AbstractUTMSimulation C++ Project Documentation

Brandon Gigous

July 6, 2017

1 Introduction

This document describes the high-level behavior of the agents and the traffic in the abstract UAV traffic management (UTM) domain. The first several sections go over the main classes used in the C++ code. Later, described is the configuration file which contains parameters that can be changed for each experiment.

1.1 Legend

Items with the `Courier` font style indicate a class or function in the code. Items in *italics* indicate a configuration parameter, which are all detailed in Section 13. Items by themselves in parentheses refer to the number of the configuration parameter.

2 AbstractUTMSimulation.cpp

This is the entry point of the program, where the `main()` function is. It is where the configuration file is loaded and where the simulation is initialized and started. The domain and the agents are also created.

3 SimNE

The `SimNE` class controls the overall simulation. Before a simulation can be run, this class needs as input the domain, the multiagent system, and the configuration parameters (from the configuration file). The simulation is started via its `runExperiment()` function (next section).

3.1 runExperiment()

This function starts a single run of training. It trains the agents for a certain number of *epochs* (**t.1**). Calls the `epoch()` function each epoch, which is explained in the next section.

3.2 epoch()

Called by `runExperiment()` every epoch. New neural networks for each agent are generated (mutation), then $2 \times numAgents$ simulations are run, one for each team of networks. The domain (Section 5) is reset each time. The domain is simulated with traffic in `simGlobal()`. After each simulation, the performance of the team G is computed (sum of partial objective values). Fitness values are also assigned to each agent in the team, however this documentation considers agents receiving only G, the global performance.

As the function goes through all the teams, it keeps track of the best performing team as `best_run_performance`. Every time this variable gets updated, extra information about delay time, moving time, and wait time achieved by this best team is also recorded.

Once the function has gone through every team, the CCEA (`MultiagentNE`, Section 12) then chooses the survivors. In other words, it eliminates the $1 \times numAgents$ worst teams. Performance data is also collected which is later written to disk (refer to Section 3.5).

3.3 simGlobal()

This function takes the current team of agent policies and simulates them in the environment. It returns a data structure (`SimHistory` in Section 4, refered to here as SH) that stores historical data on performance of the agents. **For each step** t **of the simulation** (total of *steps* (**t.3**)):

- the states S_t of the agents are (determined from current traffic) are computed (`getStates()`, Section 5.7)
- agents take actions A_t based on their neural network policies with the corresponding states as input (`getActions()`, Section 3.4)
- the simulation is advanced one timestep and a partial objective value g_t is returned (refer to `simulateStep()` for details, Section 5.9); extra information e_t about the domain is also captured from `getExtraInfo()` (Section 5.13)
- The data (g_t and e_t) is recorded in SH (for timestep t)

If *tracking* (**m.19**) is enabled, after fully simulating, data about the domain is written to disk (see Section 14.2).

3.4 getActions()

Simply calls the multiagent system's `getActions()` function, which, for the active team, computes each agent's action given each agent's state via neural network feedfoward propagation.

3.5 outputMetricLog()

This is inherited from the `ISimulator` class. Given performance information stored in the `metric_log` member variable, writes to disk a .csv file that contains the best performance achieved for each epoch.

3.6 outputExtraMetrics()

Outputs to a .csv file the total delay time, moving time, and wait time achieved by the best performing team (adding up gives total travel time) for every epoch of the run. Also gives total UAVs generated for each of these epochs.

4 SimHistory

This is a struct used to record performance of a team across a simulation of the airspace. For each timestep, the state of the agents, actions of the agents, and the performance of the agents (for that timestep, which includes information about delay, moving, and wait time) is logged. This information is used later to determine the best performance of all teams for the epoch.

5 UTMDomainAbstract

This class defines a representation of the airspace and the mechanics thereof. It constructs the airspace including the nodes and edges, and imposes edge costs onto the edges reflective of the agents' actions. It is responsible for generating UAVs, assigning them goal nodes, calling on them to plan paths through the graph, and removing UAVs as they reach their destination nodes.

This class requires as input the configuration parameters and a string which represents which type of approximation for G to use. This second parameter, however, is not taken into account in this documentation and can be ignored. As mentioned elsewhere, only the global performance is considered for the agent rewards in this documentation.

5.1 Initialization

Note that there are two constructors in the `UTMAbstractDomain` class. The second constructor calls the first constructor.

