to intersection agent i. Note that the state-action space for each intersection agent in the team can be heterogeneous in this formulation since each agent's dimensionality is defined by the number of incoming edges they manage. That is, the dimensionality of intersection agent i is $|\mathcal{E}_i|$.

1.2.3 Incorporating Travel Time

In the link agent and intersection agent formulations described in Sections 1.2.1 and 1.2.2, the state information only consists of the current number of AGVs on the edges. The *time* variants investigate the effect of including additional AGV tracking information.

For each edge, we track $d_e(t)$, the amount of time remaining until the next AGV completes its traversal and will attempt to transition to a new edge or complete its delivery. This value can range from the total time required to traverse an edge (if there are currently no AGVs present) to zero, which represents the case where an AGV has completed its traversal and will transition to a new edge at the next timestep provided it does not violate Equation (5). This travel time information is incorporated as an additional element in the state vector for each edge. Thus, the travel-time-augmented link agent state becomes,

$$\mathbf{s}_i^{link,time}(t) = [n_{e_i}(t), d_{e_i}(t)]. \tag{10}$$

Similarly, the augmented intersection agent state is defined as,

$$\mathbf{s}_i^{int.,time}(t) = [n_e(t), d_e(t)], \quad \forall e \in \mathcal{E}_i. \tag{11}$$

Note that the action space for each agent definition remains the same as in Equation (7) and Equation (9), respectively. A similar extension is used for the *centralised, time* agent.

2 Top level project file

The main project file is `testWarehouse.cpp` and is found in the `multiagent_learning` repository. Projects in the `multiagent_learning` repository depend on the following libraries:

- boost
- eigen3
- yaml-cpp (https://github.com/jbeder/yaml-cpp)
- include (https://github.com/JenJenChung/include)

More details will be provided in Section 3 on the `include` library.

2.1 How to build

Ensure that boost, eigen3 and yaml-cpp have been installed on your machine. Then, from the `multiagent_learning` directory:

```
user@computer:~/multiagent_learning $ git clone https://github.com/JenJenChung/include.git
user@computer:~/multiagent_learning $ cd build
user@computer:~/multiagent_learning $ cmake ..
user@computer:~/multiagent_learning $ make
```

NOTE: If you are unable to `make install` the yaml-cpp library or the eigen3 library, you will need to include some additional lines into the top level `CMakeLists.txt` file so that the compiler can find the libraries. namely, after line 11 include:

```
find_package(yaml-cpp)
```

You will also need to update line 18 to specify the path to eigen3 and the yaml-cpp include folder, e.g.

```
set(CMAKE_CXX_FLAGS "-g -Wall -I /path_to_eigen_version/include/eigen3/ -I /path_to_yaml_cpp/
    include/")
```

2.2 How to run

The program requires the user to specify the simulation configuration file and optionally the number of parallel threads to use. The default number of threads is set to 2.

```
user@computer:~/multiagent_learning/build $ ./testWarehouse -c ./config.yaml -t 6
```

2.3 Simulation configuration file

All parameters related to the domain setup, the neuro-evolutionary learning, the scope of the simulation and where to record logged results are specified in the simulation configuration file. The example below is provided in the `multiagent_learning/build` folder:

```
mode:
  type: train # NOTE: agent_policies and eval_file are only used if [mode][type] = test
  agent_policies: ../Domains/Small_120_AGVs_SS_link_time/Results/neural_nets.csv # file
      containing agent policies
  eval_file: ../Domains/Small_120_AGVs_SS_link_time/Results/evaluation_29.csv # file containing
       best teams
domain:
  folder: ../Domains/Small_120_AGVs_SS_link_time/ # directory to domain configuration files
  agents: link_t #{link, link_t, intersection, intersection_t, centralised, centralised_t}
graph:
  vertices: vertices.csv # name of vertices file in domain folder
  edges: edges.csv # name of edges file in domain folder
  capacities: capacities.csv # name of capacities file in domain folder
neuroevo:
  learn: true # apply neuro-evolution?
  population_size: 10 # number of initial policies in population
  epochs: 500 # evolutionary epochs; recommended that you set to 1 if [mode][type]: test
  runs: 30 # number of statistical runs (different random seeds to initialise and mutate
      weights)
simulation:
```

```
18   steps: 200 # number of timesteps for a single episode
19   agvs: origins.csv # name of AGV origins file in domain folder
20   goals: goals.csv # name of goal vertices file in domain folder
21 results:
22   folder: Results/ # folder to store all results files (will be created inside domain folder)
23   evaluation: evaluation # naming for evaluation files (will be appended by statistical run
         number)
24   policies: neural_nets.csv # file name for recording final network weights
```

Listing 1: config.txt: simulation configuration file

mode The program can run under two modes: `train` and `test`.

