Mars Explorer

MAS Project Report

Ionel Alexandru Hosu

Mihnea Dobrescu-Balaur

Introduction and problem presentation

The Mars Exploration problem is a well-known problem in AI, especially in the field of Multi Agent Systems. In its simplest form, it defines a world with obstacles and collectible objects (Mars and Martian rocks), together with a simple type of reactive agents - the explorers. The agents are supposed to explore the world, collect rocks and bring them to the base.

Being reactive agents, the explorers have a limited set of capabilities:

- they can sense if they are on a rock
- they can pick up a rock
- they can sense if they are near the base
- they can drop a rock at the base
- they can carry a finite amount of rocks
- they can move around freely, as long as they do not hit obstacles

On top of the constraints of the simple version of the problem, other ideas can be added:

- the explorer agents could leave trails representing their path from a rock back to the base
- other explorer agents could sense the trails and follow them hoping to find other rocks (assuming rocks appear in clusters)
- new types of agents can be introduced, even cognitive ones

The Mars Explorer problem is interesting because, depending on the chosen restrictions (as described above), different concepts from AI and MAS can be applied, leading to a diverse set of possible solutions.

Design decisions and implementation - Part I

For Part I, we had to implement only one kind of agent - the reactive explorer agent. It has all the capabilities mentioned above.

We made the assumption that one explorer can only carry one rock at a time. This decision was made in order to make the problem harder in the first part, and also allow room for change in the second part, where carriers will be introduced.

The first part was the more complex part of the two, because we had to start from scratch, building a framework on top of which part II (and possibly others, in the future) could be implemented.

We started with the world representation and the GUI. Having decided to code in Python (using Linux as the OS), because of the expressivity of the language and our familiarity with it, we chose the Tk graphics module, that is built-in to the Python standard library. The first step was creating the "game" window, with some statistics such as the number of ticks that have passed. We also had to implement a small feature such that the window will be displayed in the center of the screen, no matter what screen size.

After we were a bit experienced with how to use Tk for creating windows and drawing simple shapes, we started designing and then implementing the entities in our world. We identified a base "drawable entity", that represents the base class of all our entities. This class specifies a simple interface - the `draw` method, which is given the canvas as an argument. Thus, any class that is a "drawable entity" knows how to draw itself on the canvas. In addition to this, all drawable entities have a `get_bounds` method that returns the bounding rectangle of their shape. We use this in our implementation for doing collision detection.

Having the base class, we started implementing the entities from Mars Explorer, one at a time, and adding them to the GUI. The first ones were the more simpler, static entities: the world (the 2D space where everything lies - this would be planet Mars), the obstacle, the rock and the Mars base. We also implemented a set of functions that helped us generate rocks and obstacles on the surface such that they do not overlap.

After implementing every entity, we also added it to the world (and, implicitly, to the GUI). We have a simple frameork in place where the GUI "ticks" continuously until the game ends (all rocks are collected) and for every tick, it tells the world to process a "step" and then it draws everything. When processing the step of time, the world knows to tell all its inner entities (e.g. explorers) to process the step of time as well. For example, at every tick, the world tells every explorer to think and act. After every entity has finished its "tick behavior", they are drawn on the canvas.

