Laboratory: Programming Coordination with ASTRA

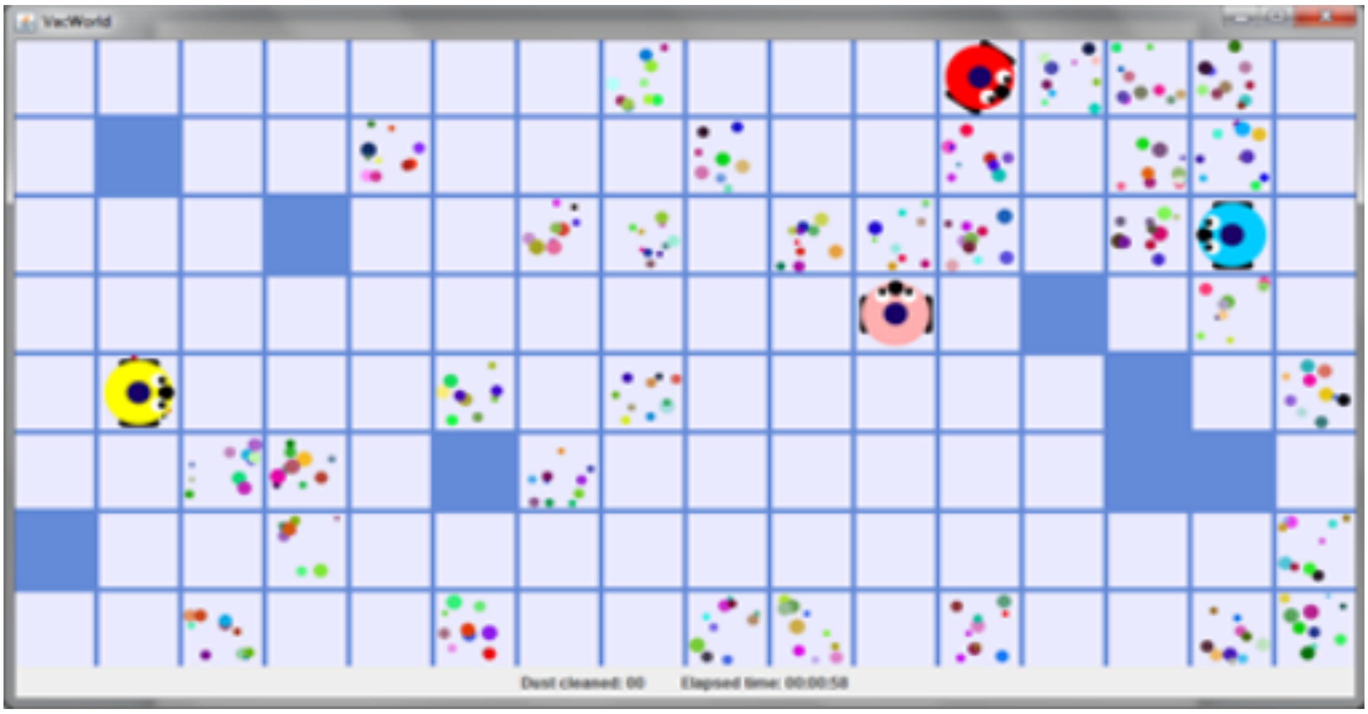
This lab is focused on implementing a coordination mechanism for VacWorld; a 2-dimensional grid world that is occupied by robotic entities known as VacBots. VacBots are designed to clean dust in VacWorld. Your mission, should you decide to accept it, is to program the VacBots to clean all the dust from VacWorld.

Submission Instructions

Create a new maven project for each part. The name of the project folder should be Part<num>. For example, the answer to Part 1 should be in a folder Part1. Each part folder should be in a folder called Lab3. At the end, please ZIP the Lab3 folder up and submit via Moodle.

The Problem: VacWorld

Source: https://github.com/eishub/vacuumworld



VacWorld is a 2-dimensional grid world. Each grid cell can be empty or contain either dust, an obstacle or a VacBot.

In its default configuration, VacWorld is inhabited by four VacBots: Henry, Decco, Harry and Lloyd. VacBots are able to move into cells that are either empty or dusty. Once in a dusty cell, VacBots are able to clean the dust (transform the cell from dusty to empty). Only one VacBot can inhabit a cell. Each VacBot has a light on top that they can turn on or off as needed.

Perception in VacWorld

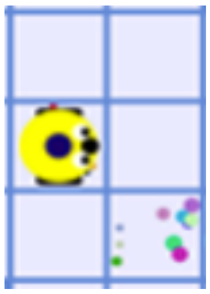
VacBots have a myopic view of the world: they can only see the cell they are residing in and the 5 cells in front or to the side of them. They also know their current location (with 0,0 being the top left of the grid) and their orientation (North, South, East, West).