- In `train` mode, the program initialises a new set of agents and conducts neuro-evolutionary learning using the parameters specified in lines 11-15. All other parameters associated with the neuro-evolution (such as the mutation rates, number of hidden nodes, etc.) are either fixed or computed deterministically from other parameters. More details can be found in Section 3.
- In `test` mode, the program will run the agent policies from the files specified by `mode:agent_policies` and `mode:eval_file` in the domain specified by `domain:folder` in line 6. The agent policies file should include all neural network weights at the end of a training run, ordered according to agent and population member index. From the `eval_file`, the program extracts the population member indices of the champion team that was logged at the end of learning so that the same team can be formed for testing. See Figure 3 for an illustrative example of the program data-cycle.

In general, `test` mode is used for testing policy generalisation. That is, the ability of a team trained on one domain configuration to provide good traffic management strategies for a different domain configuration. These tests can only be conducted over domains with the same multiagent team definition, however, other factors such as the density of traffic, episode length, location of start and goal nodes, etc. can be varied (i.e. those variables specified in lines 18-20 of Listing 1).

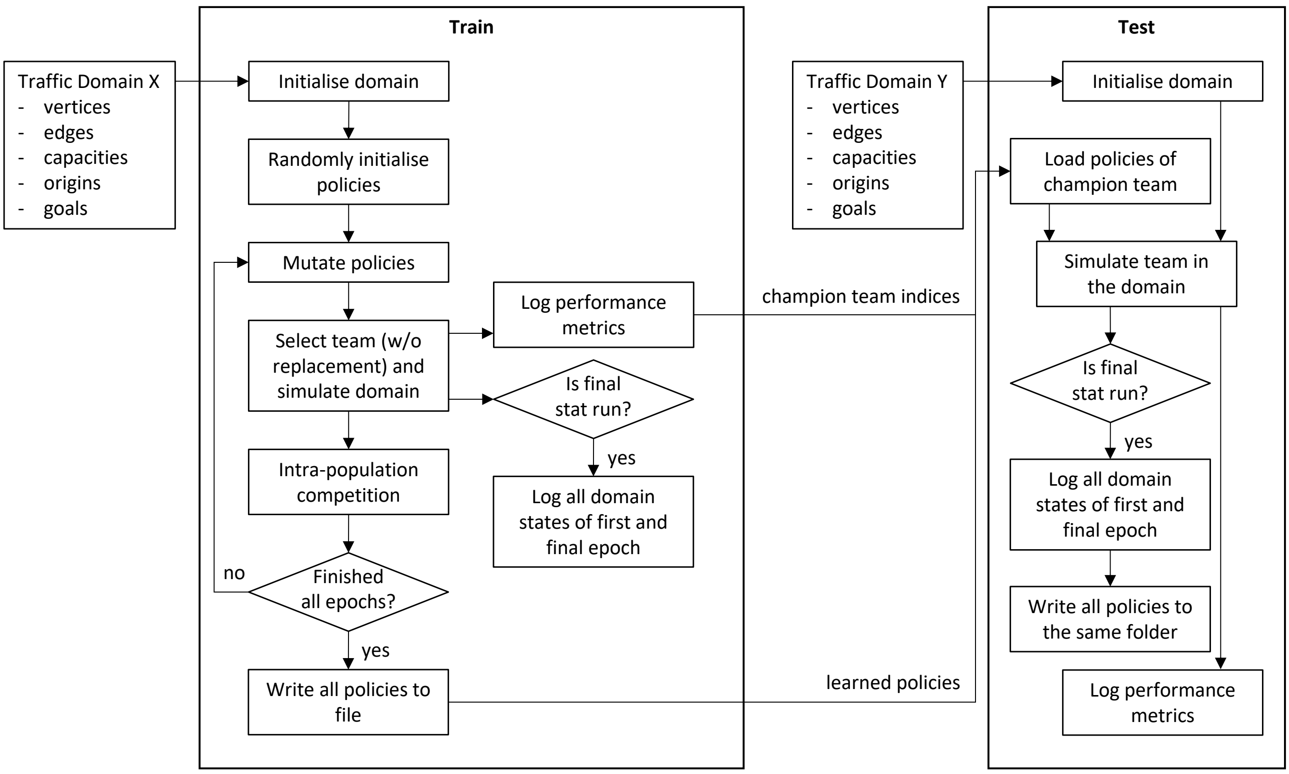

Figure 3: High level `testWarehouse` flow chart.

domain

- Path to `folder` containing all domain configuration files. The training and testing results will also be saved into a new Results folder created at this location.
- The `agents` parameter specifies the agent definition. Six options are currently available: link agents, intersection agents or a centralised agent, and each option with or without AGV travel time augmented to the agent state. See [2] for further details on the different agent definitions.

graph Names of the traffic graph configuration files. The program will expect these files to be in the folder directory specified in `domain:folder`. Each of the domain configuration files are described further in Section 2.4.

neuroevo Neuro-evolution parameters.

