CSC 361 Project
Phase #2

Out date: November 13, 2017
Due date: December 13, 2017

Abstract

Artificial Intelligence is increasingly used in many real-life fields. Path finding, for example, is a very popular application of AI. It is concerned with finding a valid path for an agent from an initial configuration to a goal configuration through an environment that may contain obstacles. Path planning is very important in games, map navigation, and robotics. In the second phase of this project, you should use the AI knowledge you acquired to model a new path finding problem environment, write search algorithms that can solve the problem, and compare the algorithms' performance.

1 Introduction

In phase 1 of this project we formulated the robot map navigation problem (AKA the maze, or the grid). In the map navigation problem you have a map consisting of a square cells grid. some of the cells are blocked (have obstacles that the robot cannot walk through) and some are empty (free areas). The robot is located on one of the empty cells, and it wants to go to a goal cell. The robot is allowed to move one step at a time. The possible moves are to the north, south, east, and west.

In this phase, you are supposed to formalize an extended version of the problem where a battery is used, and to write a fully functioning path finding system for the robot. You should implement 3 different algorithms: an uninformed search algorithm, an informed search algorithm, and a local search algorithm. The system chooses the algorithm to run based on the user input. Your code should be tested on some unsolved input maps, where the goal for the robot is to find a treasure. As in phase 1, the input maps will be a text-based representation of the problem instances, although in this phase there is not a command file. Your code should provide the output files that should include a list of the moves to be made by the robot, along with the final map after running these moves. You should finally write a report, explaining your formulation of the problem, your algorithms, your experiments, your results, and the difficulties you faced.

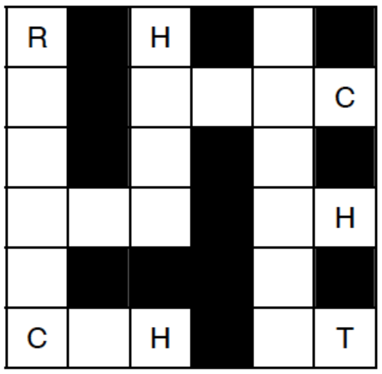

Figure 1: A maze example

Figure 1 shows an example of a maze in the new environment. The 'R' represents the robot, the 'H' represents a square that contains a hole, the 'T' represents a square that contains a treasure (the goal), and the 'C' represents a charging station where the robot can recharge its battery. The black squares are blocked.

In the following sections we explain the constraints and assumptions of this new environment, which expansions are optional and which are required, the algorithms you need to implement, and the formats and conventions of the project.

2 Constraints and Assumptions

As you know from phase 1, the goal in this environment is to let the robot find a path through the map that leads to the treasure. In this phase we add a battery to the robot that is initially fully charged. To get to the goal, the robot can use 4 move actions: `move-N`,`move-S`,`move-E`,and `move-W`. These actions can move the robot from its current cell to the next cell to the north, south, east, and west, respectively. However, the robot now consumes one unit of energy every time it makes a move. Consequently, it may run out of charge. Therefore, we add a new action called `recharge`, which will allow the robot to recharge its battery once it is in a charging station.

The environmental constraints are straight forward. The old constraints remain as is. For example, the robot cannot move to a blocked cell and cannot escape a hole. It cannot go beyond the map's edges as well. Some new constraints were added since the robot now has a battery. For example, the robot cannot move if it is out of battery charge, and the moves consume the battery charge. The robot can fully recharge its battery only in a charging station.

These constraints need to be formalized in your problem model as well as other sensible constraints you think must be there. A more concrete view of the constraints will be presented in the next sections.

We also make some assumptions to simplify things. These assumptions can be changed or removed in later phases of the project. The assumptions are:

- Currently there is only one robot and at least one treasure.

- The initial position of the robot must be in an empty cell.
- There cannot be more than one copy of the robot or the treasure in one cell. For example: there cannot be 2 treasures in one cell. There can be a treasure and a robot in one cell though.

Given these assumption, however, you should not add more simplifying assumptions. For example, you cannot assume that the treasure cannot be in a hole or in a charging station.

3 Model Extensions

In the previous phase we suggested that you try adding more constraints and extensions to the environment. In this phase you are **required** to add 2 new extensions: the battery and the charging stations. These extensions *will* be part of the evaluation. In the following we explain each of these two new extensions, and we briefly mention the other optional extensions.

3.1 Battery

We will add a battery to the robot and it will have a limited amount of power. Therefore, the robot may run out of power as it tries to do the tasks. Initially, the battery has a capacity that is equal to the number rows + the number of columns in the grid. Notice that you need to change the moves' effects and pre-conditions. For example, when the robot moves north, its battery charge shall be reduced by 1 unit, but when its battery goes out of charge, it will not be able to move.

