Programming Assignment 2: Rovers on Mars

comp440 Fall 2013

due 20 September 2013 at 8 pm on Owlspace

1 Background

You work for NASA, on the project team that is designing the successor to the Mars Pathfinder mission: the Mars Surveyor Mission. Your team will be landing a number of robotic surface rovers to survey the planet's surface. Your team has already selected a region of the Martian surface to survey. The rovers have relatively short range sensors, and a limited amount of power. The goal is to have them physically traverse as much of the region as possible. Your job is to design agents that will direct the rovers effectively within the given constraints. You will explore two radically different designs for the rover agents. The designs are distinguished by the amount and type of information which the sensors of these agents provide. For the first design, you will build **ant agents** with no state and limited percepts. Ants agents lay and update pheremone trails – the first problem in Homework 1 describes how they work. For the second design, you will build a **planning agent** that uses a version of the classical A* search algorithm, called Dynamic A*, which is designed to work in unknown or partially known environments. The planning agent uses considerably more information about the problem including the current location of the rover, the locations of the other rovers, as well as the explored terrain.

2 The simulation

Before we delve into the details of the assignment, it is important to see these rovers in action in the Martian environment. All the code for this problem is available as a zip archive (pa2.zip) on Owlspace under the Resources tab in the folder `week2`. When you unzip `pa2.zip` you will see the files and folders in Table 1.

To launch the simulator, get to the directory containing the files and run the following on the command line.

```
>> ./run.sh --gui
```

Name	Edit?	Read?	Description
README.txt	No	Yes	guide to the assignment including links to documentation
README_JAVA.txt	No	Yes	guide to setting up the code base.
assignment.pdf	No	Yes	This document
bin/	No	No	folder containing class files
boards/	No	No	folder containing board topologies
compile.sh	Yes	Yes	script for compiling rover code
doc/	No	Yes	javadocs for the project
java.pdf	No	Yes	Some useful Java hints
rovers.jar	No	No	jar file with simulation binaries (except for rover binaries)
run.sh	No	No	script for running the simulation
src/	No	No	folder containing src files. Only need to read and modify files below.
src/student/RandomRover.java	No	Yes	code for a rover than chooses actions randomly
src/student/AntRover.java	Yes	Yes	code for your ant rover
src/student/SmartRover.java	Yes	Yes	code for your search-based planning rover

Table 1: Contents of pa2.zip

You should see a simulator window pop up. Go to the `Simulation` menu and pick a board from the `boards/` folder. Pick the random rover under `Select a Rover`. Now click the run icon at the bottom of the window and watch the random rovers attempt to traverse the terrain. Clearly random rovers are not very effective for terrain coverage. Your ant and planning agents will perform much better. The `View` menu allows you to visualize the pheremone trails of the ant agent as well as the current state of the rovers' knoweldge of the terrain (`view fog` choice in the `View` menu).

You will need JDK 1.7 for the simulator to work. If you have an earlier version of the JDK, you will get the "Unsupported major.minor version 51.0" error when you try to run the GUI. At that point, you need to upgrade to JDK 1.7 for your computing platform. To help you get started in a Java development environment, we have set up three resources.

- `README_JAVA.txt` is a text file that goes through the steps of setting up the code base in Eclipse.
- `http://www.youtube.com/watch?v=QEt73vKcVQ8` – a Youtube video made by Ian Arnold to demonstrate how to set up Eclipse to use the code base for the comp440 rovers.
- `http://www.youtube.com/watch?v=uTLUrf9bQiY` – a Youtube video made by Ian Arnold on launching the GUI simluator and making simple rovers.

You can also run the simulation in CLI mode (without graphics) to generate summary statistics. The GUI is useful for developing and testing your agents, but you may find the command line easier when collecting performance information. To run the CLI simulator, on the command line type,

```
>> ./run.sh --cli
```

The CLI simulator accepts the following arguments:

- `-r`, `--rover` specifies the name of the rover to use during the simulation. This name needs to be a name of a rover available from the `student.Rovers` (`src/student/Rovers.java`) class.
- `-b`, `--board` specifies the name (relative or absolute path) of a board file to load and use for the simulation. A number of boards can be found in the `boards/` directory.
- `-n`, `--number-of-trials` specifies the number of trials to attempt.