- If learning is set to `false`, then no policy evolution occurs across the epochs (i.e. you are only testing the randomly initialised policies).
- The `population_size` refers to the number of randomly initialised policies in each agent's population. Since populations double during the mutation step, then the total number of randomised teams that are evaluated at each learning epoch is equal to twice the population size.
- The `epochs` parameter refers to the number of evolutionary (mutation, simulation, competition) cycles to undertake. Typically you would expect the team performance to converge as the number of learning epochs increases.
- The `runs` parameter specifies the number of statistical runs to conduct, i.e. each run begins with a complete reset of the learning domain with a new random seed.

simulation Domain simulation parameters.

- Specify the number of `steps` of each simulation episode.
- Define the starting vertices of each of the `agvs`.
- List of vertex indices that are potential `goals`. AGV delivery missions will always specify a goal vertex from this set and will also exclude the vertex from which the AGV starts its delivery.

results Output file directories.

- A new `folder` will be created in the `domain:folder` in which the following files will be saved.
- The best team performance for each statistical run is logged in `evaluation_X`, where `X` is the run number.
- The evolved neural network weights (for all `policies` in each agent's population) at the end of the final statistical run.

2.4 Domain configuration

The `testWarehouse` program expects five domain specification files in the `domain:folder`. These are the first five entries listed in Table 2. A few things to note:

- The entries in each row of `capacities.csv` and `edges.csv` must match. The row orders of the other three csv files are inconsequential to the `testWarehouse` program.
- The base cost of edge traversal (third column of `edges.csv`) must be non-negative to avoid errors in the path planning. **During construction of an Edge, this value is cast to a size_t and this taken to equal the number of timesteps required to traverse the edge.**
- The set of vertex IDs do not need to be sequential (or positive), the only requirement is that they are unique. The entries in `goals.csv`, `origins.csv` and the first two columns of `edges.csv` should only contain values from this set.

The final entry, `vertices_XY.csv`, contains the physical (x,y) locations of each vertex and is only used for plotting replay results in Matlab and is not used in `testWarehouse`. The rows in `vertices_XY.csv` correspond to those in `vertices.csv`. More details can be found in Section 4.

Figure 4 shows an example graph, and Table 3 provides an example set of corresponding domain configuration files. In this example, the graph contains bi-directional edges, note that each uni-directional edge must be

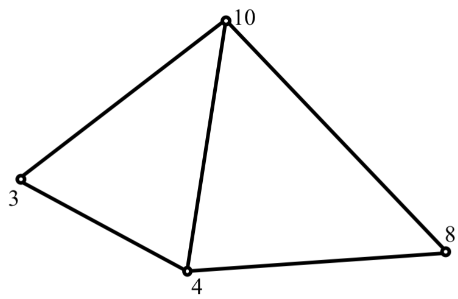

Figure 4: A simple graph example.

File name	Data size	Data type	Description
capacities.csv	$\lvert\mathcal{E}\rvert \times 1$	size_t	The maximum number of AGVs allowed on each edge. $\lvert\mathcal{E}\rvert$ is the total number of edges in the graph. The integer value in each row refers to the capacity of the edge defined in the corresponding row in edges.csv.
edges.csv	$\lvert\mathcal{E}\rvert \times 3$	[int,int,double]	[parent vertex ID, child vertex ID, base cost of edge traversal].
goals.csv	$g \times 1$	int	List of vertex IDs that are potential delivery destinations.
origins.csv	$M \times 1$	int	List of vertex IDs representing the start vertices for all AGVs. M is the total number of AGVs in the system.
vertices.csv	$\lvert\mathcal{V}\rvert \times 1$	int	List of all unique vertex IDs. $\lvert\mathcal{V}\rvert$ is the total number of vertices in the graph.
vertices_XY.csv	$\lvert\mathcal{V}\rvert \times 2$	[double,double]	[x,y] position of each vertex. Only used for plotting during replay.

Table 2: List of required domain configuration files (in alphabetical order).

listed as a separate entry in `edges.csv` and `capacities.csv`. Furthermore, in this example each pair of bidirectional edges has the same base traversal cost and capacity, however, in general this does not need to be the case. Finally, a reminder that although the base traversal costs in the third column of `edges.csv` are read in with double precision, the number of timesteps required to traverse each edge will be the rounded down integer of these values since they will be cast to a size_t.

