Aston University
Birmingham B4 7ET
http://vle.aston.ac.uk/

Multiagent Systems: Practical Task on Reactive Architectures

Maria Chli

Module CS3340 October 19, 2020

1 Introduction

This document contains a specification of the software and other documentation that form a practical task on reactive agent architectures for the module CS3340: Multiagent Systems. The solution to the task is not directly assessed, but will be discussed in tutorial sessions and will be assessed in the module's exam.

Reactive agents and the subsumption architecture

You may find it helpful to study lecture 9 and chapter 5 of the Wooldridge book (both available on BlackBoard) to refresh your knowledge of reactive agents and the subsumption architecture. Try make sure that both your solution and the accompanying discussion meet the **learning outcomes** specified in lecture 9 with respect to reactive architectures.

In subsumption architecture the control loop of the agent consists of a set of behaviours implemented as rules of the form $situation \rightarrow action$. In Java you could implement such rules as follows:

```
if (situation){
  carry out action
  return;
}
```

2 Problem Description

The task is to implement a variant specification of Steel's Mars [1990] autonomous explorer vehicle, using a *subsumption architecture*, as covered in the lectures. Steel's scenario is as follows:

> "The objective is to explore a distant planet, more concretely, to collect samples of a particular type of precious rock. The location of the rock samples is not known in advance, but they are typically clustered in certain spots. A number of autonomous vehicles are available that can drive around the planet collecting samples and later reenter a mother ship spacecraft to go back to Earth. There is no detailed map of the planet available, although it is known that the

terrain is full of obstacles (hills, valleys, etc.) which prevent the vehicles from exchanging any communication." [from "An Introduction to MultiAgent Systems" by M. Wooldridge, John Wiley & Sons, 2009.]

You are given the code for simulating Mars, the simulation controller class, a graphical user interface for setting simulation parameters and viewing the execution of the simulation as well as other auxiliary classes. The terrain on Mars is represented as a two dimensional toroidal grid (also referred to as field). At any time step, each entity occupies a location on the terrain of Mars. A single mother ship has landed on a random location on Mars and has dispatched a set of vehicles. Vehicles roam the terrain, avoiding obstacles, collecting rock samples and returning them to the mothership. A vehicle can carry **at most** one rock sample at a time. **Your task is to complete the implementation for the autonomous Mars vehicles.** If you deem it necessary you may alter the given classes, provided that you clearly document your changes.

2.1 Populating the terrain

At the start of the simulation Mars' terrain is populated with entities according to the settings given through the graphical user interface. The creation probability for a specific entity signifies the likelihood of an instance of that entity to occupy a particular location in the field. Rocks are created in clusters depending on the number of clusters and the dispersion (standard deviation) around a randomly chosen center on the terrain. There is only one mother ship at a random location on the terrain. The code for populating the terrain is given in class Simulator.

2.2 Vehicle behaviour

Only the vehicles move on the terrain. Rocks, obstacles and the mother ship are static. You are asked to implement two modes of behaviour for the Vehicles: simple and collaborative. In a simulation where the vehicles behave according to the simple mode the vehicles roam the terrain trying to accomplish their goal independently, without communicating with their peers. On the other hand, when in collaborative mode, the vehicles indirectly communicate with each other by dropping crumbs when returning to the mother ship carrying a sample. The precise rules you should implement in the agent control method (`act`) are given below. In both modes of operation the mother ship emits a radio signal that the vehicles pick up when trying to move towards the mother ship. The signal weakens as distance from the source increases, so if the agent wants to approach the mother ship it has to travel up the signal gradient.

2.2.1 The simple vehicle

The behaviour of the simple vehicle is represented by the following set of rules.

`if` carrying a sample `and` at the base `then` drop sample (1)
`if` carrying a sample `and not` at the base `then` travel up gradient (2)
`if` detect a sample `then` pick sample (3)
`if true then` move randomly (4)

These behaviours are arranged into the following subsumption hierarchy:

$$(1) \prec (2) \prec (3) \prec (4).$$

2.2.2 The collaborative vehicle

An additional set of rules is utilised by the collaborative vehicle. These are given below:

`if` carrying a sample `and not` at the base `then` drop two crumbs `and` travel up gradient (5)
`if` sense crumbs `then` pick up one crumb `and` travel down gradient (6)

The subsumption hierarchy for the collaborative vehicle is as follows:

$$(1) \prec (5) \prec (3) \prec (6) \prec (4).$$

3 Given code

A set of classes is given, in order for you to concentrate on developing the part of the simulation that concerns the modelling of the autonomous vehicles. Most of these classes you do not need to alter in any way. Make sure you read them and understand their methods, as you will need to call them. The purpose of each of the given classes is summarised below in alphabetical order:

ClusterGenerator provides code for placing the rocks on the terrain in clusters.

