Logist Platform

Documentation

Contents

1 How to setup the LogistPlatform

1.1 Directory layout

1.1.1 Workspace folder layout

We suggest that you organize your workspace as follows:

```
workspace/              -- Eclipse workspace
  logist/               -- Logist Platform binaries
    logist.jar          -- Add this to your classpath
    lib/
      <6 binaries>.jar
  reactive/             -- Eclipse project (1st assignment)
    ...
  deliberative/         -- Eclipse project (2nd assignment)
    ...
  centralized/          -- Eclipse project (3rd assignment)
    ...
  auction/              -- Eclipse project (4th assignment)
    ...
```

Create a new workspace folder and extract all the zip files, logist.zip, reactive.zip, (more will follow later...) to this folder. This should give you the above layout.

1.1.2 Classpath

You will need to add logist.jar to your classpath. In eclipse, proceed as follows: Right-click on project → Build Path → Add External Archives

1.1.3 Project folder layout

The project folder that we give you for each project contains XML configuration files for the *LogistPlatform* as well as a Java template that implements the agent behavior for the exercise.

```
reactive/              -- Eclipse project
  bin/                 -- Compiled class files go here
  config/
    agents.xml         -- Agent definitions, edit
                          this file to add your agent
    reactive.xml       -- Main configuration file
    ...
  src/                 -- Source folder
    template/
      Template.java    -- Demo agent, test if you can run it
```

Before you start coding you should check whether you can run the starter code. First, check that the java file is compiled successfully and that its class file ends up in the bin/ folder. Then create a run configuration with the following settings:

Main class : `logist.LogistPlatform`
Arguments : `config/reactive.xml reactive-random`

From the command line, use:
`java -jar ../logist/logist.jar config/reactive.xml reactive-random`

This will start the simulation and you should see one agent moving around. If you want to run the program with two agents, use these arguments:

Arguments : `config/reactive.xml reactive-random reactive-random`

1.2 Configuration Files

When you are reading this document for the first time you can probably skip this part and focus on sections 1.1 and 3.1.

A configuration file consists of five parts which are explained in detail below. The syntax of the configuration file is as follows:

```
<configuration>
  <settings ..>  ...  </settings>
  <topology ..>  ...  </topology>
  <agents ..>    ...  </agents>
  <tasks ..>     ...  </tasks>
  <companies ..> ...  </companies>
</configuration>
```

1.2.1 <settings> xml tag

The settings determine the graphical appearance as well as the timeouts for agents. Sizes are given in pixels, colors in hexadecimal notation and timeouts in milliseconds. The *setup* and *plan* timeout are used for all agents where as the *bid* timeout only applies for auctioning agents.

```
<settings name="default">
  <!-- Sizes -->
    <set world-width="640" />
    <set world-height="480" />
    <set city-radius="8" />
    <set route-width="8" />
  <!-- Colors -->
    <set color-background="#ffffff" />
    <set color-foreground="#000000" />
    <set color-city="#00aaff" />
    <set color-city-name="#002233" />
    <set color-city-circumference="#556677" />
    <set color-task-text="#000000" />
    <set color-task-pickup="#ff0000" />
    <set color-task-deliver="#0000ff" />
    <set color-task-indicator="#000000" />
    <set color-route="#99aabb" />
  <!-- Timeouts -->
    <set timeout-setup="300000" />
    <set timeout-bid="300000" />
    <set timeout-plan="300000" />
  <!-- Flags -->
    <set flag-show-ui="true" />
</settings>
```

The settings can also be loaded from an external file using:

```
<settings import="config/settings_default.xml" />
```

1.2.2 <topology> xml tag

The topology tag describes the graph through which the vehicles are travelling. It consists of cities (nodes) and routes (edges). Cities have a x- and y-coordinate so they can be drawn on the screen. Distances between two cities are given in kilometers. The travel speed of vehicles is declared by the `vehicle` tag in section 1.2.5 below.

```
<topology name="france">
  <cities>
    <city x="324" y= "87" name="Paris"/>
    ...
  </cities>
  <routes>
    <route distance="591.2" from="Paris" to="Brest"/>
    ...
  </routes>
</topology>
```