The domain model consists of the following predicates:

- `location(long, long)`: position
- `direction(string)`: orientation
- `light(string)`: whether their light is on or off
- `square(string, string)`: Information about a cell. 1st parameter is the relative location and can be one of {`here`, `left`, `forwardLeft`, `forward`, `forwardRight` or `right`} and the 2nd parameter is the contents which can be one of {`obstacle`, `vac`, `dust` or `empty`}.
- `task(string)`: the current activity of the VacBot which is one of {`turn`, `move`, `clean` or `none`}

For example: Based on the image to the right, the VacBot would see:

```
square(here, empty)
square(left, empty)
square(forwardLeft, empty)
square(forward, empty)
square(forwardRight, dust)
square(right, empty)
```



Action in VacWorld

VacBots can perform 4 actions:

- `move(string)`: Move one square in a given direction defined either in absolute terms as {north, south, east or west} or in relative terms as {forward, left, right or back). If the chosen movement is not forward, the action includes turning to face the correct direction.
- `move(string, long)`: same as move, but allows more than one square movement to be specified (i.e. to move 3 squares east, you can use move(East, 3)).
- `light(string)`: Turn the VacBots light either on or off.
- `clean()`: clean the square inhabited by the VacBot.

Part 1: Getting Set Up (D grade)

The first task is to create some template agents to set up VacWorld. For this, you will be creating 2 agent classes:

- `VacBot.astra`: the agent program that will be used to control a VacBot.
- `Main.astra`: the main agent, responsible for setting up the environment and creating the VacBot agents

(a) Start by copying the TowerWorld project and updating the `pom.xml` to use the following maven properties:

```
o groupId: lab
o artifactId: vacworld
o eis.artifactid: vacuumworld
o eis.version: 1.3.0
```

(b) Next, we will edit the `Main.astra` program. Start by stripping it back to include only the module declarations and the `!main()` rule.

(c) Modify the `ei.launch(…)` action to refer to "dependency/vacuumworld-1.3.0.jar" and delete everything after `ei.start()`.

(d) Run the program – it should create the GUI but it will also throw an exception. This is because the default environment parameters provided do not match the expected parameters (the regeneration parameter has now been changed to generation). If you want to get rid of this, modify the `ei.init()` action as follows:

```
e.init([generation("no"), level("4")]);
```

(e) Run the program again to verify that the warning is gone. Also note that the layout of VacWorld has changed. By default a random world is created each time VacWorld is run.

(f) Create a new agent program called `VacBot.astra` with the following content:

```
agent VacBot {
 module EIS ei;
 module Console C;

 rule +!main([string vacbot]) {
   ei.join("hw");
   ei.link(vacbot);
   C.println("VacBot activated");
 }
}
```

(g) Modify the `Main.astra` program to create the initial set of VacBots by inserting the following code **before** the `ei.start()` action (and run the program again):

```
list entities = ei.freeEntities();
forall(string entity : entities) {
  S.createAgent(entity, "VacBot");
  S.setMainGoal(entity, [entity]);
}
```

Part 2: Reactive Cleaners (C grade)

All the remaining tasks will focus on modifying the VacBot.astra file to create increasingly complex behaviours. The first challenge is to create what are known as reactive cleaners. These are VacBots that clean dust when they see it or move in a prescribed route when they don't.

(a) We start by creating a rule to capture the goal of deciding on the next task of the VacBot. To do this, we will introduce a goal `!task(string)` that is the mirror of the task(string) perception provided by EIS. Initially, we will focus on movement, so this goal will be adopted whenever the environment changes.

(b) We need to use an event to trigger the decision making behaviour. To do this, we can use the `task(…)` belief. If the agent adopts the belief `task("none")` then it has nothing to do (either because it was just created or because it finished its last task. We can model this with the following deliberation rule:

```
rule +$ei.event(task("none")) { !task(string action); }
```

(c) Now we need to start implementing decisions. We can begin with the simplest decision – to clean the square we are on (if it is dirty). This can be captured by the rule:

```
rule +!task(string action) : ei.square("here", "dust"){
  ei.clean();
  action = "clean";
}
```

The assignment of a value to the action variable is just for completeness and omitting it will not affect the behaviour unless you have a plan that does something after the `!task(…)` goal returns.

(d) Next, we need to develop a set of rules that will respond to the existence of dust on squares that are around the VacBot (assuming the square it is on is no longer dirty). For this we should focus on only the squares that are directly accessible by the agent. For example, if the square in front of the agent is dirty, the agent should move there (where it will clean the square based on the rule defined in (c)). We can capture this with the rule:

```
rule +!task(string action) : ei.square("forward", "dust"){
  ei.move("forward");
  action = "move";
}
```

Write additional rules to capture cleaning the square to the left or the right if dirty. The order of preference should be forward, left, right.

