

Assignment 1: Search

Assignment #1: Solving Hua Rong Dao using Search

Updates:

- **Alice mentioned on Wednesday that the time limit is 5 minutes for each puzzle and algorithm combination. However, due to MarkUs's technical constraints, we had to revise this time limit to 2 minutes. Please ask on Piazza or during office hours if you need tips on optimizing your program.**

Introduction

For this assignment, you will implement a solver for the Hua Rong Dao sliding puzzle. 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/> 

[\(http://chinesepuzzles.org/huarong-pass-sliding-block-puzzle/\)](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 can be horizontal or vertical.
- 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.

You can find many other initial configurations for this puzzle on [this webpage](#) ➞

[https://zh.wikipedia.org/wiki/%E8%8F%AF%E5%AE%B9%E9%81%93_\(%E9%81%8A%E6%88%B2\)\)](https://zh.wikipedia.org/wiki/%E8%8F%AF%E5%AE%B9%E9%81%93_(%E9%81%8A%E6%88%B2))).

Scroll down to see 32 configurations. The first 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

Implement DFS and A* Search

You will submit a file named `hrd.py`, which contains your implementation of a program that uses Depth-First Search and A* Search to solve a Huarongdao puzzle.

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

We will test your program using several initial configurations of various difficulties. For each initial configuration, we will run the following two commands:

```
python3 hrd.py --algo astar --inputfile <input file> --outputfile <output file>
python3 hrd.py --algo dfs --inputfile <input file> --outputfile <output file>
```

Each command specifies the search algorithm, one plain-text input file, and one plain-text output file containing the solution found by the search algorithm.

For example, if we run the following commands for an input file **hrd5.txt**:

```
python3 hrd.py --algo astar --inputfile hrd5.txt --outputfile hrd5sol_astar.txt
python3 hrd.py --algo dfs --inputfile hrd5.txt --outputfile hrd5sol_dfs.txt
```

The DFS solution will be found in **hrd5sol_dfs.txt**, and the A* solution will be found in **hrd5sol_astar.txt**.

Our tests will verify that

- Your DFS solution is valid.
 - The first state is the initial state. The last state is the goal state. Every state is a successor of the previous state.
- Your A* solution is valid, and
 - The first state is the initial state. The last state is the goal state. Every state is a successor of the previous state.
- Your A* solution is optimal.
 - The number of states in the solution is minimized.

The Input and Output File Formats

In the input and output files, we will represent each state in the following format.



- Each state is a grid of 20 characters. The grid has 5 rows with 4 characters per row.
- The empty squares are denoted by the period symbol.
- The 2x2 piece is denoted by 1.
- The single pieces are denoted by 2.
- A horizontal 1x2 piece is denoted by < on the left and > on the right.
- A vertical 1x2 piece is denoted by ^ on the top and v on the bottom (lower cased letter v).

Here is an example of a state.

```
^^^^
vvvv
22..
11<>
1122
```

Each input file contains one state, representing a puzzle's initial state.

Each output file contains a sequence of states. The first state in the sequence is the initial configuration of the puzzle. The last state is a goal state. There is one empty line between any two consecutive states.

Here are examples of [an input file \(https://q.utoronto.ca/courses/293717/files/24332586?wrap=1\)](https://q.utoronto.ca/courses/293717/files/24332586?wrap=1)  [\(https://q.utoronto.ca/courses/293717/files/24332586/download?download_frd=1\)](https://q.utoronto.ca/courses/293717/files/24332586/download?download_frd=1) and [an output file \(https://q.utoronto.ca/courses/293717/files/24332588?wrap=1\)](https://q.utoronto.ca/courses/293717/files/24332588?wrap=1)  [\(https://q.utoronto.ca/courses/293717/files/24332588/download?download_frd=1\)](https://q.utoronto.ca/courses/293717/files/24332588/download?download_frd=1) . The output file is an optimal solution for the puzzle found by A* search.

Two Heuristic Functions for A* Search

A* search requires an admissible heuristic function to find an optimal solution. You will implement a basic heuristic function (the Manhattan distance heuristic) as part of your A* search implementation. In addition, you will propose an advanced heuristic function that is better than the Manhattan distance heuristic.

