Assignment 1: Search

Assignment #1: Solving Hua Rong Dao using Search

Document History:

- Released Tue Sept 13 2022
- Added clarification for the Input Format in purple.
- Added **validation script (https://q.utoronto.ca/courses/278996/files/22188531?wrap=1)** **(https://q.utoronto.ca/courses/278996/files/22188531/download?download_frd=1)** to ensure that solution files will compile on teach.cs servers.

Introduction

For this assignment, you will be implementing a solver for the Hua Rong Dao sliding puzzle game. Hua Rong Dao is a sliding puzzle that is popular in China. Check out the following page for some background story on the puzzle and an English description of the rules.

http://chinesepuzzles.org/huarong-pass-sliding-block-puzzle/ (http://chinesepuzzles.org/huarong-pass-sliding-block-puzzle/)

The puzzle board is four spaces wide and five spaces tall. We will consider the variants of this puzzle with ten pieces. There are four kinds of pieces:

- One 2x2 piece.
- Five 1x2 pieces. Each 1x2 piece is either placed horizontally or vertically.
- Four 1x1 pieces.

Once we place the ten pieces on the board, two empty spaces should remain.

Look at the most classic initial configuration of this puzzle below. (Don't worry about the Chinese characters. They are not crucial for understanding the puzzle.) In this configuration, one 1x2 piece is horizontal, and the other four 1x2 pieces are vertical.

The goal is to move the pieces until the 2x2 piece is above the bottom opening (i.e. helping Cao Cao escape through the Hua Rong Dao/Pass). You may move each piece horizontally or vertically only, into an available space. You are not allowed to rotate any piece or move it diagonally.

There are many other initial configurations for this puzzle. Check out **this Chinese Wikipedia page (https://zh.wikipedia.org/wiki/%E8%8F%AF%E5%AE%B9%E9%81%93_(%E9%81%8A%E6%88%B2))** for 32 initial configurations. The link below each configuration opens another page where you can play the puzzle and see the solution.

Counting Moves: We will count moves differently from how the website does it. On the website, consecutive moves of a single piece count as one move. However, we will count the consecutive moves separately. Suppose that I move a single piece by two spaces to the right. We will count it as two moves, whereas the website counts it as one move. For the most classic initial configuration, the optimal solution takes 81 moves on the website, whereas it takes 116 moves based on our counting rule.

Your Tasks

Two Search Algorithms: A* and DFS

You will implement two search algorithms: A* search and Depth-first search.

Two Heuristic Functions for A* Search

If we want A* search to find an optimal solution, we need to provide it with an admissible heuristic function. You will implement two admissible heuristic functions for A* search.

You will first implement the Manhattan distance heuristic, the simplest admissible heuristic function for this problem. Suppose that we relax the problem such that the pieces can overlap. Given this relaxation, we can solve the puzzle by moving the 2x2 piece over the other pieces towards the goal. The cost of this solution would be the Manhattan distance between the 2x2 piece and the bottom opening. For example, for the most classic initial configuration, the heuristic value is 3.

Next, propose another advanced heuristic function that is admissible but dominates the Manhattan distance heuristic. Implement this heuristic function. Explain why this heuristic function is admissible and dominates the Manhattan distance heuristic.

Mark Breakdown

The tasks for this assignment consist of implementing the depth-first search and A* search for the Hua Rong Dao puzzle, where the A* search requires the implementation of a Manhattan heuristic and an original heuristic of your own. The mark breakdown for this is as follows:

- Depth-first search solution: **35%**
- A* solution (Manhattan heuristic): **55%**
- Original heuristic (implementation & description): **10%**

What to Submit:

You should submit two files:

- **hrd.py** contains your Python program, and
- **advanced.pdf** contains a description of your advanced heuristic function. This file should be no more than one page long. Please describe the advanced heuristic function and explain why it is admissible and why it dominates the Manhattan distance heuristic.

Your program **must use python3 and run on the teach.cs server** (where we run our autotesting script)**.**

We will test your program using a random subset of the 32 initial configurations on **the Chinese Wikipedia page (https://zh.wikipedia.org/wiki/%E8%8F%AF%E5%AE%B9%E9%81%93_(%E9%81%8A%E6%88%B2))**. For each initial configuration, we will run the following command:

```
python3 hrd.py  <input file>  <DFS output file>  <A* output file>
```

The command specifies one plain text input file and the two plain text output files containing the solutions found by DFS and A* for the puzzle.