When the class is first created (first constructor), the `initializeBase()` function is called (Section 5.2). The configuration parameters are parsed and the Domain and Tracker directories are created (see Section 14 for more info on these directories). The tracker object is also created (only used if *tracking* (**m.19**) is enabled). If *airspace* (**m.4**) is set to "generated", new files (described in Section 14.1) are created using `generateNewAirspace()` (Section 5.4) only if they do not already exist. Regardless of how the nodes and edges files are created, whether by human or generated, these files are read by the class and a graph is constructed. This includes creating the high-level representation (`LinkGraph`, briefly covered in Section 7) as well as the low-level representation, which means constructing all the individual `Link`'s using the `addLink()` function (see Section 8 for `Link` class). Each `Link` has its individual capacity assigned according to the *capacity* (**m.17**) parameter. Each `Link` also has its connections defined, that is, all the links incoming to the `Link`.

Next, the entities representing the agents are created. If *agent* (**m.7**) is set to "sector", the agents are represented in a class instance of `SectorAgent`. Otherwise, if set to "link", a `LinkAgent` (Section 9) instance is created.

The generation and destination sectors are then defined according to *generation* (**m.16**) and *destination* (**m.15**) respectively.

In the second constructor, each of the sectors are given a list of destination sectors, meaning that UAVs generated at this sector can be given a goal to one of these other sectors. This list of sectors does not include the sector in question and it depends on the setting of *destination* (**m.15**). Finally, if *position* (**m.6**) is "constant", UAV are generated according to an initial pose file. (Constant position mode is not usually used.)

5.2 initializeBase()

Based on *agent* (**m.7**) and *state* (**m.18**) as well as the domain files (Section 14.1), computes how many agents and states there are for the experiment and records these quantities.

5.3 addLink()

Given an edge with source and target nodes and a capacity, this function creates a new `Link` (Section 8) and adds it to the list of links. The direction of the link and Euclidean distance are computed from the geometry of the high-level representation.

5.4 generateNewAirspace()

Generates a completely new airspace according to an algorithm. First, nodes are created and randomly placed in the airspace. The number of nodes placed is determined by *sectors* (**c.1**). The algorithm then randomly connects nodes in the graph, until it cannot make another connection without overlapping two edges. The point of this function is to create a graph that still represents a valid abstraction of an obstacle map, that is, a map not created by a human.

5.5 getPerformance()

Returns the global performance achieved by the agents, depending on *objective* (**m.10**).

5.6 getRewards()

Calculates rewards for each agent, returned as a 1D matrix. If *fitness* is set to “global”, the global performance is used as the reward for all agents. Other modes of reward calculation can be defined. See Section 13.1.

5.7 getStates()

Returns the state of the agents, at the current timestep, as a 2D matrix. This involves simply calling the `computeCongestionState()` function (Section 9.2), which is different depending on what *agent* (**m.7**) is defined as.

5.8 incrementUavPath()

Called from `simulateStep()` (Section 5.9). UAVs, if possible, travel for one timestep along their planned paths. Keeps track of how many UAVs were able to move (`moving`) and how many were delayed (both on a link (`delay`) and at origin sector (`ground_hold`)).

It first attempts to add UAVs waiting in their origin sectors onto the links in their individual paths. If successful, the UAV is removed from the sector and its status is no longer “waiting.” Otherwise, it remains at the sector. Next, it considers UAVs that have already left their origin sectors. For those that are currently traversing a link and have not yet reached the end, each of these UAVs are moved further along their links. For those that have reached the end of their current link, an attempt is made to add them to the next link in their planned path. If unsuccessful, the UAVs are delayed one timestep.

UAVs replan their paths in two cases:

- if a UAV is still waiting at origin sector when the function is called,before an attempt is made to add it to a link
- if a UAV gets delayed, meaning it could not be successfully added to the next link in it’s (previous) path plan

5.9 simulateStep()

This function takes the agent actions as inputs and advances the simulation one timestep. The actions of the agents are used to update the costs of the individual edges in the airspace (see Section 7). The `actionsToWeights()` (Section 9.3) function is this transformation between agent actions and edge costs. Next, the UAVs that have reached their destinations are removed from the airspace as described in Section 5.11. New UAVs are then generated (if applicable) as described in Section 5.12. If the cost map changed from the last time this function was called, the path plans of UAVs not currently traveling are updated (`getPathPlans()`). The UAVs then travel on their paths (`incrementUavPath()`, Section 5.8). If *tracking* (**m.19**) is enabled, the tracker is updated with information about the current step. Finally, if the links use an enriched state representation (*state* (**m.18**) is set to “incoming”), the links’ data structures are updated.