(1) The Manhattan distance heuristic function

The simplest admissible heuristic function for this problem is the Manhattan distance heuristic function. Suppose that we relax this problem such that the pieces can overlap. Given this relaxation, we can solve the puzzle by moving the 2x2 piece over the other pieces until it is at 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.

(2) Your advanced heuristic function

Your next task is to propose an advanced heuristic function better than the Manhattan distance heuristic function. Your advanced heuristic function should satisfy the two requirements below.

1. It should be admissible.
2. It should dominate the Manhattan distance heuristic.

Submit a one-page file named **advanced.pdf**. (We will only grade the first page if your file is longer than one page.) In this file, answer the three questions below.

1. Describe how one can calculate the advanced heuristic value for any state of the puzzle.
2. Why is your advanced heuristic admissible?
3. Why does your advanced heuristic dominate the Manhattan distance heuristic?

Submission and Mark Breakdown

You should submit two files on MarkUs.

- **hrd.py**: This file contains your Python program that executes DFS or A* search with the Manhattan distance heuristic on a Huarongdao puzzle.
- **advanced.pdf**: This file contains a one-page write-up answering the three questions regarding your advanced heuristic function.

The mark breakdown is as follows:

- Depth-first search: **30%**
- A* search with Manhattan distance heuristic: **60%**
- Advanced heuristic description: **10%**

We will test your program on 15 puzzles. There are 5 easy puzzles, 5 medium puzzles, and 5 hard puzzles. For each puzzle and search algorithm combination, your program must terminate in under 2 minutes ~~5 minutes~~ to earn marks. This time limit is quite generous. For your information, our A* search implementation has the following runtimes:

- ≤ 5 seconds for any easy puzzle
- ≤ 15 seconds for any medium puzzle
- ≤ 25 seconds for any hard puzzle

Optimize your program and test your program on the **teach.cs** server to ensure that it terminates under the time limit.

We will provide 3 of the 15 puzzles to you on MarkUs. We highly recommend that you test the correctness and efficiency of your program using these 3 puzzles. In addition, we recommend you create other test cases using the other puzzle configurations on the Wikipedia page.

Starter File:

We have provided some starter code on MarkUs in a file named **hrd_starter.py**. This file is provided as is, and we do NOT guarantee that the code is correct or bug-free. You can feel free to copy or modify the starter code in any way you like.

The file has the following components:

- A Piece class
- A Board class
 - A function to convert the board to a 2d grid representation.
 - A function to display the board.
- A State class

- The `read_from_file` function reads a puzzle from a file and returns a board.
- The main function contains code to read in command line arguments.

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 **you write the program incrementally** --- start by **writing helper functions and testing them individually**.

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

Write a few classes to represent the elements of the puzzle.

- **Piece:** 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.
- **Board:** 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.
 - You may want to write a function to print the board for debugging.
 - 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.
- **State:** 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.

- **Read in a puzzle:** Write a function to take an input file and return a board containing a set of pieces.

Write some useful helper functions before writing the main search functions.

- **Goal test:** Write a function which takes a state and returns true if and only if the state is a goal.
- **Heuristic function:** Write a function that takes a state and returns the state's heuristic value (or the h value).
- **Generate successors:** 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.** We strongly suggest you **break down this function into several helper functions**.
 - Start by writing an outline for the function. Assume that there exists a helper function to perform each step. Give each helper function an intuitive name to describe its purpose. Then proceed to write each helper function. This top-down approach will force you to think about the overall logic before writing code.
- **Get solution:** Given a goal state, backtrack through the parent state references until the initial state. Return a sequence of states from the initial state to the goal state.

Write the main search functions.

- **DFS:** Write a function that takes an initial state and returns the first solution found by DFS.
 - Start with the pseudocode for the Generic Search Algorithm discussed in class.
 - **Frontier:** We recommend using **heapq** as the frontier. You can use **heappush** and **heappop** to modify the frontier.
 - Although we did not explicitly mention pruning in the requirements, you do need to implement pruning. Without pruning, your program is unlikely to terminate in a reasonable time. Follow the pseudocode for the Generic Search Algorithm with Multi-Path Pruning discussed in class.
 - Be careful with choosing a data structure for the explored set. Hints: Lists are faster than sets when you want to iterate over the values. Sets are faster than lists when you want to check whether an element is present.
- **A* Search:** Write a function that takes an initial state and returns the optimal solution found by A* search.
 - See the above tips for implementing the function for DFS.