It is possible to define the topology in a seperate file and then import it using:

```
<topology import="config/topology/france.xml" />
```

1.2.3 <agents> xml tag

The agents file is used to bind the agent's name to its class and its parameters. This is particurlarly useful for the last exercise when several agents are running on the same platform. The class path can either be a jar file or the output directory of the java compiler which is typically `bin/` in eclipse. The class name must be fully qualified, i.e. include the package name. User-defined parameters can be given as `name="value"` pairs. They are read using the *readProperty* function of the *Agent* class (see section 3.2.8).

Some exercises require that you implement certain parameters in order to test them more easily.

```
<agents>
  <agent name="reactive-.7">
    <set class-path="agents/reactive.jar"/>
    <set class-name="reactive.ReactiveExercise"/>
    <set discount-factor=".7"/>
  </agent>
  <agent name="deliberative-bfs">
    <set class-path="bin/"/>
    <set class-name="deliberative.DeliberativeExercise"/>
    <set algorithm="BFS"/>
  </agent>
  ...
</agents>
```

The agents can also be read from an external file using:

```
<agents import="config/agents.xml" />
```

In tournament mode, the path of the agents file may be overriden by a command line argument (see section 2.1).

1.2.4 <tasks> xml tag

The tasks tag defines the number of tasks and the probability distribution of pickup/delivery locations, rewards and weights. For the reactive exercise the number of tasks is ignored because there is a continous supply of tasks.

```
<tasks number="10" rngSeed="35904202421921524 24">
  <probability distribution="uniform" min="0.0" max="1.0" />
  <reward distribution="constant" policy="short-distances"
          min="1000" max="99999" />
  <weight distribution="constant" value="5" />
  <no-task distribution="uniform" min="0.2" max="0.4" />
</tasks>
```

The four tags describe the (relative) probability of a task, the reward of a task, the weight of a task and the (absolute) probability of having no task in a city. The probability of having no task is only relevant for the reactive exercise and is ignored otherwise.

The distributions can either be deterministic (constant) or created randomly (uniform). It is also possible to create distributions that are proportional to the distance between the pickup and the delivery location of the task. This allows to create distributions that are biased to favor certain tasks. Together, there are 4 combinations of these two concepts:

- A constant value or probability for each pair of cities.

  ```
  <... distribution="constant" value="5" />
  ```

- A uniformly drawn value or probability for each pair of cities.

  ```
  <... distribution="uniform" min="0.0" max="1.0" />
  ```

- A constant distribution for each pair of cities that is proportional to their distance. The policy can be either *long-, medium-* or *short-distances* indicating which tasks should receive a higher value or probability.

```
<... distribution="constant" policy="long-distances"
     min="100" max="1000" />
```

- A uniform distribution for each pair of cities that is proportional to their distance. The policy can be either *long-, medium-* or *short-distances* indicating which tasks should receive a higher value or probability.

```
<... distribution="uniform" policy="short-distances"
     min="0.0" max="1.0" />
```

1.2.5 <companies> xml tag

The companies tag describes the companies and their vehicles. At runtime the first agent will control the first company, the second agent controls the second company and so on. If there are fewer agents than companies then not all companies will be used. Each company owns a number of vehicles that the agent can use. Reactive and deliberative agents, however, will only ever use the first vehicle of the company.