The function returns the value of the (partial) objective for the timestep.

5.10 reset()

Resets the domain. All UAVs are removed from the airspace, and all links and sectors are reset. Additionally, the tracker is cleared.

5.11 absorbUavTraffic()

Removes UAVs that have reached their destination (unless *disposal* (**m.11**) is set to “keep”). The condition for a UAV being at its destination sector is that it must have reached the end of a link that connects to the destination node.

5.12 getNewUavTraffic()

There are two versions of this function, one where there are no arguments and one which accepts the ID of a sector to generate UAVs at. The former is called every timestep from `simulateStep()` (Section 5.9). This function iterates through all the generation sectors and calls their `generateUavs()` function (which may or may not generate UAVs, see Section 6.2).

5.13 getExtraInfo()

This function returns the total moving time, delay time, and wait time of all UAVs in airspace for the last timestep. This is returned as a 1D matrix. (UAV count is also included, but this only matters at the last timestep.)

6 Sector

This class represents a node in the airspace graph. It has the ability to generate UAVs and assigns them goal nodes. UAVs may travel from one node to another node via links.

6.1 Initialization

To create a sector agent, the following information must be given: its location in the airspace, its ID, the IDs of the other sectors it connects to, the locations of all sectors, the configuration parameters, the high-level graph, and the IDs of all other sectors that can assigned to generated UAVs as goals,

The constructor is where the behavior of UAV generation is determined by the sector (**m.5** and **m.14**). The number of UAVs to generate is also determined (**c.6**).

6.2 generateUavs()

This function is used to add UAVs to the airspace. It takes as input the current timestep of the simulation, as well as the current count of total generated UAVs (as a pointer). Although this function is called every timestep, UAVs are only generated if certain conditions are met. The function calls `shouldGenerateUAVs()` to determine if *num_generated* (**c.6**) UAVs should be generated that timestep, which returns true depending on the *traffic* (**m.5**) mode:

- **constant:** Never returns true. UAVs are instead only generated at the beginning of the simulation.
- **deterministic:** Returns true if timestep is 0 or is a multiple of the *generation_rate* (**c.5**).
- **probabilistic:** Returns true with *generation_probability* $\times$ 100% chance (**c.4**).

7 LinkGraph

This class is a high-level representation of the airspace graph. Each time the link agents calculate their actions (`getActions()`, Section 3.4) and output a cost (`actionsToWeights()`, Section 9.3), this class assigns these costs to its own representation such that it can be utilized by the `UAV` class (Section 10) to determine a UAV's path plan.

8 Link

This class represents a directed edge (link) in the abstract UTM domain, and also encapsulates some of the the low-level mechanics of the domain.

8.1 Initialization

To create an instance of the class, an ID number must be given, as well as: the source and target node IDs, the time it takes to traverse the link, the capacity of the link, the cardinal direction (indicated by an integer), its window size (*window_size* (**c.7**)), and optionally, if the window mode is cumulative (*window_mode* (**m.20**)). This information is given by the domain (Section 5).

Other than initializing the list of current traffic on the link and setting some other initial values, the constructor does not do anything else. Depending on the window settings (**m.20** and **c.7**), however, further initialization may be done in the `initHist()` function, discussed in Section 8.7.

8.2 grab()

Takes a specific UAV and another link (we'll call **L**) as input. The link attempts to `add()` (Section 8.3) the UAV, and if successful, the UAV is removed from **L** (via its `remove()` function, Section 8.4) and the function returns true. Otherwise, the function returns false and the UAV is not removed from **L**.

8.3 add()

Attempts to add a specified UAV to the link. This function returns false if unsuccessful (link at capacity), or true if the UAV was successfully added.

8.4 remove()

Removes a specified UAV from the link.

8.5 predictedTraversalTime()

Returns the predicted amount of time it would take to cross the link. This is the addition of the traversal time of the link and the instantaneous wait time at the link.

8.6 Time Window

The use of an optional time window is meant to scale the second state input, which gives information about any incoming traffic from other links that connect to the link (only applicable when *state* (**m.18**) is set to "incoming"). For simplification, let $\delta = \mathit{window\_size}$ (**c.7**). The i_{th} link L_i keeps track of, from the last δ timesteps:

- the number of UAVs that reached the end of L_i and got added to another link (or absorbed in the case that a UAV reaches its destination). This quantity is denoted as exit_i.
- the number of UAVs that directly enter L_i from $L_{j \neq i}$. This quantity is denoted as $\mathit{enter}_{i,j}$.