(e) The penultimate part of the reactive behaviour is what to do if there is no dirt to be seen. We can start with a default behaviour of moving forwards because this will bring any dirt in forwardLeft or forwardRight to the left or right of the VacBot triggering the cleaning rules from part (d). After this we need to make decisions about whether to prioritise turning left or turning right. Finally, the best way of doing this is to introduce the concept of a free square denoted `free(string)`. You can infer that a square is free if the square is either empty or dusty. This concept can be implemented as an inference rule or directly encoded in the context. The format of the rule is the same as for part (d) but with a different context.

(f) Spend some time watching the output. See what happens if you make the level parameter of `ei.init(…)` 1 instead of 4. You should start to see that occasionally, the task selection does not work correctly. The reason for this is the way that EIS updates the perceptions of the agents. When it finishes cleaning the dust, it does not return the full state, but only a partial state. This is probably a bug in VacWorld, but there is a solution.

Change the event that triggers the task selection from:

```
+$ei.event(task("none"))
```

to:

```
+$ei.event(location(long X, long y))
```

This means that the agent only selects a new task when it moves. The implication of this is that the agent will stop the first time it cleans a square. To get around this, the agent needs to repost the `!task(…)` goal in the rule when it cleans:

```
rule +!task(string action) : ei.square("here", "dust"){
  ei.clean();
  !task(action);
}
```

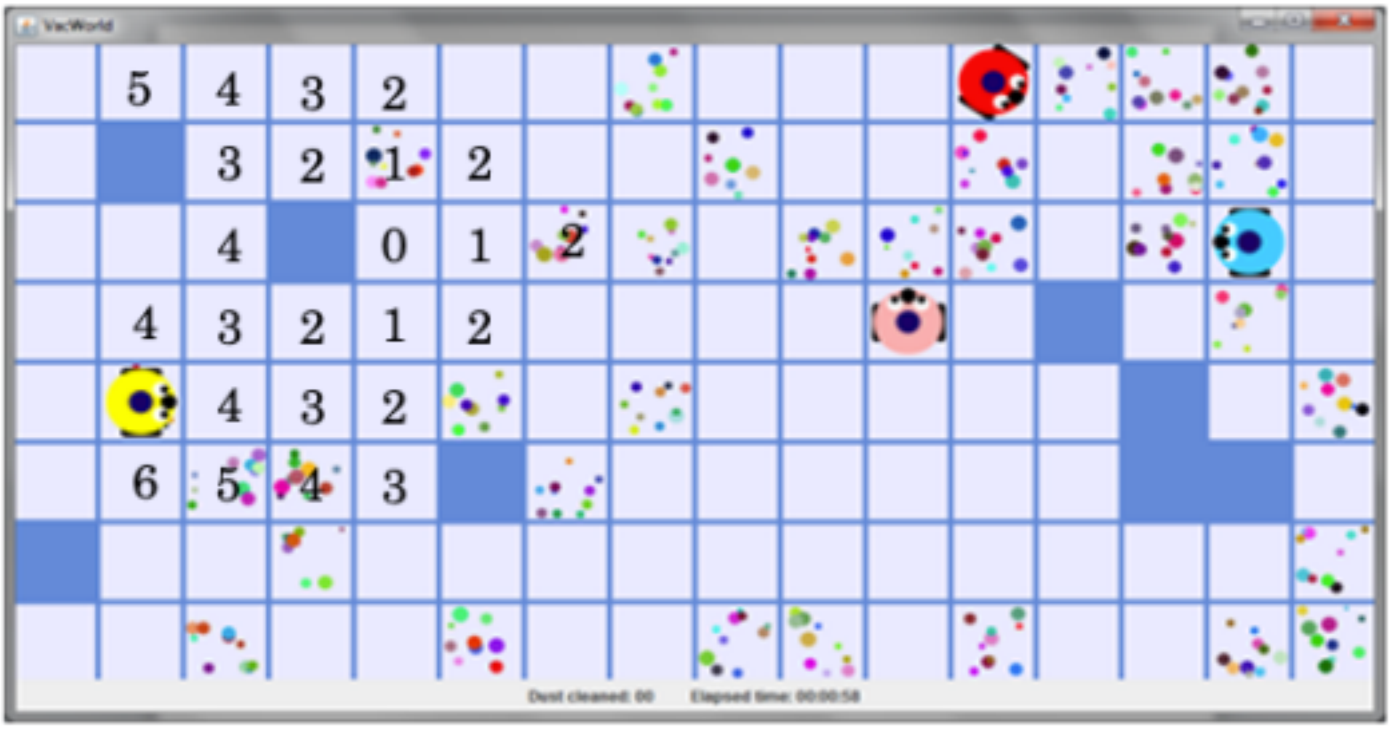
This will make the agent choose a second task after it has finished cleaning the square it is on. Once you modify the code to work this way, it should be more stable.