Counter stores a current count for one type of entity to assist with the counting.

Entity acts a super class for all objects that can be placed on the field (obstacles, rocks, mothership and vehicles).

Field represents a 2D field, the Mars terrain. It is composed of a fixed number of locations, arranged in a grid. The topology of the grid is torus-shaped. This means that it is like a chessboard but when a piece goes beyond the bottom row it reappears from the first row and vice versa. Similarly when it goes beyond the rightmost column it reappears from the leftmost column and vice versa. (The code for this is already given - you only need to understand it.) At most one entity may occupy a single location in the field. Each field location can hold a reference to an entity, or it can be empty. The class Field contains methods you can use to query adjacencies as well as information about the gradient field and the number of crumbs on each location.

FieldStats provides counts of the numbers of different entities in the field to the visualisation.

GUIMain prepares and displays a window for obtaining user input parameters, with functionality to run the simulation step-by-step or for longer periods. It feeds the input parameters to the fields of class ModelConstants.

LabelledCheckBox and LabelledTextArea are auxiliary classes for the drawing of the graphical user interface.

Location represents a two dimensional position within the field grid. Its position is determined by a row and a column value.

ModelConstants contains all the constants used throughout the simulation. Each these fields has a default value which is used when the simulation is run without a GUI (from `main` in class Simulator). When the simulation is run from the GUI (from `main` in class GUIMain) the values of these fields are set according to the user input on the GUI window.

Mothership class represents the mother ship entity and contains code to simulate the emission of the radio signal.

Obstacle class represents the obstacle entities.

RandomGenerator provides functionality to assist us with random number generation.

Rock class represents the rock sample entities.

Simulator is responsible for creating the initial state of the simulation (by populating the terrain with entities) and then controlling and executing it (by letting all vehicles act at each step of the simulation). We often refer to it as the controller class of the simulation.

Simulator View provides a graphical visualisation of the state of the field terrain. It draws the entities in the colours specified in class ModelConstants.

Vehicle models the autonomous Mars vehicle entities. It has a method `act` which either calls either the method which lets the vehicle act in the simple mode or the method that allows the vehicle to act in a collaborative mode. Your task is to complete these two methods.

4 Experiments

Once you have completed and tested the Vehicle implementation, run experiments to compare the performance of the simulation with regards to the time taken to collect all the rocks when the vehicles are operating in (i) simple and (ii) collaborative mode. Show how the performance is affected by having different numbers of vehicles in the terrain. Plot your results on a graph that contains two lines (one for simple and one for collaborative mode) where the x-axis is the vehicle creation probability and the y-axis is the number of steps to collect all the rocks.

You need to keep all input parameters constant and vary the vehicle creation probability. Each data point in your graph should be the average of the number of steps taken to collect all the rocks on the Mars terrain over a number of random generator seeds. For the purposes of this exercise repeat each run for **at least three** different seeds.

Note in a table the values of all the constants in the simulation. Write a brief (less than half a page) discussion of the results and their implications.

5 Deliverables

You need to complete the implementation of class `Vehicle`, making changes to given classes as needed, along with a report, the contents of which are outlined below.

1. Implementation
   (a) Implement method `actSimple` using the subsumption architecture outlined in section 2.2.1.

(b) Implement method `actCollaborative` using the subsumption architecture outlined in section 2.2.2.

(c) The *travel down gradient* action can be modified to make the vehicle's search for rock samples more efficient by having the vehicle sense *crumbs*. Implement the optimised version for this action.

(d) Class `Field` contains a method `reduceCrumbs`. Use this method appropriately to optimise the collection of the rocks.

You may use any compiler or development environment you like to develop the program.

2. The report should contain the following:

(a) A brief description of each method implemented in class `Vehicle`.

(b) An explanation of how parts (c) and (d) of the implementation task should work (even if you have not implemented them).

(c) The results of the experiment detailed in section 4 and brief analysis.

(d) A discussion of what you have learned from carrying out the implementation, with respect to the type of solution and architecture used (in comparison to other possible agent architectures).

If you wish, consider *further optimisation* of the solution (which is also clearly documented in the report) and further experiments. As part of the optimisation task you may create alternative versions of the `act` methods and/or modify the given code. In any case, please make sure that the methods `actSimple` and `actCollaborative` function exactly according to the specification. Any other (optimised) `act` method should be given a different name.

6 Feedback and support

Monday 5-6pm slots will be used as support sessions to assist you with completing the tasks and discuss feedback for your solutions on weeks 5 and 6. Please bring along all your questions to these sessions.