In this example, there are six AGVs in the domain, three starting at vertex 3, two at vertex 8 and one at vertex 10. There are only two goal vertices, 4 and 10. Thus, after the first delivery, all AGVs will continue to plan paths between vertices 4 and 10.

vertices.csv	edges.csv			capacities.csv	origins.csv	goals.csv
3	3,	4,	3.5	2	3	4
4	3,	10,	7	4	3	10
10	4,	3,	3.5	2	3	
8	4,	10,	6.8	3	8	
	4,	8,	11.1	6	8	
	10,	8,	13.4	7	10	
	10,	4,	6.8	3		
	10,	3,	7	4		
	8,	4,	11.1	6		
	8,	10,	13.4	7		

(a) List of required domain configuration files (in alphabetical order).

vertices_XY.csv	
0,	1
3,	-1
4.5,	5
10.2,	-0.5

(b) (x,y) vertex locations for post-process replay only.

Table 3: Example domain configuration.

3 Libraries

3.1 Domains

The `Domains` library contains all classes related to the simulation of a domain's dynamics. The domain classes defined here provide the main access points for their respective top level programs. In general, they should each contain member functions for initialising, resetting and simulating a single epoch, policy evolution and any desired file I/O capabilities. For the `testWarehouse` program, we use the `Warehouse*` and `AGV` classes.

3.1.1 Warehouse* classes

Each of the `Warehouse*` classes inherit from the basic `Warehouse` class.

Public member functions	
	`Warehouse`(YAML::Node) Constructor. Takes in the simulation configurations as a YAML::Node object.
virtual void	`SimulateEpoch`(bool = true) Simulate an epoch with (default) or without evolution.
virtual void	`SimulateEpoch`(std::vector<size_t>) Simulate an epoch with the multiagent team defined by the vector of agent indices.
virtual void	`InitialiseMATeam`() Initialises the neuro-evolution components and domain housekeeping component of the agents.
void	`EvolvePolicies`(bool = false) Triggers competition (default) and mutation for all agents. If input is set to true, then the competition step is skipped (typically only during the first epoch).
void	`ResetEpochEvals`() Triggers all agents to reset.
void	`OutputPerformance`(std::string) Opens the specified file to write out evaluations at each epoch (actual writing occurs in `SimulateEpoch`).
void	`OutputControlPolicies`(std::string) Writes all neural networks (entire population for all agents) to the input file.
void	`OutputEpisodeReplay`(std::string, std::string, std::string, std::string) Opens the specified files for writing all AGV traversal states, AGV current edges, agent states, and agent actions, respectively.
void	`DisableEpisodeReplay`() Disables recording of domain states.
void	`LoadPolicies`(YAML::Node) High level handler for reading in (learned) policies for simulation
virtual	`~Warehouse`(void) Destructor.
Protected member functions	
void	`InitialiseGraph`(std::string, std::string, std::string, YAML::Node) Creates graph object according to the given domain configuration parameters and file.
void	`InitialiseAGVs`(YAML::Node) Creates AGV objects according to the given domain configuration file.
void	`InitialiseNewEpoch`() Performs all housekeeping required to reset AGVs to original starting location.
std::vector<std::vector<size_t>>	`RandomiseTeams`(size_t) Shuffles the populations for multiagent team formation and simulation. Input provides the size of each population.

virtual void	`QueryMATeam`(std::vector<size_t>, std::vector<double>&, std::vector<size_t>&) Queries the multiagent team for its current action set (graph costs).
void	`UpdateGraphCosts`(std::vector<double>) Updates the edge costs in the graph.
Protected attributes	
size_t	`nSteps`
size_t	`nPop`
size_t	`nAgents`
size_t	`nAGVs`
std::vector<double>	`baseCosts`
std::vector<size_>	`capacities`
bool	`neLearn` Set to true for neuro-evolutionary competition and mutation.
struct	`iAgent` Agent-warehouse bookkeeping
std::vector<Agent*>	`maTeam` Manages agent neuro-evolution routines
std::vector<iAgent*>	`whAgents` Manages agent vertex and edge lookups from graph
Graph*	`whGraph` Vertex and edge definitions, access to change edge costs at each step.
std::vector<AGV*>	`whAGVs` Manages AGV A* search and movement through graph
bool	`outputEvals`
bool	`outputEpReplay`
std::ofstream	`evalFile`
std::ofstream	`agvStateFile`
std::ofstream	`agvEdgesFile`
std::ofstream	`agentStateFile`
std::ofstream	`agentActionFile`

The major differences between the various `Warehouse*` domains lie in the `InitialiseMATeam`, `SimulateEpoch` and `QueryMATeam` member functions. Each of these are adapted to the particular agent definition of their domain (e.g. intersection agents for `WarehouseIntersections`). This mainly involves various bookkeeping procedures to manage and track AGV-agent membership, and also requires a different joint state calculation (called during `SimulateEpoch`) depending on if AGV travel time is included in the domain state:

Private member functions	
void	`GetJointState`(std::vector<Edge*>, std::vector<size_t>&) Computes the state of each agent (current count of AGVs in its region of control).
void	`GetJointState`(std::vector<Edge*>, std::vector<size_t>&, std::vector<double>&) Computes the state of each agent (current count of AGVs in its region of control and the time until next complete traversal on each edge).