3 The environment

The Mars environment is modeled as a two dimensional 50×50 Manhattan grid. Some squares contain obstacles that the rovers cannot move over, and other squares are open space. At the start of the mission, the rovers do not know the locations of the obstacles in the environment. They find obstacles by exploration. There are many different terrains for the rovers to explore; you need to design a **single** exploration strategy that works well in all of them. The rovers are evaluated as a team. At the end of a 1000-time-step simulation, the score given to the team is the number of open squares in the environment that have been visited by a rover. Note that there is no advantage in having more than one rover explore an open square, because the score is based on whether or not a navigable square has been visited at least once by any rover. Your goal is to design each rover's policy so as to maximize this score. Since there is more than rover in the environment, part of strategy design is how to divide up the exploration work between the rovers effectively.

4 Sensors and Actuators

4.1 Actuators for ant and planning agents

The ant and planning agent rovers have five actions available to them: four movement actions and one idle action. The four movement actions are *north*, *south*, *east*, and *west* and they move the rover one square north, south, east, and west, respectively, from its current location, when there is no obstacle in the way. If there is an obstacle in the way, the move action has no effect. The idle action does nothing.

In addition to movement actions, the ant agent can set the pheromone level of the square that it currently occupies. The planning agent only has the five movement actions at its disposal.

4.2 Sensors for ant agents

The ant rovers get three percepts.

- The first percept is the elapsed mission time.
- The second percept is a vector of length four with the pheromone levels of the Manhattan neighbors of the cell that the rover is on. The value Double.POSITIVE_INFINITY is the pheromone level associated with a square to which the rover cannot move to (wall or obstacle).
- The third percept is the pheromone level at the rover's current location.

Ant rovers have no information about the topology of the environment, the location of other rovers, or their own location in the environment.

4.3 Sensors for planning agents

The search-based planning agent rovers get four percepts at each time step.

- The first is a bump sensor, which turns on when the rover attempts to move into a square with an obstacle in it.
- The second percept is the elapsed mission time.
- The third percept is a two dimensional grid of locations in the world. Locations on the grid are marked *unexplored* if no rover has visited them, *clear* if a rover has visited them, and *impassable* if a rover has tried and failed to occupy that square.
- The last percept is the current location of all the rovers in the environment.

5 Question 1: Implementing the ant agent (20 points)

- **Standard Ant Agent (10 points):** The standard ant agent at a square s moves to the Manhattan neighboring square with the lowest pherenome level (ties broken randomly). It updates the pheremone level $u(s)$ at square s to be 1 plus the pheremone level of the square that it moves to. The update rule can be written as

$$u(s) \quad = 1 + u(\text{succ}(s, a))$$

where s is the current location of the agent, $a = \text{argmin}_{a \in A}(u(\text{succ}(s, a)))$ and $succ(s, a)$ is the successor of s with action a. You will implement this rule in `AntRover.java`.

What are the performance scores for the eight terrains Terrain1, Terrain2, Terrain3, Terrain4, Terrain5, Terrain6, Terrain7, and Terrain8? Generate a table with means and standard deviations of the score over at least 10 runs of this algorithm, for each terrain. Your table should have eight rows and three columns (terrain name, mean, standard deviation).

- **Custom Ant Agent (10 points):** Design and implement your own update rule for the pheromones, and report the scores for the same eight terrains, averaged over 10 runs over each terrain. Be sure to describe your update rule and your reasoning for it in your writeup.

5.1 Implementation hints

You need only to implement the method `nextAction(AntPercepts p)` in `src/student/AntRover.java`. The simulation will automatically call the method when the rover type is set appropriately. The `AntPercepts` class contains the sensors for Ant rover. To access the percepts, you should call the following methods on the percepts object `p` passed to you in the `nextAction(Antpercepts p)` method.