For example, if we run the following command for puzzle 5, specified in a file called `puzzle5.txt`:

```
python3 hrd.py puzzle5.txt puzzle5sol_dfs.txt puzzle5sol_astar.txt
```

The DFS solution will be found in `puzzle5sol_dfs.txt` and the A* solution will be found in `puzzle5sol_astar.txt`

After submitting your code, we will run a script to **verify that your DFS solution is valid and your A* solution is valid and optimal**.

We have provided a **validation script (https://q.utoronto.ca/courses/278996/files/22188531?wrap=1) (https://q.utoronto.ca/courses/278996/files/22188531/download?download_frd=1)** that you can use to verify that your code compiles and will work with our autotesting script. To run it, unzip the file hrd_validate.zip, copy your hrd.py into the directory with the unzipped files and run the following command:

```
python3 hrd_validate.py
```

Note that **this does not test the correctness of your solution.** It is meant to prevent compilation errors with our autotester, and test the format of your output file by comparing your A* output with ours.

Input Format

The input to your program is a plain text file that stores an initial Hua Rong Dao puzzle configuration. See below for an example of the input file content. It contains 20 digits arranged in 5 rows and 4 digits per row, representing the initial configuration of the puzzle. **The empty squares are denoted by 0. The single pieces are denoted by 7. The 2x2 piece is denoted by 1. The 5 1x2 pieces are denoted by one of {2, 3, 4, 5, 6}, but the numbers are assigned at random.**

```
2113
2113
4665
4775
7007
```

Output Format

The two output files should store the DFS and A* solution for the input file provided.

See below for an example of the content of the output file. On the first line, print the cost of the solution. Next, print the sequence of states from the initial configuration to a goal state. Two consecutive states are separated by an empty line. **The empty squares are denoted by 0. The single pieces are denoted by 4. The 2x2 piece is denoted by 1. The horizontal 1x2 pieces are denoted by 2. The vertical 1x2 pieces are denoted by 3.** Due to limited space, we only show the beginning of the output file below.

Make sure that your output files match this format exactly.

```
Cost of the solution: 116
3113
3113
3223
3443
4004

3113
3113
3223
3443
0404

3113
3113
3223
3443
0440
```

Suggested Steps for Tackling This Assignment:

This assignment requires you to write a substantial amount of code. Also, your code needs to be reasonably efficient to terminate within the time limits. Therefore, we strongly suggest that **you write the program incrementally**. Start by writing helper functions and test them individually. Then, gradually build up a complete program.

Here are some suggested steps to tackle this assignment. Good luck!

- Implement a class to represent a **piece**.
  - You may want to keep track of the type of each piece. Is it part of the 2x2 piece? Is it a single piece? If it is part of a 1x2 piece, you may want to keep track of its orientation and location (e.g. the coordinates of its upper-left corner).
  - You can hardcode the numbers representing different types of pieces (1 for the 2x2 piece, 0 for single pieces, etc.).
  - You may want to write a function to move a piece.
  - You may want to write a function to print out the attributes of a piece for debugging purposes.
- Implement a class to represent a **board**.
  - A board has a fixed width and height. You can hardcode these.
  - You may want to write a function to create a grid of characters given the set of pieces on the board. This will be useful for printing the board to standard output or a file.
  - You may want to write a function to find the two empty spaces on the board. This will be useful for figuring out the legal moves for a board.
- Implement a class to represent a **state**.
  - A state is a wrapper around a board. The state keeps track of extra information relevant to the search. See some examples below.

- Each state can keep a reference to its parent state. This way, we do not have to store the sequence of states in the frontier. Instead, once we reach a goal state, we can use the parent state references to backtrack and recover the path from the initial state.
- Each state can keep track of the number of states from the initial state to the current state. This is the cost of the solution or the g value of the current state. Once we reach a goal state, this value gives us the cost of the solution.
- Each state can keep track of its heuristic value and f value. These will be convenient for A* search.

- Write a function to take an input file and return a board containing a set of pieces.
- We recommend using heapq as the frontier. You can use heappush and heappop to modify the frontier.
- Write a function which takes a state and returns true if and only if the state is a goal.
- Write a function which takes a state and returns the heuristic value (or the h value) of the state.
- Implement a function which takes a state and returns a list of its successor states.
  - Warning: This will likely be the most complicated function in your program.
  - Therefore, we suggest you break down this function into several helper functions.
- Write a function that takes an initial state and returns the optimal solution found by A* search.
- Write a function that takes an initial state and returns the first solution found by DFS.