The vehicle's speed is given in km/h. One second in the simulation corresponds to one hour in real time. The simulation speed can be increased in the user interface.

```
<companies>
  <company name="Company A">
    <vehicle name="Vehicle 1">
      <set color="#0000ff" />
      <set home="Lausanne" />
      <set speed="120" />
      <set capacity="30" />
      <set cost-per-km="5" />
    </vehicle>
    ...
  </company>
  ...
</companies>
```

2 How to run the LogistPlatform

2.1 Normal Execution

```
Usage:
  simulate game:
    [-o history.xml] [-a agents.xml] config.xml name1 [name2 ...]
```

The first argument is the configuration file. One or more agent names follow after that. The configuration file defines a number of companies with some vehicles each. At runtime, the first agent will control the vehicles of the first company, the second agent will control the second company and so on. The translation from agent name to agent class file is performed using the agents.xml file. You can also use the agents.xml file to pass your own parameters to your agent, such as the discount factor for the reactive exercise (see 1.2.3).

There are two optional arguments, which are mainly used by the tournament runner script.

-o changes the name and location of the history file, the output of the simulation.

-a forces that the specified agents file is used, instead of the agents tag in the configuration file.

2.2 Tournament support

```
Usage:
  create tournament:
    -new 'tournament name' 'agent directory'
  run tournament:
    -run 'tournament name' 'config (directory or file)'
  tournament scores:
    -score 'tournament name' [scores.txt]
```

2.2.1 Create a tournament

First, create a directory *agents* and a directory *tournament* in the project folder. Move the jar files of all participating agents to the *agents* directory. Suppose we want to call our tournament *MyEvent*. Run the *LogistPlatform* with the arguments `-new MyEvent agents`. This will create a subdirectory *MyEvent* in the *tournament* folder. Also, the script will find all classes that

implement the AuctionBehavior interface and generate a file `agents.xml` in the tournament folder *MyEvent*.

Before running the tournament, inspect the generated `agents.xml` and verify that no agent is missing or superfluous.

2.2.2 Run a tournament

The arguments for running the tournament *MyEvent* with configuration *config.xml* are `-run MyEvent config.xml`. The script will then start to run auction games between all pairs of players. For fairness, all games are run twice with players switching the roles of company A and company B. Depending on the chosen timeout, these simulations can be very time-consuming. There are two ways to speed things up. On multi-core machines we can launch multiple instances of the Runner on different CPUs. Also, if we are using a network file system (NFS) we can execute the Runner on different machines in a PC cluster.

The *LogistPlatform* uses a simple file-based synchronization scheme. Before we run a simulation we try to create the output using the java.io.File createNewFile() method. This will only succeed if the file did not exist before and we will only simulate the game if a new file was created. The operation is guaranteed to be atomic on local file systems, but usually not on NFS. If a race condition occurs we might simulate a game twice but this is very unlikely to happen.

2.2.3 Create tournament score board

When the tournament is finished, the tournament folder (*MyEvent* in our example) will contain a lot of history files. With the arguments `-score MyEvent scores.txt` we can invoke the score board tool that will create a grid showing how each agent performed against every other agent.

3 How to implement an agent

3.1 Behavior interfaces

The *LogistPlatform* supports 4 different kinds of agents: the reactive, the deliberative, the centralized and the auctioning agent. Depending on the type of the agent that you are writing you have to implement one of the following behavior interfaces.

3.1.1 Reactive interface

The behavior of a reactive agent is defined by a class that implements the `logist.behavior.Reactive` interface. A reactive agent has to have the following methods implemented:

```
void setup( Topology topology,
            TaskDistribution distribution,
            Agent agent )
```

The setup method is called exactly once, before the simulation starts and before any other method is called. The `agent` argument can be used to access important properties of the agent, see 3.2.8

```
Action act( Vehicle vehicle,
            Task availableTask )
```

This method is called every time the agent arrives in a new city and is not carrying a task. The agent can see at most one available task in the city and has to decide whether or not to accept the task. It is possible that there is no task in which case `availableTask` is `null`.

- If the agent decides to pick up the task, the platform will take over the control of the vehicle and deliver the task on the shortest path. The next time this method is called the vehicle will have dropped the task at its destination.
- If the agent decides to refuse the task, it chooses a neighboring city to move to. A refused task disappears and will not be available the next time the agent visits the city.