Also note that the `WarehouseIntersection*` classes include an additional private helper function:

Private member functions	
size_t	`GetAgentID`(int) Returns the index of the agent handling incoming traffic on a particular vertex.

3.1.2 AGV class

The AGV class handles the delivery mission planning and logging for an AGV in the domain. The main planning routine is accessed via PlanAGV through its agvPlanner member variable.

Public member functions	
	AGV(int, std::vector<int>, Graph*) Constructor. Takes in the AGV's origin vertex, a vector of available delivery destination vertices, and the traffic graph.
void	ResetAGV() Resets the AGV for its next delivery mission.
void	Traverse() Increments the AGV along its path and manages end of path transitions. If the AGV cannot move due to capacity constraints, it will be logged here.
void	EnterNewEdge() Transitions the AGV to a new edge and updates the expected next vertex.
void	CompareCosts(std::vector<double>) Checks if replanning is necessary, i.e. the AGV is waiting to enter a new edge **and** the graph costs have changed since the last plan was generated.
void	PlanAGV(std::vector<double>) Takes the current graph costs as input and queries agvPlanner for the optimal path from its current vertex to its goal.
int	GetNextVertex() Returns the next vertex that the AGV will reach. This value is -1 if the AGV is waiting to enter the traffic graph.
Edge*	GetCurEdge() Returns the current edge that the AGV is on.
size_t	GetT2V() Returns the time to complete the current edge traversal and reach the next intersection.
bool	GetIsReplan() True if replanning is needed.
Edge *	GetNextEdge() Returns the next edge for traversal.
size_t	GetMoveTime() Return the total number of timesteps that the AGV was moving.
size_t	GetMoveTime() Return the total number of timesteps that the AGV was moving.
size_t	GetEnterTime() Return the total number of timesteps that the AGV was waiting to enter the graph.
size_t	GetWaitTime() Return the total number of timesteps that the AGV was waiting to cross an intersection.
size_t	GetNumCompleted() Return the total number of completed missions.
size_t	GetNumCommanded() Return the total number of commanded missions.
Search*	GetAGVPlanner() Returns the AGV's Dijkstra planner.
void	DisplayPath() Print the planned path to the command line.
	~AGV() Destructor.
Private member functions	
void	SetNewGoal() Set a new goal vertex.
Private attributes	

Edge*	`curEdge`
int	`nextVertex`
size_t	`t2v` Time to next intersection.
int	`origin`
int	`goal`
size_t	`nsDel` Number of successful deliveries
size_t	`ncDel` Number of commanded deliveries
size_t	`tMove` Moving time.
size_t	`tEnter` Time spent waiting to enter the graph.
size_t	`tWait` Time spent waiting to cross intersections.
vector<int>	`agvGoals` Vector of valid goal vertices.
bool	`isReplan` True if replanning is needed.
Search*	`agvPlanner` AGV planning routine.
vector<double>	`costs` Graph costs used to generate current plan.
list<Edge*>	`path` Current path as an ordered list of edges.

3.2 Agents

The `Agents` library contains class definitions for learning agents of various domains (see more in Section 3.1). For the `testWarehouse` program, we use only `Link` and `Intersection` agents, which inherit completely from the generic `Agent` class. Note that the `Centralised` agent is implemented as an `Intersection` agent under the case where all edges in the graph fall under the control of the one agent.