Using these two quantities, we can update the ratio of UAVs that enter L_i from incoming link L_j as:

$$p_{i,j} = \frac{\mathit{enter}_{i,j}}{\mathit{exit}_j} \tag{1}$$

In the case that exit_j is zero, $p_{i,j}$ is not updated. Before UAVs are generated, all probabilities are initialized to p_{init} (set to 1.0 by default).

See also the `LinkAgent` (Section 9) class for more info on the scaling of the input.

8.7 initHist()

This function initializes a few data structures used by the link. The function only creates these structures if the link uses a time window to record historical, as specified in *window_mode* (**m.20**) and *window_size* (**c.7**). If *window_mode* is set to "variable" and *window_size* is set to 0 (zero), no time window is used.

If a "variable" *window_mode* is specified, three structures are created for each link. Notation L_i and L_j for the links will be used again as in the previous section.

- **incoming ratio:** This is a mapping from an incoming link L_j's source node to the ratio of UAVs that have entered L_i from L_j. This indirectly represents $p_{i,j}$ shown earlier.
- **traffic history:** This is represented as a list of size δ which holds how many UAVs exited L_i (to go on another link) for each of the last δ timesteps. The sum of the numbers in the list is $exit_i$.
- **incoming from:** This is represented as a mapping from an incoming link L_j's source node to a list of size δ which holds how many UAVs entered L_i from L_j for each of the last δ timesteps. The sum of the numbers in a list gives the corresponding $enter_{i,j}$.

For a *window_mode* of "cumulative", the **incoming ratio** data structure is created just the same. For **traffic history** and **incoming from**, the lists are replaced with a single number which represents the sum of the quantities mentioned. For example, **traffic history** would simply be how many total UAVs exited L_i since the beginning of the simulation.

8.8 How the Structures Work

To illustrate what these structures do, imagine a toy simulation shown in Figure 1. There are four UAVs traveling on the links in a simple airspace with four nodes. One UAV gets on L_5, then two UAVs jump on L_1. Let's imagine $\delta = 2$ and that we ignore capacity for now. To calculate the **incoming ratio** of L_1 from L_2 and L_4 ($p_{1,2}$ and $p_{1,4}$) for each timestep, we must determine the **traffic history** of L_2 and L_4 and the **incoming from** of L_1 (L_6 is also an incoming link, but for this example we will ignore it). Below is a table of the values of each of these structures.

t	traffic history of L_2	traffic history of L_4	incoming from L_2	incoming from L_4
0	{0, 0}	{0, 0}	{0, 0}	{0, 0}
1	{0, 0}	{0, 0}	{0, 0}	{0, 0}
2	{1, 0}	{0, 0}	{0, 0}	{0, 0}
3	{1, 1}	{1, 0}	{1, 0}	{1, 0}
4	{0, 1}	{0, 1}	{0, 1}	{0, 1}

The simulation at timestep $t = 4$ is not shown in the Figure, but it is shown in the table to demonstrate the effect of sliding the time window. It is assumed that at $t = 4$, no other UAVs are generated and none of the existing UAVs enter another link or reach their goal.

The **incoming ratio** of L_1 is then:

t	incoming ratio (from L_2)	incoming ratio (from L_4)
0	1.0	1.0
1	1.0	1.0
2	0.0	1.0
3	0.5	1.0
4	1.0	1.0

Note that the ratios are initially set to 1.0, as explained in The **incoming ratio** of L_1 from L_4 appears unchanged. However, be aware that the ratio is updated at $t = 3$, even though the quantity remains the same.

8.9 updateIncomingHist()

Called from `grab()` (Section 8.2). Takes a source node as input. This source node and the source node of the link (note the distinction) define the source and target of the incoming link, respectively. The record of how many UAVs entered the link from this incoming link (**incoming from**) is updated.

8.10 updateTrafficProb()

Computes $exit_i$ and $enter_{i,j}$ mentioned in Section 8.6. The link's corresponding ratios ($p_{i,j}$) are updated here, if applicable.

8.11 slideWindow()

If *window_mode* (**m.20**) is "variable", the values in the lists of the data structures are shifted by one and a zero is inserted for the new value.

8.12 reset()

This function reinitializes the link such that it has no UAVs on it. It also reinitializes the other data structures.