Part 3: Becoming Proactive (B grade)

This focuses on route planning as a way of being able to move to a specified location. Start by downloading the **Route Planning Source Code**. This code should be placed in the `src/main/java` folder of your project (`Routing.java` and `GradientMap.java` should be in the `src/main/java` folder). Spend a bit of time familiarising yourself with the code. `GradientMap.java` is an implementation of the shortest path algorithm (based on a flood fill approach) and `Routing.java` is an ASTRA module.

The basic idea behind a flood fill algorithm is to fill in a grid with values that represent the distance to some target location. The destination cell is valued at 0 or 1 (0 in the diagram, 1 in the code). Valid movements are used to assign values to surrounding cells, with the algorithm stopping when there are no more cells to assign a value to. The route is then selected based on some initial (source) cell by repeatedly identifying what move results in the agent going to a cell with the lowest value (this is gradient descent).

In the example below, the yellow VacBot goes north, then east, then east, then east, and then north to reach its destination (the cell with value 0).



In order to use this approach, you also need to record any obstacles you find as you explore the environment. Obstacles are given large scores so that they can never be selected in route finding. The approach also requires that you have an estimate of the size of the grid.

Steps to completing the part:

(a) Add the Routing module to the VacBot.astra file.

(b) Implement a rule based on the `!update()` goal that updates the set of obstacles as they are discovered. Two methods have been created in the Routing module to support this. A useful snippet of code is given below:

```
foreach(ei.square(string location, "obstacle")
          & ~routing.obstacle(X, Y, D, location)) {
  routing.recordObstacle(X, Y, D, location);
}
```

The additional variables include the X and Y coordinates of the agents' current location and the direction (D) that the agent is facing.

(c) The second issue to be resolved is the size of the grid. This can be derived from the location of the agent plus one in both the X and Y coordinates. The plus one reflects the fact that VacBots can sense the border (which is a square at the edge of the grid. The Routing module provides 2 methods: `maxX()` and `maxY()` that can be used as TERMS (i.e. `int X = routing.maxX()`). In addition, an `updateBoundary(long,long)` method is provided that can be used to update the values of maxX and maxY. This can also be done as part of the rule that handles the `!update()` goal.

(d) Now that we have a model of the environment, we can start to use the route finding part of the system. To do that, simply call:

```
list L = routing.routeTo(sx, sy, tx, ty);
```

Here sx and sy are your current location and tx and ty are the target location. Pick somewhere that is definitely on the map and see if it outputs the route to it.

Remember that you can use: `C.println("L=" + L);` to display the route found, and: `routing.displayMap();` to see the current map that the agent has of the environment.

Ultimately, you have to decide how to trigger this code. It could be after a fixed number of moves or you could remove the reactive movement rules and insert a rule that is associated with a `!task(…)` goal event and has a context that there is no route. The rule should then have a plan that creates a route to a specific destination and adopts a belief `route(L)` which will be used to trigger the route following behaviour in the final step of this part.

(e) To actually follow the route, you need to write some code that is similar to the way you built towers in the tower environment. You need to use the splitter `[funct move | list T]` to extract the next move from the list. You then need to use a goal that contains the functional term to make the robot move:

```
rule +!task(string action) : route([funct move]) {
  -route([move]);
  !go(move, action);
}

rule +!task(string action) : route([funct move| list L]) {
  -+route(L);
  !go(move, action);
}

rule +!go(move(string direction), string action) {
  ei.move(direction);
  action = direction;
}
```

Note: You may detect some scenarios where the VacBot fails (e.g. there is an obstacle in the planned path. This can happen because the route was created before the VacBot discovered the obstacle. If you can modify the code to detect this situation and get the agent to replan its route successfully, you are guaranteed a B+.

Part 4: Getting Collaborative (A grade)

The penultimate task is to implement a coordination mechanism based on the Hierarchy organisational pattern. Designate one VacBot to be the Leader. Create a new `Leader.astra` file. And modify the basic VacBot functionality to explore only. When it detects a piece of dust, the VacBot should store the location in a list.

Create a second file called `Follower.astra`. Followers should request a task from the Leader. A task is basically the location of a dirty cell. The follower should navigate to that cell and clean it. It should then ask the leader for the next task.

If you can get this to work once, you get an A-. If it works consistently, you get an A grade.

Part 5: The Full Monty (A+ grade)

If you can extend the code to ensure that it cleans the entire grid every time (for all levels), you will get an A+.