3.2.1 Agent class

The `Agent` class is currently only designed for fully connected neural network policies with a single hidden layer. The constructor requires the population size and network architecture, as specified by the number of input nodes, output nodes and hidden nodes. This class provides basic housekeeping for agent simulation and neuro-evolution functionality:

Public member functions	
	`Agent`(size_t, size_t, size_t, size_t) Constructor. Takes in the population size as well as the number of input, output and hidden nodes for the agent's fully connected neural network policies.
void	`ResetEpochEvals`() Resets the evaluation (fitness) vector to zeros. Should be called at the start of each new epoch.
Eigen::VectorXd	`ExecuteNNControlPolicy`(size_t, Eigen::VectorXd) Queries a specific policy in the agent's population, the index of the policy is provided in the first input and the second input is the agent's current state.
void	`SetEpochPerformance`(double, size_t) Assigns the evaluation (fitness) of a specific policy, the evaluation is provided in the first input, the index of the policy is provided in the second input.

vector<double>	GetEpochEvals() Returns the policy evaluations of the current epoch.
void	EvolvePolicies(bool = false) Triggers competition (default) and mutation of the population. Typically competition is only skipped during the first epoch.
void	OutputNNs(std::string) Writes all neural network weights to file.
NeuroEvo*	GetNEPopulation() Returns a pointer to the agent's neuro-evolutionary population.
size_t	GetNumIn() Returns the number of input nodes.
size_t	GetNumHidden() Returns the number of hidden nodes.
size_t	GetNumOut() Returns the number of output nodes.
	~Agent() Destructor.
Protected attributes	
size_t	popSize
size_t	numIn
size_t	numOut
size_t	numHidden
std::vector<double>	epochEvals
NeuroEvo*	AgentNE

3.3 Learning

Neuro-evolution is implemented through the `NeuroEvo` and `NeuralNet` classes. Currently, the code only supports single hidden layer, fully connected neural networks. Furthermore, the mutation rate and noise are fixed variables in the `NeuroEvo` class. A number of functions within these classes are migrated from https://github.com/rebhuhnc/libraries/tree/master/SingleAgent [5].

3.3.1 NeuroEvo class

Public member functions	
	NeuroEvo(size_t, size_t, size_t, size_t, actFun = TANH) Constructor. Takes in the population size as well as the number of input, output and hidden nodes for the agent's fully connected neural network policies. The final input specifies the activation function to use {LOGISTIC,TANH (default)}
void	MutatePopulation() Doubles the population size by adding neural networks with mutated weights of existing neural networks.
void	EvolvePopulation(std::vector<double>) Evolves population according to evaluation signal input and survival function.
std::vector<double>	GetAllEvaluations() Returns the evaluation for all current members in the population (used for debugging).
NeuralNet*	GetNNIndex(size_t) Returns the policy corresponding to the input index.
size_t	GetCurrentPopSize() Returns the size of the population vector.
void	SetMutationNormLog(bool = true) Sets the program to log (default) or not log the mutation norms.

std::vector<double >	`GetMutationNorm()` Returns the Frobenius norm of the difference between the network weight matrices before and after mutation.
	`~NeuroEvo()` Destructor.
Private member functions	
void	`BinaryTournament()` Evolutionary competition method that randomly compares pairs within the population (without resampling) and keeps only the best member.
void	`RetainBestHalf()` Evolutionary competition method that keeps only the highest performing half of the population.
static bool	`CompareEvaluations`(NeuralNet*, NeuralNet*) Comparitor function to sort neural networks according to evaluation signal (must have strict weak ordering).
double	`ComputeFrobeniusNorm`(Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd) Computes the summed Frobenius norms of the two pairs of weight matrices.
Private attributes	
size_t	`numIn`
size_t	`numOut`
size_t	`numHidden`
actFun	`activationFunction` Either LOGISTIC or TANH.
size_t	`populationSize`
std::vector<NeuralNet*>	`populationNN` Current population of neural networks.
void	`(NeuroEvo::*SurvivalFunction)()` Function handle to the evolutionary competition procedure.
bool	`computeMutationNorms`
std::vector<double>	`mutationFrobeniusNorm`

3.3.2 NeuralNet class

Public member functions	
	`NeuralNet`(size_t, size_t, size_t, actFun = TANH, nnOut = BOUNDED) Constructor. Takes in the number of input, output and hidden nodes for a single hidden layer, fully connected neural network. The final two inputs specify the activation function to use LOGISTIC,TANH (default), and whether or not to apply (default: BOUNDED) the activation function to the final layer or not (UNBOUNDED).
Eigen::VectorXd	`EvaluateNN`(Eigen::VectorXd) Evaluates a forward pass of the neural network given the input vector.
Eigen::VectorXd	`EvaluateNN`(Eigen::VectorXd, Eigen::VectorXd&) Evaluates a forward pass of the neural network given the input vector. It also stores the hidden nodes in the container provided as the second input.
void	`MutateWeights()` Mutates the weights of the neural network according to the mutation rate using mutation noise drawn from $\mathcal{N}\left(0, \texttt{mutationStd}^2\right)$.
void	`SetWeights`(Eigen::MatrixXd, Eigen::MatrixXd) Assigns weight matrices, the first input matrix is used as the weights connecting the input and hidden layers, the second input matrix is used as the weight connecting the hidden and output layers.
Eigen::MatrixXd	`GetWeightsA()` Returns the network weights connecting the input and hidden layers.