9 LinkAgent

This class provides an interface for link agents to interact with the simulator. It translates link agent actions into weights and also computes the states of the agents. Despite the name, it represents all link agents.

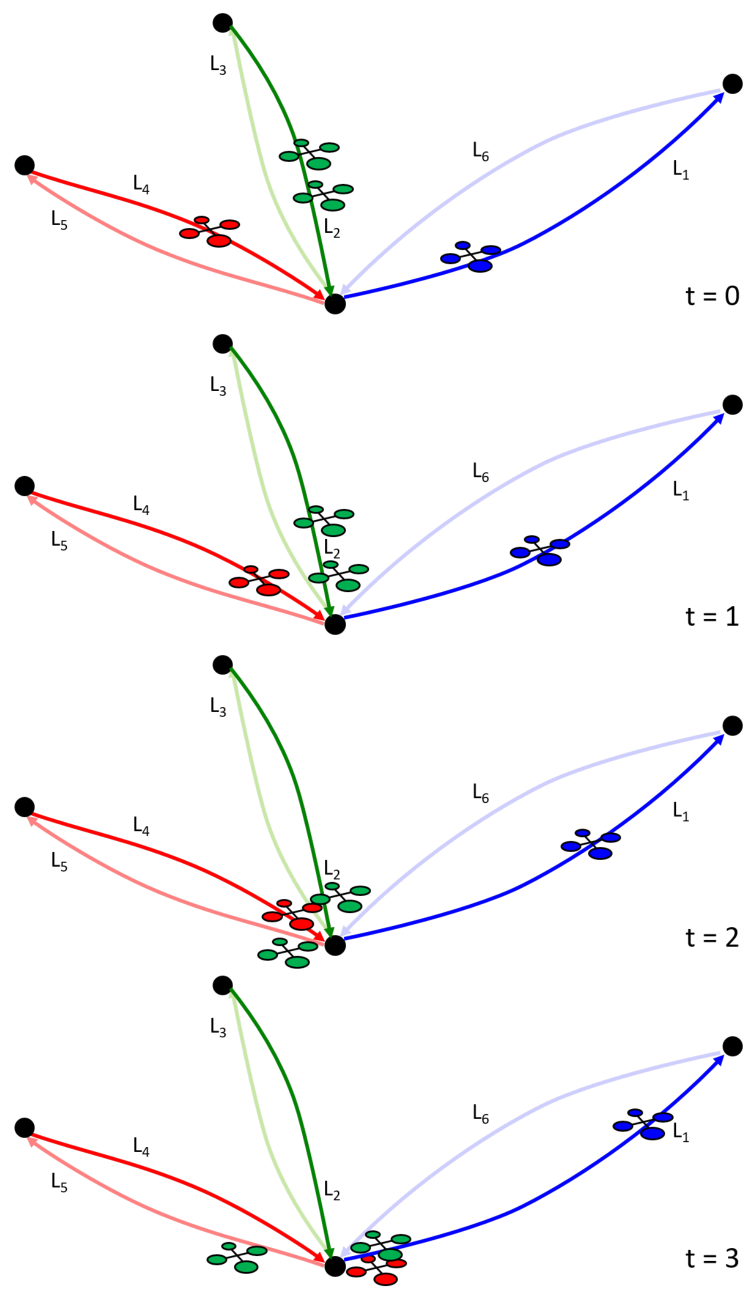

Figure 1: Toy simulation showing the transitions of UAVs over

9.1 Initialization

This class requires as input the number of edges in the airspace graph, a list of all links in the airspace, and the number of state elements for the agents.

When an instance is created, the class creates a mapping from a source and target node to the corresponding link's ID. This only includes nodes for which a connection between them exists. A second mapping is created that maps a source node to all other nodes that it connects to. This is used later to help compute the state of the link agents in the case that the enriched, or "incoming," *state* (**m.18**) representation is used.

9.2 computeCongestionState()

This function computes the state of all link agents given all UAVs currently in the system. There are two possible state representations:

- **Simple** – each agent has access only to information about its own edge's traffic (1 state, *state* (**m.18**) set to "traffic")
- **Enriched** – each agent has access to information about its own edge's traffic as well as information about potential incoming traffic (2 states, *state* (**m.18**) set to "incoming")

For the **Simple** state representation, this function simply calculates each link agent's state as the number of UAVs it currently has traveling on its edge. For the **Enriched** state representation, in addition to the previous calculation, also calculates the number of UAVs incoming from other links with possible scaling. Assuming we consider the link agent we're calculating state for as L_i, and an incoming link is denoted $L_{j \neq i}$, then the second input state ϕ_i is calculated as $\phi_i = \sum_j^N p_{i,j} \times \zeta_j$, where ζ_i is the traffic (number of UAVs) on L_i and N is the number of incoming links to L_i. See Section 8.6 for the definition of $p_{i,j}$.