3.2 Charging Stations

Since the moves consume power, and since the battery has a limited capacity, the robot may run out of power before reaching the goal. Therefore, we will add a number of charging stations to the map, to which the robot can go and fully recharge it battery. Notice that we need to add a `recharge` action that can be performed only in a charging station.

3.3 Optional Extensions

The other 2 extension suggested in phase 1 are still optional. If you add them, you get extra marks. The extensions are: the booster moves, and the cells altitude. However, as usual, the optional extensions' code should be separate from the required code. If you have already done them in phase 1, they will not be considered in this phase.

4 Algorithms

You are required to implement three search algorithms: **Breadth First Search (BFS)**, **A***, and **Hill-climbing (HC)**. The user of your program should be able to choose one of these algorithms to solve the problem. Each of the algorithms should be able find a solution to the problem (if it is a complete algorithm of course.) The solution must show how the robot can move from its initial position to a position where there is a treasure. If the algorithm finds that solution, you should write the sequence of actions that represent the solution in the output action file as we will explain in the next sections. If the algorithm cannot find a solution, then you should write `NOOP` in the output action file.

The A* and the Hill-climbing algorithms need heuristics. The heuristic that you must use with these algorithms is the **Manhattan distance** heuristic. It simply counts the number of squares needed to reach the goal regardless of the blocked cells or the battery charge level.

The algorithmic main frame of the BFS and A* must be either Tree-search or Graph-search. But you are allowed to make minor changes to them. It is important to think about that. Are you going to use plain Tree-search? plain Graph-search? or a modified Tree-search that prevents cycle? This is important because the number of states that can result from repeatedly expanding the same state can be astronomical. You have 3 choices: (1) to ignore this warning and use the plain Tree-search algorithm without change for both BFS and A*, allowing the risk that you will frequently face segmentation faults or out of memory errors, (2) to modify the Tree-search algorithm by preventing cycles within the same path (just check the same path for repeated states), or (3) to use the Graph-search algorithm (this implies implementing a new data structure for the visited states we called *closed*).

In the following subsections, we present optional extensions to the heuristics and the algorithms.

4.1 Extra Heuristics

In case you did not notice, the Manhattan distance heuristic is not perfect when you add batteries and charging stations. You can add a more accurate heuristic to improve your search. However, in the submitted code, only the Manhattan distance heuristic should be used. If you implement the extra heuristic, then you should explain it in the report and compare it with the Manhattan distance heuristic, and then submit its code separately.

4.2 Extended Hill-climbing

As you know, Hill-climbing algorithm is incomplete due to plateaux and local minima (local maxima if you use objective functions). You can extend the Hill-climbing algorithm to escape local minima or plateaux that are found during the search. For example, you can invoke a random walk strategy once you are in a plateau to escape it. This extension is only optional, so you should not include it in the main code.

5 Program Interface

In this section we introduce how we pass the input to your agent and get the output. However, we do not want to lose what we wrote in phase 1 of the project. We would like to give the user the choice to either use the robot as a command follower (phase 1), or as a thinking agent that can automatically find a correct path to the treasure (phase 2). Therefore, the execution of the Agent code will be done in two modes:

1. command-following mode (c)
2. and search mode (s).

Your program must be invokable from the command prompt. The program name must be `'Agent'`. So, the general usage should be as follows:

```
./Agent  [mode]  [par1]  [par2]  [par3]  [par4]
```

where `[mode]` is the command mode and can take one of two values: `c` or `s`, and `[par1]`, `[par2]`, `[par3]`, and `[par4]` are the four parameters required for each mode. Depending in the mode, the parameters will change as we will explain in the following subsections. Of course if you are using Java, you should add the word `java` in the beginning.

5.1 Command Following Mode

The command following mode has already been implemented in phase 1 of the project. As discussed in phase 1, the command mode will have four parameters. Therefore `[par1]` will be equal to `[mapFile]`, `[par2]`= `[commandsFile]`, `[par3]`= `[finalMapFile]`, and `[par4]`= `[logFile]`. So, the usage in this mode will be as follows:

```
./Agent  c  [mapFile]  [commandsFile]  [finalMapFile]  [logFile]
```

5.2 Search Mode

In the search mode (phase 2), the executable file should accept four different parameters; two for input and two for output. The input parameters are:

1. A number that represents the algorithm to be used: `[na]`. The possible values for `[na]` are: 1 for BFS, 2 for A*, and 3 for Hill-climbing.
2. The name of the map file: `[mapFile]`.