- `missionTime()` returns the elapsed mission time.
- `coinToss()` returns a random boolean.
- `pheromoneLevels()` returns a `Map` with the pheromone levels of the agent's Manhattan neighbors. You can access elements of the `Map` using the possible values of the `Action enum`. For example:

```
nextAction(AntPercepts percepts) {
    Map<Action, Double> pheromones = percepts.pheromoneLevels();

    // You can access a single value via Map.get(...)
    double northValue = pheromones.get(Action.NORTH);

    // You can also loop through all of the entries.
    for (Map.Entry<Action, Double> entry : pheromones.entrySet()) {
        Action direction = entry.getKey();
        double pheromoneValue = entry.getValue();

        // use those values...
    }
}
```

Any square to which the rover cannot move (because it is off the map or blocked by an obstacle) will have a pheromone level of `Double.POSITIVE_INFINITY`. All pheromone levels start at zero.

- `setPheromoneLevel(double level)` sets the pheromone level at the agent's current location.
- `currentPheromoneLevel()` returns the pheromone level at the agent's current location.

6 Question 2: Implementing A* (20 points)

Before you work on a strategy for coordinating the planning agent rovers, it is important to have a robust A* implementation. We separate the A* implementation from the planning agent implementation for a number of reasons.

- It is easier for *you* to test and debug the planning agents (called SmartRovers). When your planning agents do not cover terrain well it is helpful to know whether the fault lies in a poor A* implementation or a poor strategy for coordinating the rovers. By separating concerns and working on these two aspects sequentially, we make it easier for you to build high-quality planning agents.
- It is easier for *us* to grade your work and provide you meaningful feedback.

You will now implement the A* graph search algorithm in an application-agnostic fashion. That is, your A* implementation should not be aware of this specific problem context – agents, percepts, terrain, etc. Instead, it should work with any weighted undirected graph and any distance estimator function.

- **Build A* (15 points)** Implement the `student.provided.IUndirectedGraph` interface. The interface specifies methods for representing a weighted undirected graph and path finding in the graph using the A* algorithm. We have also provided the abstract class `student.provided.AUndirectedGraph`. Feel free to alter the `AUndirectedGraph` class, or to use it as a base when deveoping your own `IUndirectedGraph` implementation.
- **Test A* (5 points)** Use the `instructor.AStarTest` class to test your A* implementation. This class is included in the provided rovers.jar JAR archive, and can be imported as long as the JAR is included in your classpath (use the "-classpath rovers.jar" command line flag; see compile.sh and run.sh). The specific method that will test your implementation is `instructor.AStarTest.testAStar(...)`. The included JavaDoc has information about its parameters and return value. You may also use the `instructor.AStarTest.getTestMaze(...)` to generate the same maze graph that is used in `instructor.AStarTest.testAStar(...)`. This class is provided in

the `rovers.jar` distributed with the assignment. Report the results of your testing in your writeup both on the provided test case and any new test cases you generate with getTestMaze.

7 Question 3: Designing search-based planning agents using dynamic A* (60 points)

- **Designing a planning agent: (10 points)** : Describe a search-based agent for the terrain coverage problem. Your agent should make use of the percepts described in Section 4.3 on page 4. Your agent should also use the dynamic A* algorithm described in Section 8 to find paths in partially known terrains. Since there is no direct communication between rovers, you need to have the rovers implicitly coordinate their choice of unexplored locations to visit in order to maximize the area explored within the given time limit. One idea is to have each rover choose squares to explore that are furthest from the other rovers' current locations. You should devise your own implicit coordination strategy - in fact, this is the real meat of the assignment and the choice you make will here have a big impact on the overall performance of the rover team. Use the GUI mode to visualize the performance of your rovers and the CLI mode to collect statistics later.

  Be sure to explain your coordination strategy in your report, as well as any other detail or techniques that you use to improve performance. We should not need to consult your code for an understanding of your algorithm. Put this description in your report (pdf) to be submitted with the code for this problem.

- **Agent implementation and evaluation (30 points):** Implement the search agent you designed above. Place your implementation in `SmartRover.java` in `src/student`.

- **Agent evaluation (10 points):** Report the scores (means, standard deviations) of your agent for the terrains shown below, averaged over 10 runs.