9.3 actionsToWeights()

Scales the actions of each link agent to costs that are then applied to the links in the graph which the UAVs see when they plan their paths (Section 7). Included in the cost is the "predicted" cost of the link, computed as the traversal time plus the instantaneous wait time at that link (see `predictedTraversalTime()` in Section 8.5). Added to this predicted cost is the action of the agent (output of neural network) times the *alpha* (**c.2**) value. If the output of the neural network is negative, the cost of the link is simply the predicted cost. The return value of this function is a 1D matrix that represents the cost of all links.

10 UAV

This class represents an individual UAV in the airspace. Each UAV interacts with the environment through planning, and is also responsible for keeping track of how close it is to the end of a link and which sector it currently considered a member of. Planning is currently done with A* search using the Boost library. See Section 15 for hints on how to change the planning behavior of UAVs.

Each UAV keeps an internal model of the path it plans to take as it travels through the graph. The struct `Path` in the class is used to hold the model and also modify it as the UAV reaches subgoals in its path.

10.1 Initialization

To create a UAV, the following information must be given: the configuration parameters, the UAVs start sector, end sector, the high-level graph, and the UAV's ID. This information is supplied from the `Sector` that generates it.

10.2 planAbstractPath()

This function makes the UAV replan its path, which is stored internal to the `UAV` class. Currently, only A* is implemented. The function returns true if the UAV's path plan did not change from the previous iteration.

11 NeuroEvo

This class represents a neuroevolutionary algorithm. For the application described in this documentation, it is used to keep and evolve a population of neural network policies for a learning agent.

12 MultiagentNE

This class represents the cooperative coevolutionary algorithm (CCEA) used to simultaneously evolve a team of agents (which use the `NeuroEvo` class). This class automatically assigns teams of agent policies and one can cycle through them using `set_next_pop_members()` function (which returns false when all teams have been cycled through).

13 Configuration Parameters

The configuration file (config.yaml) in the AbstractUTMSimulation directory is a file written in the YAML format (https://en.wikipedia.org/wiki/YAML). This specific file uses only associative arrays to define the various fields used in the UTM Simulation. The yaml-cpp library is used in the code to parse these fields. (Please note that my terminology of YAML stuff might be incorrect, but this should give you the gist.)

The fields in the configuration file are presented as a hierarchy. Each subfield is indented by a single space. Each subsubfield is indented by two spaces. And so on.

There are five main fields: modes, neuroevo, approximator, time, and constants. Each contains various parameters. Note: when files are mentioned, refer to Section 14.

- **modes:** This field defines several parameters for changing the high-level behavior of the simulation

m.1 *resimulation* - not currently used in code

m.2 *fitness* - how the agents are given fitness

- global: agents in a team are awarded the global fitness collectively achieved by all agents (this is the only mode considered in this documentation; more parameter values can be added as described in Section 13.1)

m.3 *prop_record* - true if UAV generation for each sector is to be recorded in files (experimental); applies only when *traffic* (**m.5**) is set to "probabilistic"

m.4 *airspace* - this tells the program how it should create the airspace

- saved: load a predefined airspace
- generated: use the generated graph, or if it has not been created yet, generate a new one (see Section 5.4)

m.5 *traffic* – how UAV traffic is generated in the airspace

- deterministic: UAVs are generated at each generation node every few timesteps (see **c.5**)
- probabilistic: UAVs are generated at a generation node with a certain probability (see **c.4**)
- constant: referenced in code, but not used
- playback: UAVs are generated according to a series of files (experimental)

m.6 *position* - not considered in this documentation

m.7 *agent* - the agent formulation to use for learning to reduce congestion

- sector: use sector agents; this is not recommended (because it will probably break the program as of this writing)
- link: the agents are link agents; the number of agents is equal to the number of edges in the graph

m.8 *search* - the algorithm the UAVs use to plan their paths

- astar: UAVs use A* to plan their paths
- ¡other¿: another option for UAV path planning (See Defining New Parameters, Section 13.1, as well as Section 15)

m.9 *square* - not used in code

m.10 *objective* - the objective that the learning agents attempt to optimize

  - delay: the objective is to reduce total delay time of all UAVs (time delayed on a link before entering another)
  - travel_time: the objective is to reduce total travel time of all UAVs; this includes moving time (time spent actually moving in graph), delay time (time delayed on a link before entering another), and wait time (time delayed at origin sector)

m.11 *disposal* - how UAVs should be dealt with when they reach their destinations