The executable should accept two more parameters that tells it where to store the output files. The outputs should be stored in two files:

1. The action file: `[actionFile]`.
2. The final map file: `[finalMapFile]`.

The `[mapFile]` and the `[finalMapFile]` represent maps and, therefore, have the same file formate. All of these files must be text-based, i.e. you can edit and view them using a text editor. All files' format will be explained in the next sections.
So, the usage in this mode should be as follow:

```
./Agent  s  [na]  [mapFile]  [actionFile]  [finalMapFile]
```

or if you are using Java you should invoke it as an executable class as follows:

```
java  Agent  s  [na]  [mapFile]  [actionFile]  [finalMapFile]
```

where the input and output files must be located at the same directory as `Agent`.

For example, assume that someone wants to solve the map `map.txt` using your Java implementation of the A* algorithm. He wants to store the solution in a file called `solution.txt` and he also wants to store the final shape of the map after running the algorithm in a file called `finalmap.txt`. Then, he should write the following in the command prompt:

```
java  Agent  s  2  map.txt  solution.txt  finalmap.txt
```

It is important to follow these conventions, since we will test your code using a script which assumes that you strictly followed them. The following section explains the format of the input and output files mentioned above.

6 Files Format

In this phase, the robot should figure out how to reach the goal without any commands to tell him how to do that. This means that the output file will be different from the previous phase's outputs. Also, since we have two new extensions to the problem description, the input maps will change as well.

There are two kinds of files in this phase: the map file and the final action file. The map file explains the map and is contents (the robots, the treasures, the charging stations, etc.) at a certain point during the search. The final action file represents a list of moves which are supposed to lead the robot from its initial position to goal. The final action file is similar to the commands file in phase 1, except that it is an output file this time. In the following we explain each file format in detail.

6.1 Map File Format

Like phase 1, the map file will have the following format:

```
[n]
[m]
[the cells]
```

where `[n]` is the number or rows, `[m]` is the number of columns, and `[the cells]` is a sequence of lines of characters representing the contents of the map grid. The cells are shown in `n` lines of characters. The number of characters in each line is `m`. The character can represent the static as well as the dynamic features of the cell. Static features cannot change in a cell. For example, blocked cells remain blocked, holes remain holes. There is one new static feature, though, which is the charging station. The static cell features are represented by the following characters:

- ' ':the space character represents an empty cell, or a cell that the robot can move on.
- 'H':this represents a hole, or a cell in which the robot can fall. The robot can never go out of the hole once it falls in it.
- 'B':this represents a blocked cell. The robot cannot move to or originate at a blocked cell, as well as all other objects (e.g.
- 'C':this represents a charger cell. The robot can move freely on the charger cell and can recharge its battery.

Also notice that there is only one static feature in the cell. So, for example, no two charging stations can exist at the same cell, and a charing station cannot be found in a hole.

The dynamic cell features are those that may change and can exist with other features. The dynamic features remain the same as in phase 1. The dynamic features are represented by the robot and the treasures. In a map, there is exactly one robot and at least one treasure.

When the dynamic features exist in a cell with each other and with static features, new characters are needed to represent the cell on the map . Here is the list of all cases of the cells with their corresponding characters:

- Empty cell with nothing on it: ' '.
- Empty cell with only a robot on it: 'R'.
- Empty cell with a treasure: 'T'.
- Empty cell with a treasure and a robot: 'U'.
- Hole cell with nothing in it: 'H'.
- Hole cell with a robot only in it: 'X'.
- Hole cell with a treasure only in it: 'Y'.
- Hole cell with a treasure and a robot in it: 'Z'.
- Blocked cell: 'B'. The blocked cell cannot contain any dynamic features.
- Charger cell with nothing on it: 'C'.
- Charger cell with a robot only on it: 'D'.
- Charger cell with a treasure only on it: 'E'.
- Charger cell with a treasure and a robot on it: 'F'.

An example of the text file representing the map of Figure 1 is as follows:

```
6
6
RBHB_B
_B___C
_B_B_B
___B_H
_BBB_B
C_HB_T
```

where the underscore symbol is used here to represent the space. Do not use the underscore symbol in your output final map file.

6.2 Action File Format

Unlike phase 1, your code should output a file that contains a list of actions (moves and/or recharges) called the action file. This list of actions should lead the robot to the goal. For example, given the map file presented above, your code might generate the following action file:

```
move-S
move-S
move-S
move-E
move-E
move-N
move-N
move-E
move-E
move-E
recharge
move-W
move-S
move-S
move-S
move-S
move-E
```

Notice that using `move-?` actions solely in this map will not lead to the goal, since the amount of charge in the battery is initially = `n+m=12` while the treasure is 14 moves away. Also notice that recharging in the lower charging station is not a good idea, since the distance from that charging station to the treasure requires more than 12 move.