  Here is a table of the scores considered perfect for each terrain:

  You can expect to earn points for your search agent that are roughly commensurate with its average performance over all of the provided terrains; that is, if it discovers an average of 75% of the passable area, you can expect to earn about 75% of the points available for its design and implementation.

- **Analysis: (10 points)** Compare the performance of your search-based agent against that of the ant agent you have implemented in the previous problem. To make the comparison fair, we need to charge the state-based agent more for the extra percepts that it receives, and the extra computation that it performs to plan each rover's motion. We will charge a flat fee of 10% of the score for the percepts. So, the team score

Terrain Name	*Perfect Score*
`hostile_topology`	2,306
`hostile_topology2`	2,181
`hostile_topology3`	2176
`longroute`	2,404
`maze`	1,163
`stuck`	2,496
`Terrain1`	2,500
`Terrain2`	1,850
`Terrain3`	1,972
`Terrain4`	2,188
`Terrain5`	1,702
`Terrain6`	2,227
`Terrain7`	1,978
`Terrain8`	1,848

Table 2: Perfect scores for the terrains in boards/

should be adjusted to 90% of the actual score earned. Next, we subtract one score point for every 5000 nodes expanded by each rover during search. Instrument your search program to count the number of nodes expanded during search and adjust the team score to reflect the cost of search. Tabulate scores for the ant agent against the adjusted score for the search-based agents for each of the eight terrains. Explain your results. What can you conclude about the relationship between the complexity of the environment and the complexity of agent design?

7.1 Implementation details

In order to implement the SmartRover, you need only provide an implementation for the method `SmartRover.nextAction(SmartPercepts p)`. You can find the method in the `src/student/SmartRover.java` file. The simulation will automatically call the methods when the rover type is set appropriately. The rover type can be set through the "Simulation" menu in the GUI, or by using the `-r` or `--rover` flags on the command line.

The `SmartPercepts` class contains all of needed sensors for each rover type. To perceive the world, you should call the methods on these percepts objects passed to you in the `nextAction(...)` method. You can access the `SmartRover`'s percepts using the following methods in the `SmartPercepts` class, an instance of which is given to the `SmartRover` in `nextAction(...)`.

- `isBumping()` returns whether or not this rover bumped into something as a result of its most recent action.

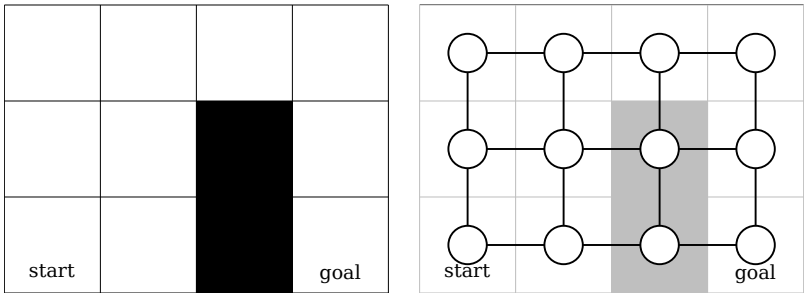
- `missionTime()` returns the elapsed mission time.
- `coinToss()` returns a random boolean.
- `getMap()` returns the current map of the world, using the combined exploratory knowledge of all of the rovers. Thus, a square that has been explored by any agent will be visible to all agents. This map is an instance of the `ExploredTerrain` type. You can call the method `getTerrainType(...)` to find the terrain type of any location on the map. Possible terrain types are `CLEAR`, `IMPASSABLE`, or `UNEXPLORED`.
- `rovers()` returns a collection of all of the rovers exploring the map.
- `getPosition(AbstractRover<?> rover)` returns the position of a rover.
- `myPosition()` returns the position of the calling rover.
- `getMemory()` returns an instance of the `SmartRoverMemory` class that this rover can use to store things if it so desires.

8 An online search algorithm: dynamic A*

Dynamic A*, or D*, is a variation of A* that has been successfully used in robot navigation in unknown terrains. In particular, it is now widely used in the DARPA Unmanned Ground Vehicle Program. It has been integrated into Mars Rover prototypes, and military robots for urban reconnaissance.