Eigen::MatrixXd	`GetWeightsB()` Returns the network weights connecting the hidden and output layers.
void	`OutputNN`(const char*, const char *) Wrapper for writing neural network weight matrices to specified files.
double	`GetEvaluation()` Return the evaluation (fitness) of the neural network.
void	`SetEvaluation`(double) Store the input value as the network's evaluation (fitness).
void	`BackPropagation`(std::vector<Eigen::VectorXd>, std::vector<Eigen::VectorXd>) Performs backpropagation on the network weights (not fully tested).
	`~NeuralNet()` Destructor.
Private member functions	
void	`InitialiseWeights`(Eigen::MatrixXd&) Initialises the neural network weights to random values.
Eigen::VectorXd	`HyperbolicTangent`(Eigen::VectorXd, size_t) Hyperbolic tan activation function. Outputs between [-1,1].
Eigen::VectorXd	`LogisticFunction`(Eigen::VectorXd, size_t) Logistic activation function. Outputs between [0,1].
double	`RandomMutation()` Generates random mutation noise `mutationRate%` of the time.
void	`WriteNN`(Eigen::MatrixXd, std::stringstream&) Writes the values of the specified weight matrix to the specified file.
Private attributes	
double	`bias` Additional fixed bias node in the hidden layer, set to 1.0.
Eigen::MatrixXd	`weightsA`
Eigen::MatrixXd	`weightsB`
double	`mutationRate` Set to 0.5.
double	`mutationStd` Set to 1.0.
double	`evaluation`
double	`eta` Learning rate for backpropagation.
std::vector<size_t>	`layerActivation`
Eigen::VectorXd	`(NeuralNet::*ActivationFunction)(Eigen::VectorXd, size_t)` Function handle to the network activation procedure.

3.4 Planning

The `Planning` library contains all the classes related to performing Dijkstra's algorithm for path search on a graph. The `Graph` and `Search` classes are the entry points, the former stores all connectivity and cost information related to the graph while the latter performs the search itself. The output path is stored as a link list connecting the goal vertex to the start vertex.

3.4.1 Search class

Public member functions	
	`Search`(Graph*, int, int) Constructor. Takes in the search graph, start and goal vertices.
Graph*	`GetGraph()` Returns the search graph.

Queue*	`GetQueue`() Returns the priority queue.
void	`SetQueue`(Queue*) Assigns the priority queue.
int	`GetSource`() Returns the index of the start vertex ID.
void	`SetSource`(int) Assigns the start vertex ID.
int	`GetGoal`() Returns the goal vertex ID.
void	`SetGoal`(int) Assigns the goal vertex ID.
Node *	`PathSearch`() Main A* search routine. Returns the path as a link list of Nodes. The output Node pointer is associated with the goal vertex and tracing along the link list will end up at the Node associated with the start vertex.
void	`ResetSearch`() Deletes the search priority queue.
	`~Search`() Destructor.
Private member functions	
size_t	`FindSourceID`() Find the source vertex index given the vertex ID.
Private attributes	
Graph*	`itsGraph`
Queue*	`itsQueue`
int	`itsSource`
int	`itsGoal`

3.4.2 Queue class

Public member functions	
	`Queue`(Node *) Constructor. Takes in the start node.
std::vector<Node*>	`GetClosed`() Returns the closed set.
bool	`EmptyQueue`() Returns true if the priority queue is empty.
size_t	`SizeQueue`() Returns the size of the priority queue.
void	`UpdateQueue`(Node *) Pushes the input node into the priority queue.
Node*	`PopQueue`() Returns the top Node* entry in the priority queue, which is placed into the closed set and popped off the priority queue.
	`~Queue`() Destructor.
Private attributes	
QUEUE	`itsPQ` typedef std::priority_queue<Node*, std::vector<Node*>, CompareNode>QUEUE See associated objects for priority queue comparitor.
std::vector<Node*>	`closed`
Associated objects	

struct	`CompareNode` Comparitor used to order the priority queue.