Note: If multiple tasks are available then the *LogistPlatform* randomly selects the task that is shown to the agent. If no task is available then the agent must return a move action.

3.1.2 Deliberative interface

The behavior of a deliberative agent is defined by a class that implements the `logist.behavior.Deliberative` interface. A deliberative agent has to have the following methods implemented:

```
void setup( Topology topology,
            TaskDistribution distribution,
            Agent agent )
```

Same as for the reactive agent.

```
Plan plan( Vehicle vehicle,
           TaskSet tasks )
```

This signal asks the agent to compute the transportation plan for its vehicle. All tasks in `tasks` must be picked up and delivered by the plan. In a single agent system, the agent can assume that the vehicle is carrying no tasks initially. In a multi agent system this might not case if the plan of an agent has just become invalid and needs to be recomputed (see below).

```
void planCancelled(TaskSet carriedTasks)
```

In a multi agent system the plan of an agent might get *stuck* when the agent tries to pick up a task that has already been picked up by another agent. In this case this method is called followed by an another *plan* computation (a call to the method above). This time the vehicle might be carrying some tasks initially (the `carriedTasks` argument) and these tasks have to be considered when the next plan is computed. You can also use the `getCurrentTasks()` method of the vehicle to obtain the set of tasks that the vehicle is holding (see section 3.2.9).

3.1.3 Centralized interface

The behavior of a centralized agent is defined by a class that implements the `logist.behavior.Centralized` interface. A centralized agent has to have the following methods implemented:

```
void setup( Topology topology,
            TaskDistribution distribution,
            Agent agent )
```

Same as for the reactive agent.

```
List<Plan> plan( List<Vehicle> vehicles,
                 TaskSet tasks )
```

This signal asks the agent to compute the joint plan for several vehicles. The agent has to create a plan for each vehicle such that together, all tasks in `tasks` are picked up and delivered. The plans of the vehicles are returned as a list, in the same order that the vehicles were given in the input. The agent can assume that no vehicle is carrying a task.

3.1.4 Auction interface

The behavior of a auctioning agent is defined by a class that implements the `logist.behavior.Auction` interface. A auctioning agent has to have the following methods implemented:

```
void setup( Topology topology,
            TaskDistribution distribution,
            Agent agent )
```

Same as for the reactive agent.

```
Long askPrice( Task task )
```

This signal asks the agent to offer a price for a task and it is sent for each task that is auctioned. The agent should return the amount of money it would like to receive for delivering that task. If the agent wins the auction, it is assigned the task, and it must deliver it in the final plan. The reward of the task will be set to the agent's price. It is possible to return `null` to reject the task unconditionally.

```
void auctionResult( Task lastTask,
                    int lastWinner,
                    Long[] lastOffers )
```

This signal informs the agent about the outcome of an auction. `lastWinner` is the id of the agent that won the task. The actual bids of all agents is given as an array `lastOffers` indexed by agent id. A `null` offer indicates that the agent did not participate in the auction.

```
List<Plan> plan( List<Vehicle> vehicles,
                 TaskSet tasks )
```

Same as for the centralized agent.

3.2 Logist API

3.2.1 Topology class (`logist.topology.Topology`)

```
Iterator<City> iterator()
```

Returns an iterator over all cities in the topology. The cities are returned in order of their id fields. You can use the shortcut `for (City city : topology) ...` to iterate over all cities in the topology.

`List<City> cities()`

The list of cities.

`int size()`

The number of cities.

3.2.2 City class (`logist.topology.Topology.City`)

`int id`

The unique id of the city in the range $[0, size)$. This allows the cities to be used as indices in array- or list-based data structures, for example in `distance[from.id][to.id]`.

