Artificial Intelligence

Search Algorithms Coursework

Assigned: 16 March

Due: Tuesday, 28 March, 19h30m

Prof. Felipe Meneguzzi

Ramon Fraga Pereira (assistant)

March 14, 2017

1 Hyrule's Maze

You must work on this project **individually**. You are free to discuss high-level design issues with the people in your class, but every aspect of your actual implementation must be entirely your own work. Furthermore, there can be no textual similarities in the reports generated by each student. Plagiarism, no matter the degree, will result in forfeiture of the entire grade of this assignment.

The Legend of Zelda is series of games originally designed by Shigeru Miyamoto[1], Takashi Tezuka and Eiji Aonuma whose first game was released in 1986. Games in the series are played on a fantasy world called Hyrule, which (up until the Super Nintendo) is often represented as a 2D grid map in which the character has to navigate, avoiding obstacles and reaching a goal[2]. In this assignment, we greatly simplify Link's movement by allowing only orthogonal movements (up, down, left, right) within the grassland environment. An example of a set of legal moves is shown below in Figure 1.

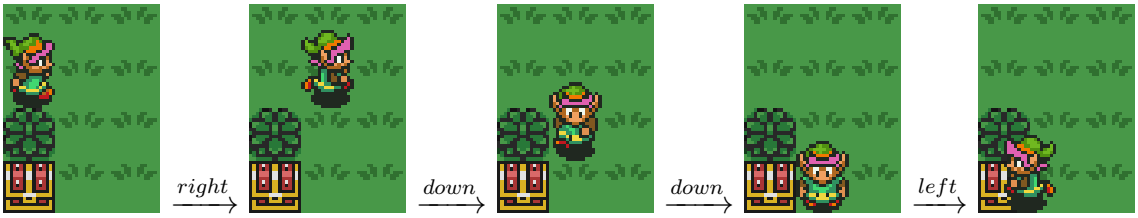


Figure 1: A sequence of valid moves for Link's navigation problem

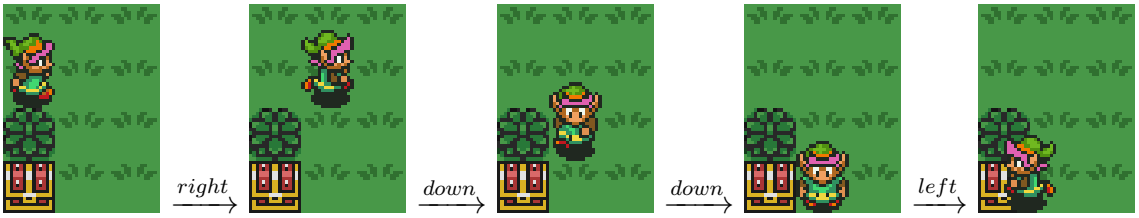
In this assignment, we help Link with a simple navigation task to move between an initial state and the goal state, represented by a treasure chest. Thus, we may specify problems such as the one shown in Figures 2a and 2b.

[1] Yes, the same guy from Mario.

[2] Following with a patriarchal view of society, our helpless princess Zelda, is usually captured by some villain, and our hero Link, must rescue her.

(a) Initial State

(b) Goal State



(c) Solution and intermediary steps.

Figure 2: Example of an arbitrary problem with solution.

2 Overview

For this assignment, you will be implementing the A* informed search algorithm to solve instances of the navigation problem posed to Link. As an informed search method, A*, relies on an heuristic distance function in order to efficiently prune the state space and speed up the search tree expansion towards the solution. In class, we have seen two possible heuristics that can be applied to this problem, but in this assignment, we will focus on the Manhattan Distance Heuristic. This heuristic consists of distance in orthogonal movements between Link and its desired destination. Thus, if we consider the same example, Link is two blocks away from his desired position.

(a) Initial State

(b) Goal State

$$H(Fig\ 3a) = 2$$

Figure 3: Manhattan Distance Example

3 Implementation and Deliverables

Organize your implementation in an appropriate number of classes, modules and methods. At the bare minimum, you are required to implement only the API in Listing 1, below. However, it is highly recommended that you create individual classes to represent elements

in the state space as well as nodes in the search tree. We note that the API shown below in Listing 1 is deliberately simplified to give students the maximum freedom to develop their own internal APIs (and minimize coincidental similarities in implementation), and that well developed code is a component of the final mark. By way of comparison, the instructor's implementation has 3 classes and 6 additional methods (beyond those of Listing 1).