We will assume for our problem that robots are capable of error-free sensing and motion. This assumption will be relaxed in the second half of the course. Navigation in an unknown terrain is modeled as a search task over a finite undirected graph $G = (V, E)$ where a subset B of the edges E are blocked (due to obstacles). The robot knows V and E. It begins at a designated start vertex, and is given a goal vertex to get to. What it does not know is which subset of the edges are blocked. When the robot is at a vertex v, it learns which edges are incident to v are blocked. Dynamic A* uses A* to compute the shortest path to the goal vertex G assuming that all the edges it hasn't encountered are not blocked. It follows the computed path, until it meets a blocked edge, at which point it updates its map by adding the observed blocked edge to a list of blocked edges. It continues by finding another shortest path from the current vertex to the goal vertex in the updated map with the added blocked edge (and assuming unknown edges are traversable.).

For example, consider the terrain presented in 1(a) and the optimistic representation in Figure 1(b). Also consider the example run of dynamic A* through that terrain presented in Figure 2.



(a) A possible terrain containing obstructions.

(b) An optimistic graph that could be created for the given terrain. Notice that edges that are blocked in reality are added optimistically until they are discovered to be untraversable.

Figure 1: An example of creating an optimistic graph-theoretic representation of a terrain.

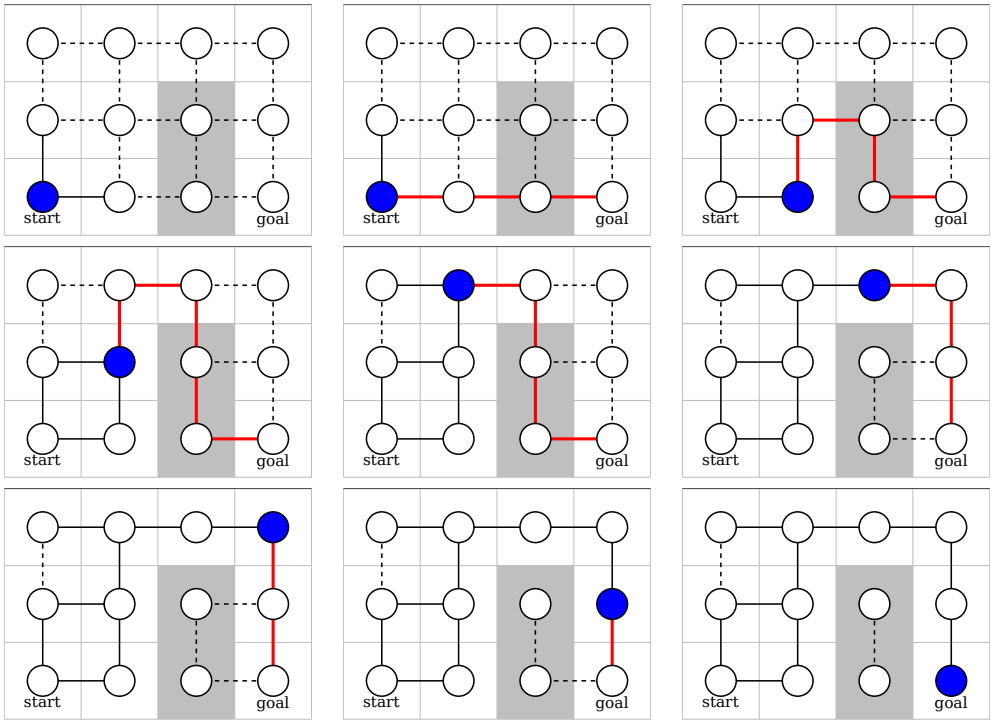


Figure 2: An example execution of dynamic A* on the previously presented terrain. In the upper left, a dynamic A* agent (blue) is placed into the terrain. In the frame to the right, it plans a course (red). It then continues to reach its goal. Optimistic but unconfirmed edges are dashed. Edges known to exist are solid black, whereas edges known not to exist (or to be impassable) are not shown.

Acknowledgements

I would like to thank Ian Arnold (senior in the CS department) for his hard work in building the rover simulation. Ian developed the code base and helped write this document. He will serve as teaching assistant for this module – you can reach him on piazza and in person during office hours the week of September 16.