3.4.3 Graph class

Public member functions	
	`Graph`(std::vector<int>&, std::vector<std::vector<int>>&, std::vector<double>&) Constructor. Takes in the vertices, edges and edge costs.
std::vector<int>	`GetVertices`() Returns the set of graph vertices.
std::vector<Edge*>&	`GetEdges`() Returns the set of graph edges.
size_t	`GetNumVertices`() Returns the number of vertices in the graph.
size_t	`GetNumEdges`() Returns the number of edges in the graph.
size_t	`GetEdgeID`(Edge*) Returns the index of the input edge.
std::vector<Edge*>	`GetNeighbours`(Node *) Returns all non-ancestor edges connected to the input Node.
	`~Graph`() Destructor.
Private member functions	
void	`GenerateEdges`(std::vector<std::vector<int>>&, std::vector<double>&) Creates a vector of Edge objects from the input graph configuration variables.
Private attributes	
std::vector<int>	`itsVertices`
std::vector<Edge*>	`itsEdges`
size_t	`numVertices`
size_t	`numEdges`

3.4.4 Node class

Public member functions	
	`Node`(int) Constructor. Defined by its associated vertex's ID.
	`Node`(int, nodeType) Constructor. Defined by its associated vertex's ID and identifies if it represents the start vertex of a path.
	`Node`(Node*, Edge*) Constructor. Defined by its parent node and the edge connecting it to the parent node.
Node *	`GetParent`() const Returns the parent Node in the link list.
void	`SetParent`(Node*) Assigns the parent node.
double	`GetCost`() const Returns the cost to reach the associated vertex from the start of the path.
void	`SetCost`(double) Assigns the cost to reach the associated vertex from the start of the path.
int	`GetVertex`() const Returns the ID of its associated vertex.
void	`SetVertex`(int) Assigns the associated vertex's ID.
void	`DisplayPath`()

	Prints the path from the associated vertex to the start vertex to the command line.
Node*	`ReverseList`(Node*) Reverses the link list and returns the new Node that points to the start of the list.
	`~Node`() Destructor.
Private attributes	
int	`itsVertex` The associated vertex's ID.
Node*	`itsParent`
double	`itsCost`

3.4.5 Edge class

Public member functions	
	`Edge`(int, int, double) Constructor. Takes in the parent and child vertex IDs and the base cost of traversal. Note that the edge length is assigned by casting this value to a size_t.
int	`GetVertex1`() const Returns the parent vertex ID.
int	`GetVertex2`() const Returns the child vertex ID.
double	`GetCost`() const Returns the cost of traversal.
void	`SetCost`(double) Assigns the cost of traversal.
size_t	`GetLength`() Returns the length of the path (number of timesteps required to traverse).
friend bool	`operator==(const Edge&, const Edge&)` Returns true if the input Edge has matching parent and child vertex IDs.
	`~Edge`() Destructor.
Private attributes	
int	`itsVertex1` The associated parent vertex's ID.
int	`itsVertex2` The associated child vertex's ID.
double	`itsCost`
size_t	`itsLength`

3.5 Utilities

This library contains a number of small helper functions that are often used throughout the code base. These functions reside within the `easymath` namespace.

3.5.1 The easymath namespace

Namespace	`easymath`
double	`rand_interval`(double, double) Returns a random number between the two input values.
double	`pi_2_pi`(double) Normalises angles between $\pm\pi$.
double	`sum`(std::vector<double>) Sums elements in a vector.

4 Data post processing

Two example post processing MATLAB files are provided inside the `multiagent_learning/Domains` folder.

- `postProcess.m`: produces plots the various metrics logged over the evolutionary epochs, e.g. team performance (total number of successful deliveries), travel times, etc. It also provides violin plots comparing the distributions of the final team performances for each of the requested agent types. An example of the plots generated by this function can be found in [2] (figures 4-6). It depends on the `distributionPlot` [4] and `rotateticklabel` [1] functions, which can be found on MathWorks File Exchange.
- `simReplay.m`: plots the given traffic graph and replays the specified log files, showing the positions of all AGVs, as well as the costs and capacity states of each edge.

These functions have been tested with Matlab2017b.

References

[1] Andrew Bliss. Rotate tick label. https://mathworks.com/matlabcentral/fileexchange/8722-rotate-tick-label, 2005. Accessed 16-11-2018.

[2] Jen Jen Chung, Damjan Miklic, Lorenzo Sabattini, Kagan Tumer, and Roland Siegwart. The impact of agent definitions and interactions on multiagent learning for coordination. In *Proceedings of the 18th International Conference on Autonomous Agents and Multiagent Systems*, 2019. Under review.

[3] Jen Jen Chung, Carrie Rebhuhn, Connor Yates, Geoffrey A. Hollinger, and Kagan Tumer. A multiagent framework for learning dynamic traffic management strategies. *Autonomous Robots*, pages 1–17, 2018. Online first.

[4] Jonas. Violin plots for plotting multiple distributions (distributionplot.m). https://mathworks.com/matlabcentral/fileexchange/23661-violin-plots-for-plotting-multiple-distributions-distributionplot-m, 2017. Accessed 16-11-2018.

[5] rebhuhnc. libraries. https://github.com/rebhuhnc/libraries, 2016. Accessed 16-11-2018.