Listing 1: `pathfinder.py`

```
class PathFinder_A_Star:

    def __init__(self):
        # TODO initialize your attributes here if needed
        pass

    def function(self, p1, p2):
        # TODO priority function to use with the PriorityQueue
        # You are free not to use this function
            # (it is not tested in the unit test)
        return None

    def heuristic(self, p1, p2):
       # TODO heuristic function
       # You are free not to use this function
           # (it is not graded in the unit test)
       return 0

    def solve(self, map):
        # TODO returns a list of movements (may be empty)
                # if plan found, otherwise return None
        return None

    def get_solvable(self):
        # TODO returns True if plan found,
        # otherwise returns False
        return False

    def get_max_tree_height(self):
        # TODO returns max tree height if plan found,
                # otherwise, returns None
        return None

    def get_min_moves(self):
        # TODO returns size of minimal plan to reach goal if plan found,
                # otherwise returns None
        return None
```

At the end of this assignment, you will upload **one zip file** named `s<ID>.zip`[3]. In this zip file, there must be one folder named `s<ID>.zip`(same as before). This file must contain:

[3]Where `ID` is your student id number, e.g. if your ID is 13303580, then you must create `s13303580.zip`

- your implementation of `pathfinder.py` (using the Manhattan distance heuristic function) and any other classes you use to implement your solution (evidently excluding those already present in the core of Python);
  You are **not allowed** to use any external libraries besides the one supplied with the assignment package, nor are you allowed to modify the classes you received in any way (besides your own code in `pathfinder.py`).

- one `readme.txt` file answering the questions in the appendix;

- any unit tests you developed to test your classes, modules and methods.

Your deliverables must be handed in via Moodle using the appropriate upload room: Heuristic Assignment Upload.

Note that the code you develop must be able to run the unit test script from the assignment package, `test_pathfinder.py`.

Hand in: To facilitate your the process of handing in your code, we have included a python program called `zipper.py` that creates this zip file for you, in the correct format. Use it to create your deliverable package.

4 Grading

In order to properly evaluate your work and thought process (and to help improve future iterations of this course), you will complete a `readme.txt` answering the following questions.

1. Explain briefly how you implemented the datatype for states (i.e. to make it efficient in the closed list).

2. Explain briefly how you represented a search node (state + number of moves + previous search node).

3. Explain briefly how you detected unsolvable problems.

4. If you wanted to solve random 10^8 problem (i.e. a 10000×10000 grid), which would you prefer: more time (say, 2x as much), more memory (say 2x as much), a better priority queue (say, 2x as fast), or a better priority function (say, one on the order of improvement from Hamming to Manhattan)? Why?

5. If you did the extra credit, describe your algorithm briefly and state the order of growth of the running time (in the worst case) for isSolvable().

6. Known bugs / limitations.

7. Describe whatever help (if any) that you received. Don't include readings, lectures, and precepts, but do include any help from people (including staff, classmates, and friends) and attribute them by name.

8. Describe any serious problems you encountered.

9. List any other comments here. Feel free to provide any feedback on how much you learned from doing the assignment, and whether you enjoyed doing it.

Grading will take into consideration elements of your code, testing and reporting of the work done. The criteria, as well as their weight in the final grade is as follows:

- Basic Correctness (30%) — correctness of the algorithm as determined by the unit tests provided with the assignment;[4]
- Full Correctness (40%) — correctness of the algorithm for a more comprehensive suite of tests, which includes unsolvable problems and a **timeout of one minute** for problems in a 500×500 grid;
- Coding style (10%) — code organisation, object orientation, commenting and readability;
- Overall report readability (20%) — this relates to the `readme.txt` file you must fill out, we will assess how accessible and coherent the explanation of your code and solutions is; and
- Efficient unsolvable detection (bonus 20%) — writing an efficient algorithm to detect unsolvable problems without trudging through the entire state space.

5 Code-Specific Advice

- In common.py, you can change the current map modifying the `DEFAULT_MAP` constant;
- You can also change Link's movement speed modifying the `MOVE_SPEED` constant in the same file;
- All classes and methods which you must complete yourself are marked with the `TODO` keyword (search for that using your favorite editor);

6 Miscellaneous Advice

Here are some lessons we learned in creating our own solution and writing papers/reports:

- It took the instructor approximately 3 hours to code and debug the solution for this problem from scratch based on the pseudocode at Artificial Intelligence: A Modern Approach, plan your time accordingly;
- The instructor's implementation takes about 60 seconds to explore the *entire* state-space of a problem in a 500×500 grid;

[4]Do note that an implementation that simply returns the correct values for the test cases without an actual search algorithm will result in forfeiture of the entire score.

- You can automatically test your own code with `test_pathfinder.py` file that comes with the project package, simply open a terminal, go to your project folder and run `python -C test_pathfinder.py` in the command line.
- If your program is taking overly long to solve the problem, you can use the python profiler to find out where the problem is by running `python -m cProfile pathfinder.py hard.txt`, where `hard.txt` is your 500 by 500 map.