`Iterator<City> iterator()`

Returns an iterator over all neighboring cities. You can use the shortcut `for (City neighbor : myCity) ...` to iterate over neighbors of this city.

`List<City> neighbors()`

The list of neighboring cities, i.e. the list of cities that share a route with this city.

`List<City> hasNeighbor( City city )`

Returns true if and only if `city` is a neighbor of this.

`double distanceTo( City to )`

Returns the distance in kilometers from this city to another city. If the two cities are not directly connected by a route then the length of the shortest path is returned.

`List<City> pathTo( City to )`

Returns the list of cities on the shortest path from this city to another city. The returned list does not include the first city on the path. For example if the shortest path from A to D is $A \to B \to C \to D$ then the list $[B, C, D]$ is returned.

3.2.3 Task class (`logist.task.Task`)

`int id`

The id of the city in the range $[0, numTasks)$. The task id is unique within a round.

`City pickupCity`

The pickup location

`City deliveryCity`

The delivery location

`long reward`

The reward for this task

`int weight`

The weight of this task

3.2.4 TaskSet class (`logist.task.TaskSet`)

A specialized implementation of Set<Task>. All elements in a TaskSet **must** come from the same task batch that is generated at the beginning of each simulation round. TaskSets are represented internally as bit vectors.

`int weightSum()`

The sum of weights of all tasks in the set.

`int rewardSum()`

The sum of rewards of all tasks in the set.

3.2.5 TaskDistribution class (`logist.task.TaskDistribution`)

```
double probability( City from,
                    City to )
```

Returns the probability that a task to `to` is available in city `from`. If `to` is `null` then the probability that there is no task available in city `from` is returned.

```
int reward( City from,
            City to )
```

Returns the (expected) reward for a task between two cities.

```
int weight( City from,
            City to )
```

Returns the (expected) weight for a task between two cities.

3.2.6 Action class (`logist.plan.Action`)

The class `Action` is abstract and has 3 subclasses. Actions are created as follows:

```
import logist.plan.Action.Move
new Move( City city )
```

Creates a move action.

```
import logist.plan.Action.Pickup
new Pickup( Task task )
```

Creates a pickup action.

```
import logist.plan.Action.Deliver
new Deliver( Task task )
```

Creates a delivery action.

3.2.7 Plan class (`logist.plan.Plan`)

```
new Plan( City initialCity )
```

Creates a new plan for a vehicle. The `initialCity` should be the current city of the vehicle.

```
void append( Action action )
```

Appends an action to the plan. There are also convenience methods for `appendMove`, `appendPickup` and `appendDelivery`.

```
double totalDistance()
```

Computes the total distance (in km) of this plan.

3.2.8 Agent interface (`logist.agent.Agent`)

```
int id()
```

This function returns the unique id of the agent. The id is always in the range `[0,numAgents)`.

```
List<Vehicle> vehicles()
```

Returns the list of vehicles controlled by the agent.

```
TaskSet getTasks()
```

Returns the set of tasks that the agent has accepted, but not yet picked up or delivered.

```
<T> T readProperty( String paramName,
                    Class<T> paramType,
                    T default_)
```

Reads a user defined property from the *agents.xml* configuration file. The property is converted to `paramType` before it is returned. If the conversion fails or if `paramType` is not set in the *agents.xml* file then the `default` value (if non-null) is returned. Section 1.2.3 describes how user properties can be set in the xml file.

3.2.9 Vehicle interface (`logist.simulation.Vehicle`)

`int id()`

This function returns the id of the vehicle which is unique for all vehicles of the same agent/company. The id in the range $[0, numVehicles)$, where numVehicles is the total number of vehicles of the agent.

`int capacity()`

The capacity of the vehicle.

`int costPerKm()`

The cost/km of the vehicle.

`City getCurrentCity()`

The current city of the vehicle.

`TaskSet getCurrentTasks()`

The tasks that are currently being transported by the vehicle. These tasks have been picked up but have not yet been delivered.