Also notice that the search algorithm may fail to find a solution. The Hill-climbing algorithm for example can fall into a local minima and exit before finding a solution. In that case, your action file must end with the word: `NOOP`. Finally, as in phase 1, you must verify that all actions are valid, i.e., their precondition are present when you invoke them.

7 The Report

In a small report (no more than 10 pages), write the following sections:

- In the introduction, write a short, informal description of the new problem (with the charging stations), and briefly describe how you will solve it (what algorithms you will use.)
- Then write a full, formal problem definition of the new problem. Namely, find:
  - the state description, specifying explicitly the initial state if known.
  - the actions and their transition model.
  - the goal test.

- the path cost (the step cost of the actions.)

- Write a brief description of the algorithms you implemented and the heuristics you used.
- Write a short description of the new features that you have added to the algorithms. For example, if you have implemented a new heuristic for the A*, or a new method to escape local minima in the Hill-climbing algorithm, write their description in this section. This is an optional section if you have only implemented the mandatory features.
- write a comparison of the 3 algorithms in terms of:
  1. the runtime
  2. the number of nodes explored
  3. the solution cost

  It is recommended that you use tables, graphs and/or diagrams to illustrate the difference between the algorithms. You should also analyze the results and draw reasonable conclusions.

  You should think carefully before writing a conclusion. To avoid mistakes, we suggest that you write conservatively, i.e. do not generalize the conclusion unless there is an undoubted, very clear evidence to support it. For example, if the average run time for the A* algorithm was 10 minutes after running it on 5 different maps, 10 times each, while the runtime for the Hill-climbing was 30 seconds after running it on the same maps for the same number of time after finding the solutions, then you could conclude that the Hill-climbing was faster than the A*. However, if A*'s runtime was 10 seconds and the runtime for the Hill-climbing was 9.5 seconds , it is hard to conclude something concrete about the speed.
- Write a small summary section, describing the problem, the algorithms tested, the analyzed results, and the conclusions.

In the report, you should clearly explain the points of weakness and strength of your work. That is, write about any difficulties you faced in the project. For example, if your A* did not find the correct solution with some files, write about that and explain the possible reasons. You should also write about anything you think is positive or you think is important to mention.

8 Deliverables

You should submit the following:

1. The **report** that explains the work. The report should be submitted using a separate link in LMS. **(40/100 marks)**
2. The compilable, runnable **source code** with instructions on how to build the executable. Also you must include the **executable file**. It should be the same file that results after compiling. **(20/100 marks)**
3. The **output files** that resulted after running the executable on the provided input files. Notice that every student will be provided with input files for the test, and these files may differ from a student to another. **(40/100 marks)**

9 How to Submit the Deliverables?

First, let's assume that your student ID is 433000123. Change this number to your real ID when you are about to submit. You should submit everything using LMS. You will find 3 links, one for each deliverable:

1. Under the `Project/Phase 2/REPORT-SUBMIT-2` link in LMS, you should submit a PDF copy of your report. The file name should be:

   `433000123.report.pdf`.

   The report will go through a plagiarism test, where the percentage of similarity with other reports (e.g. your colleagues', the Internet) is shown. If you have a high similarity percentage, then I will check the report further, since you might have copied something from other reports and did not cite it. **If I discover that this was the case, then it is considered cheating, which will cause you to FAIL the project, and perhaps the course.**
2. Under the `Project/Phase 2/SOURCE-AND-EXEC-SUBMIT-2` link, you should submit the **source code** and the **executable file**. The source code will also go through a code plagiarism test. All files in this case should be put in a **flat** ZIP file called:

   `433000123.source-exec.zip`.
3. Under the `Project/Phase 2/OUTPUT-SUBMIT-2` link, you should submit the **output files** that resulted after running the code on the provided input files. Notice that every student will be provided with input files for the test, and these files may differ from student to student. Submit **your** output files only. I will check if these were your original output files by running your code. All files in this case will be put in a **flat** ZIP file called:

   `433000123.output.zip`.

   More about the input and output files naming convention will be given to you later.

Notice that submitting on time is your responsibility. Late submissions will *not* be accepted.

10 Important Guidelines

- Write your name, student ID number, section number, and project phase number on the report and as a comment in the code.
- Work individually. Discussing the project with your colleagues is encouraged as long as there is not a transfer of code or text. **Cheating will not be tolerated.**
- Use LMS to submit the soft copy for all of the deliverables, including the report, as described above. Emails will not be accepted.
- Make your report be as clear, precise and concise as possible.
- If you have questions about the project, *do not* email them to me, but rather *post* them on the discussion board. Either I or a student colleague may answer them for everyone to see. By using the discussion board, it is likely either that you will find your questions answered on the board, or that someone else has the same questions where posting them will help him also.