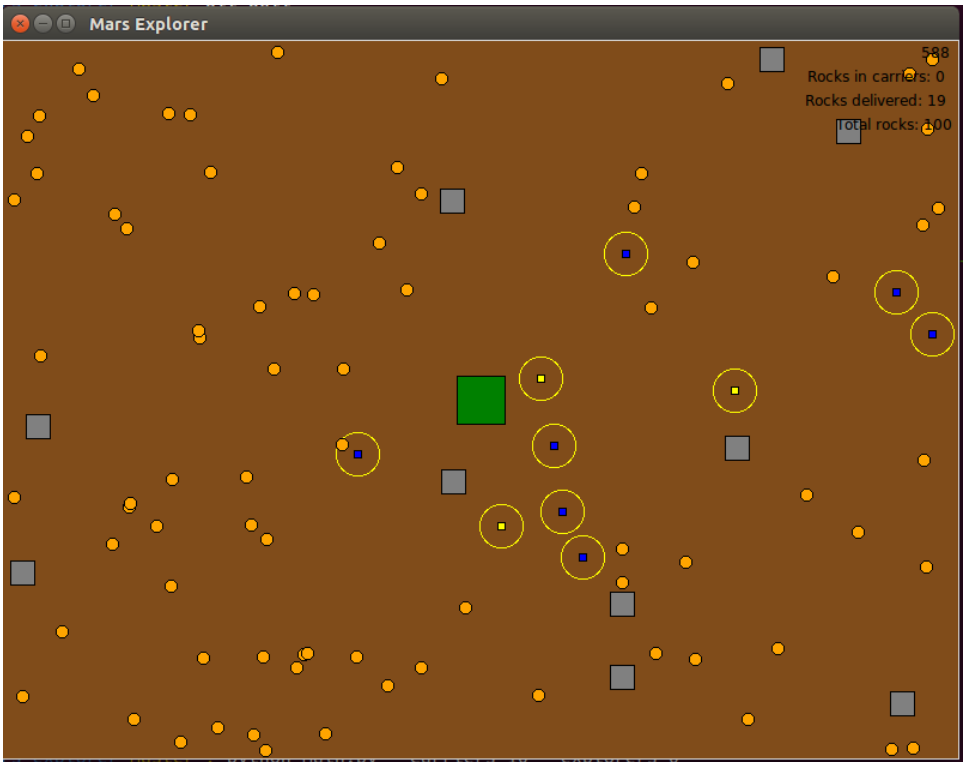
Having implemented all the static entities, we continued with the explorer. We started with a simple agent that simply moves randomly across the world, and then we improved it by adding behaviors such as:

- sensing nearby rocks
- pursuing sensed rocks
- knowing how to return to the base
- avoiding obstacles

The explorers move in a 2D continuous world. All entities have a position described by X and Y coordinates, and a size. Using these, we can compute the bounding box for any entity. The explorers move using a normalized direction vector.

The inner logic of the explorers relies mostly on their state - whether they are carrying a rock or not. If they are carrying a rock, they try to move to the base to drop it. Otherwise, they try to keep exploring by moving in the current direction, until they sense a rock. When a rock is sensed, they set their direction towards that rock and pursue it until they pick it up.

We decided to draw the sensor range in the GUI as well. We also change the explorer's color depending on their state - if they have a rock or not. We feel this makes the evolution easier to track.



Design decisions and implementation - Part II

The second part required adding a new type of agent - the carrier. These are smarter agents that can communicate with and reason about other agents. In addition, they can carry an unlimited amount of rocks.

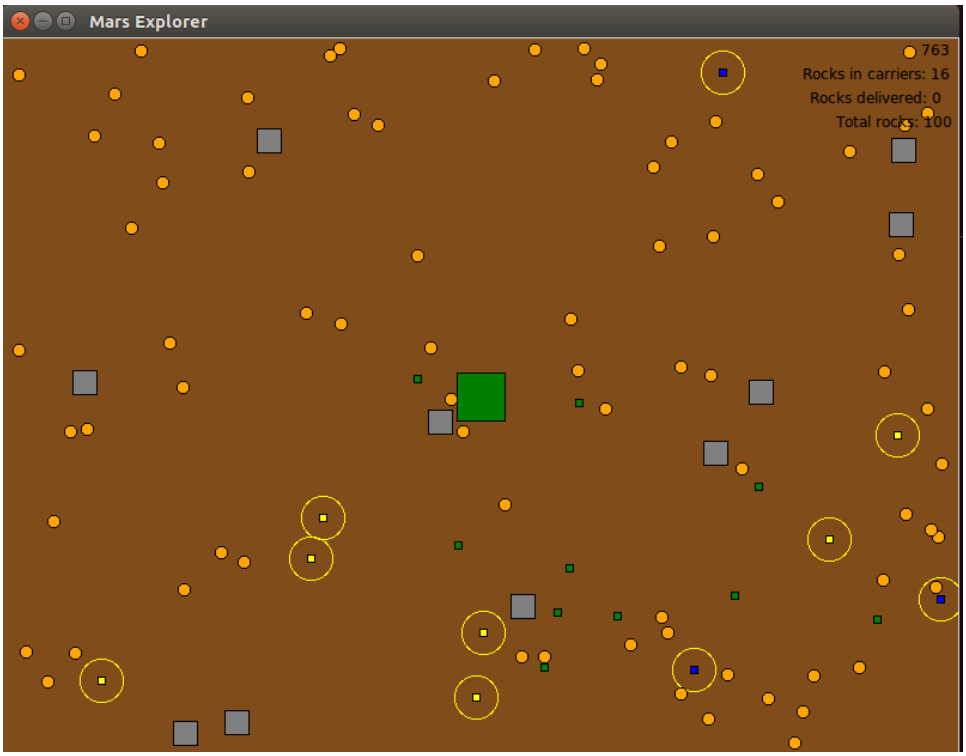
The strategy we decided to use was for explorer agents to keep exploring and finding rocks like in part 1, but when they find a rock they stop and broadcast a signal. The signal requests from carriers to come and pick up the rock. The carriers receive a number of broadcast messages from various explorers and decide to which explorer to go. They start heading towards that explroer and when they reach it they pick up their rock. The explorer then continues looking for other rocks, and so on.