  - keep: UAVs stick around after reaching their destination; this is meant more for hardware tests, and is probably not operational at the time of this writing
  - trash: UAVs are removed from the domain as soon as they reach their destination; in this way, UAVs that are "done" don't contribute any longer to the congestion in the system

m.12 *types* - not currently used in code

m.13 *numbered_domain* - not currently used in code

m.14 *destinations* - how goal nodes (destination sectors) are assigned to the UAVs by the sectors that generate them

  - static: the goal nodes assigned to UAVs are assigned in a cyclical manner; for example, if goal nodes can be one of 4, 7, or 13, and four UAVs are generated at the same time (at a single sector), two UAVs will be assigned to travel to node 4, while one will travel to node 7 and one to node 13
  - random: the goal nodes assigned to UAVs are chosen at random; in other words, generated UAVs have an equal chance of being assigned to any one of the goal nodes

m.15 *destination* - which sectors can be assigned to UAVs as goal nodes by each generation sector

  - all: UAVs can be assigned to any node in the graph as a destination (except the node from which it is generated)
  - list: UAVs can only be assigned destinations nodes listed in the destination_points.csv file (except the node from which it is generated)

m.16 *generation* - which sectors UAVs generate from

  - all: UAVs are generated at all nodes in the graph
  - list: UAVs can only be generated at the nodes listed in the generation_points.csv file

m.17 *capacity* - how capacity is assigned to the links

  - flat: all links have the same capacity, equal to the value in (**c.3**)
  - list: links are assigned capacity individually; the capcity for the ith edge (in the edges.csv file) is given by the ith capacity in capacity.csv.

m.18 *state* - the state representation used by the agents (link agents only)

  - traffic: use 1 state for the link agents (simple state representation); this state is simply the number of UAVs currently on the link
  - incoming: use 2 states for the link agents (enriched state representation); first state is the number of UAVs currently on the link, the second state is the number of UAVs incoming to the link from other links (which may or may not be scaled, depending on (**m.20**) and **c.7**)

m.19 *tracking* - whether or not to record historical data about the domain; true or false (See Section 14.2)

m.20 *window_mode* - defines how the size of the window used by the link agents is determined (see Section 8.6 for more information)

  - variable: the size of the time window is defined by *window_size* (**c.7**)
  - cumululative: use a full window (with size equal to the number of *steps* (**t.3**)); cumulative refers to the fact that the link keeps information over all timesteps of each simulation

- **neuroevo:** contains parameters of the neuroevolutionary algorithm used to train the agents, including the structure of the neural network controllers used as policies for the agents

  n.1 *popsize* - the population size of each agent's population

  - **nn:** parameters for each neural network in the populations

    n.2 *gamma* - not currently used in code

    n.3 *eta* - step size used for backpropogation; not explicitly used for neuroevolution, but may be required because of class constructor stuff

    n.4 *mut_rate* - the percentage of weights to perturb during mutation; for example, 0.5 would mean 50% of the weights are perturbed each mutation

    n.5 *mut_std* - the standard deviation of values drawn from a normal distribution with zero mean, which are added to the weights during mutation; the default value is 1.0

    n.6 *layers* - how many hidden layers the networks have; default is 1

    n.7 *sigmoid_output* - whether or not the output values are scaled by the Sigmoid function; true or false

    n.8 *hidden* - integer that defines the number of hidden units

    n.9 *input* - integer that defines the number of inputs (Note: must be changed when changing *state* (**m.18**)!)

    n.10 *output* - integer that defines the number of outputs (the number of actions)