Carriers communicate between them, and when they know that no other rocks are to be picked up, they all rush to the base to deliver all the rocks. When this happens, the game ends.

To implement the carrier, we decided to extend the explorer in order to reuse some of its behavior (collision detection, finding a direction to the base etc.).

For deciding which explorer to choose, we first tried to choose the closest explorer by euclidean distance. However, this did not yield the best results and we ended up choosing a random explorer. We found that this way the distribution of carriers across the map is better.

We represent the carriers with a different color and without showing the sensor range - because they do not use a sensor. Also, the carrier changes color depending on its state - there are two choices: en route to an explorer or waiting for an explorer to call for help.



In this part we also decided to implement command line arguments, such that the user can start the game with any given number of carriers, explorers, rocks and obstacles. Also, if there are no carriers, the explorers know to return to the base on their own, so there is no need for two executables for changing between part 1 and part 2 - one can simply pass in 0 as the number of carriers.

Results and interpretations

We have tried multiple configurations. We list below the most important with the average number of ticks needed to collect (and drop at the base) all rocks.

No carriers

5 explorers, 100 rocks, 10 obstacles - 18041

10 explorers, 100 rocks, 10 obstacles - 12539

15 explorers, 100 rocks, 10 obstacles - 6278

20 explorers, 100 rocks, 10 obstacles - 4351

25 explorers, 100 rocks, 10 obstacles - 5309

Carriers

10 explorers, 5 carriers, 100 rocks, 10 obstacles - 11449

10 explorers, 10 carriers, 100 rocks, 10 obstacles - 9520

20 explorers, 10 carriers, 100 rocks, 10 obstacles - 8138 (but also some results as low as 5500)

We noticed that for part 1, increasing the number of explorers helped until a point, from where the benefits were no longer seen. As the results show, 25 explorers had worse results than 20.

For part 2, we did indeed see that carriers help reducing the time but in our case not by much. We also noticed (especially in part 2) that different runs might have completely different results (e.g. 10000 ticks 5000). We will list possible improvements in the next section.

We also tried implementing a behavior where explorers do not stop and wait for carriers, but they continue moving and the carrier tries to meet them.

20 explorers, 10 carriers, 100 rocks, 10 obstacles - 11057

Proposals for improvements

There are a number of possibilities for improving the results. For explorer agents, we could make them detect obstacles from a distance (the same way they detect rocks) and do better at avoiding them. This is a major issue with the current implementation, as explorers sometimes get stuck near an obstacle and it takes many iterations for them to get past it.

We could also improve carriers. Since they are cognitive agents, we could use pathfinding algorithms such as Dijkstra and A* to compute more efficient paths to the explorer agents. Also, the carriers could coordinate themselves such that two carriers do not target the same explorer. Finally, we could implement something similar to the traveling salesman problem and have a carrier decide a more complex plan of multiple explorers to target.