- **approximator:** not considered in this documentation
- **time:** contains parameters which directly influence how long the experiment will run

  t.1 *epochs* - how many epochs to train the agents for

  t.2 *trials* - not currently used in code

  t.3 *steps* - the number of timesteps of simulation of the airspace

  t.4 *runs* - how many statistical runs to perform

- **constants:**

  c.1 *sectors* - how many nodes are in the airspace graph (can be used to generate a graph with that many nodes)

  c.2 *alpha* - the value used to scale link agent actions by; recommended to be the length of the longest link as this ensures that costs are within a reasonable range (the maximum possible of predicted travel time plus agent action times *alpha* is around twice the length of the longest link)

  c.3 *capacity* - capacity of each link (only applies when **m.17** is set to "flat")

  c.4 *generation_probability* - chance that UAVs will be generated at a generation sector at each timestep; applies only when *traffic* (**m.5**) is set to "probabilistic"

  c.5 *generation_rate* - period of time before more UAVs are generated; applies only when *traffic* (**m.5**) is set to "deterministic"

  c.6 *num_generated* - how many UAVs to generate at each sector, each time they are to be generated

  c.7 *window_size* - the size of the window used by link agents; applies only when *window_mode* (**m.20**) is set to "variable" (also see Section 8.6 for more information on the time window)

  c.8 *xdim* - when generating an airspace, the maximum position in the x-direction a node can be placed

  c.9 *ydim* - when generating an airspace, the maximum position in the y-direction a node can be placed

13.1 Defining New Parameters

Of the existing fields in Section 13, the values listed are not final. For example, one could define "sectors_passed" as a value for *objective* (**m.10**), which would require the agents to try to reduce the number of sectors the UAVs pass when getting to their destinations (after implementing this new objective in the code). Note: This is a toy example. Don't try this at home... or at work, for that matter.

To define a new parameter in the configuration file, simply include a new field with the appropriate indentation and unique name (among its sibling fields). The value of the field can be any of the following types:

- **string:** This is simply a series of character values. Examples include "incoming", "list", or "canteloupe".
- **number:** This is a numeric entry. It can be an integer (such as 4) or a floating point number (such as 3.14).
- **boolean:** Can have values true or false.

To read the new fields from the configuration file, it is easiest to use the API provided by the yaml-cpp library. See the constructor in the `UTMAbstractDomain` class code for examples on loading parameter values of various types.

14 Folders

There are two primary folders created and used when running experiments. These are the Domain and Tracker directories.

14.1 Domain

This directory contains the files necessary to create the graph that represents the airspace. The Domain directory contains subdirectories corresponding to the number of sectors used in a graph. So, a graph with 11 sectors would be in the subdirectory "11_Sectors". Each of these subdirectories contains the following files, some of which are optional:

- **nodes.csv:** Contains a certain number of rows of data (equating to the number defined in *sectors* (**c.1**). The ith row defines the 2D position of the ith node.
- **connections.csv:** The adjacency matrix for the graph. This is a square matrix which defines the connections between the nodes. 1 indicates that two nodes are connected **in one direction**, and 0 means there is no connection between the two nodes (also only in one direction).
- **edges.csv:** Contains all the edges in the graph. Each row contains a pair of numbers, the first corresponding to the number of the source node, the second corresponding to the number of the target node (indexed at 0).
- (optional) **generation_points.csv:** Defines the generation nodes of the airspace graph. This is a list of the node numbers (starting at 0) which are designated generation nodes, each of which generate UAVs according to the UAV generation policy (see Section 13). This file is required only when *generation* (**m.16**) is set to "list".

- (optional) **destination_points.csv:** Defines the destination nodes of the airspace graph. This is a list of the node numbers (starting at 0) that are the designated goal nodes, any one of which may be assigned as a goal for the UAVs. This file is required only when *destination* (**m.15**) is set to "list".

- (optional) **capacity.csv:** Defines the capacity for each individual edge in edges.csv. This is required only when *capacity* (**m.17**) is set to "list".

When generating an airpsace (Section 5.4), the first three files mentioned are created.

14.2 Tracking

This directory keeps data about the domain for each timestep, epoch, and run. It is organized by the number of sectors just like in the Domain directory. Information captured may include position of the UAVs, each UAVs planned path, the UAVs that are waiting at each sector, and plenty of other things. Also captured is the time to travel each link and each link's capacity. This information is captured and written to disk by the `UTMTracker` class.

15 How to Change UAV Path Planning Algorithm

To get you on the right path (pun unintended, but it's staying), take a look at `planAbstractPath()` (Section 10.2) in the code. The code currently uses "astar" as the value for *search* (**m.8**), but this can be changed to another value of your choosing (really, anything). However, the new algorithm it must also be implemented! The A* algorithm in this code uses some help from the Boost C++ library, so maybe that's a place to start.