National University of Singapore
School of Computing
CS3243 Introduction to Artificial Intelligence

Project 1: An Evaluation of Uninformed and Informed Search Algorithms on the *k*-puzzle Problem

Issued: 15 May 2020 Due: 29 May 2020, 2359hrs

Objectives

The aims of this project are to:

1. Become familiar with the implementation of uninformed and informed search algorithms through their application on a simple problem.
2. Analyse the correctness and complexity of the aforementioned search algorithms.
3. Design experiments to further empirically evaluate the aforementioned search algorithms.
4. Discuss the effectiveness of the implemented algorithms on the *k*-puzzle problem.

This project is worth 10% of your module grade.

Introduction: The *k*-puzzle Problem

The *k*-puzzle problem is a generalized version of the 8-puzzle problem, which is described in detail in the **AIMA text**.

2	3	6
1	5	8
4	7	

Figure 1: An example of an initial state for the 8-puzzle problem.

In the 8-puzzle version, you are given a 3×3 grid, where all but one grid cell contains a unique number between 1 and 8 (inclusive), and the last cell is left empty. An example of this is depicted in Figure 1.

The goal is to move the cells such that all the cells are ordered. More specifically, for the 8-puzzle to be solved, all the cells must be identical to the positions depicted in Figure 2.

1	2	3
4	5	6
7	8	

Figure 2: An example of an initial state for the 8-puzzle problem.

The only valid moves in the *k*-puzzle problem correspond to the movement of a numbered cell to a neighbouring cell that is empty. Thus, given the example in Figure 1, a solution may be achieved by taking the following sequence of valid moves:

$$8\downarrow \quad 6\downarrow \quad 3\rightarrow \quad 2\rightarrow \quad 1\uparrow \quad 4\uparrow \quad 7\leftarrow \quad 8\leftarrow$$

For the generic *k*-puzzle, you are to assume that the grid has dimensions $n \times n$, where $k = n^2 - 1$, $n \geq 2$ and $n \in \mathbb{Z}$.

General Project Requirements

The general project requirements are as follows:

- Group size: **3-4 students**
- Submission Deadline: **29 May 2020** (Friday), **2359 hours** (local time)
- Submission Platform: **LumiNUS** > **CS3243** > **Project Submission Folder**
- Submission Format: One standard (non-encrypted) **zip file**[1]

As to the specific project requirements, you must complete and submit the following:

1. An implementation of **either BFS or UCS** algorithm.
2. An implementation of **A* search** algorithm with Manhattan Distance heuristic.
3. An implementation of **A* search** algorithm with one other heuristic that dominates the Manhattan Distance heuristic (add some comments to briefly explain why it dominates).
4. A `.txt` file detailing the time taken and number of moves for all 3 files on its respective public test cases.

[1]Note that it is the responsibility of the students in the project group to ensure that this file may be accessed by conventional means.

Late Submissions

For projects submitted beyond the submission deadline, there will be a 20% penalty for submitting after the deadline and within 24 hours, 50% penalty for submitting after 24 hours past the deadline but within 48 hours, and 100% penalty for submitting more than 48 hours after the deadline For example, if you submit the project 30 hours after the deadline, and obtain 92%, a 50% penalty applies and you will only be awarded 46%.

Submission Specifications

Your submissions should contain the following:

- **THREE** Python code files (one for BFS or UCS, two for A* search with one heuristic each).
- One `.txt` file detailing the time taken and number of moves for all 3 files on its respective public test cases. This is used merely as a guide for us when grading. You should be using the results obtained in the `_tests.txt` file from the CodePost autograder (refer to end of file). In case of major discrepancies between our experiments and your declared timing, we will be checking CodePost or ask you to reproduce the timings for us. If the group is unable to obtain the declared timings within a reasonable margin, 0 marks will be awarded for the corresponding algorithm, regardless of whether it has been implemented correctly.

The files mentioned above should be named:

- **`CS3243_P1_XX_1.py`** - for your BFS or UCS implementation
- **`CS3243_P1_XX_2.py`** - for your A* search implementation using the Manhattan Distance heuristic
- **`CS3243_P1_XX_3.py`** - for your A* search implementation using the other heuristic
- **`CS3243_P1_XX_Timings.txt`**

The zip file should be named **`CS3243_P1_XX.zip`**. For all the filenames listed, **`XX`** refers to your assigned group number. For example, **`CS3243_P1_03.zip`**. Note that any single-digit group numbers should be named with **`03`** and not **`3`**.

Failure to adhere to these naming conventions will result in loss of marks.

Input

The input is provided in a text file, which will contain n lines, with n integers on each line. Each such integer included will correspond to one integer from the sequence 0 to $(n^2 - 1)$, with `0` representing the empty cell. This will correspond to the initial state of the *k*-puzzle. For example, the initial state given in Figure 1 would be encoded in the text file as follows:

```
2 3 6
1 5 8
4 7 0
```

Output

The output of your code should correspond to a List containing the strings: `'LEFT'`, `'RIGHT'`, `'UP'`, `'DOWN'`, and `'UNSOLVABLE'`. For example, the output specified as the solution to the initial state given in Figure 1, would be: `['DOWN', 'DOWN', 'RIGHT', 'RIGHT', 'UP', 'UP', 'LEFT', 'LEFT']`.

If the given puzzle (input) is not solvable, your output should be `['UNSOLVABLE']`. Else, you should output the actions of each step required to solve the puzzle, where each step here corresponds to the movement of the non-empty cell (i.e., the non-zero cell). You are required to properly implement this, i.e. your program should not declare a puzzle as unsolvable after a set number of moves if the puzzle is indeed solvable.

You must ensure that all moves performed are valid. For example, you may not specify `'DOWN'` if the empty cell (i.e., `0` cell) is on the top row. Further, the sequence of moves specified must bring the puzzle to the goal state, with the empty cell in the bottom-right corner.

Code

Please use Python 2.7 (the default Python version on SoC's Sunfire) to do this assignment. The template has been provided to you. You may import Python Standard Libraries if necessary. However, you may not use any external libraries. In addition, please note the following:

- You may only change the code inside the Puzzle class for each search implementation file, but you can create new classes within the provided template file.
- You cannot use additional python files, you cannot modify the `__main__` method, and you cannot use global variables.
- You are required to implement ONE uninformed (BFS or UCS) and ONE informed (A*) search algorithm, with TWO heuristics (Manhatten Distance and one other dominating it) for the latter.

- Your code must be executable; if your program cannot be executed (at least on CodePost), you will get 0.
- Your code should be correct. It must follow the given input and output specifications and pass all test cases in reasonable time (refer to the guide at the end of this report for what constitutes reasonable).

Testing on our Grader

Note that running the code on your own and running on our grader may lead to different output (e.g. using global variables would work when running on your own, but not on our grader). It is imperative that you test out your code on our grader. If your code does not work on our grader, you will get a 0 for the corresponding section. Thus, we will be utilizing a platform that allows you view the output by running your code on our grader.

Register for the CS3243 course on CodePost.io to gain access to the autograder.

Logging in for the first time - Invite link: `https://codepost.io/signup/join?code=J3CJO2R1I2` (Remember to check your spam/junk mail for the activation email (it'll most likely go there). Contact the course staff if you don't receive it after 30 minutes.)

Subsequent access: `https://codepost.io/student/CS3243%20Introduction%20to%20AI/AY19%2F20%20Special%20Term%20I/`

1. Rename your file to `CS3243_P1_01_1.py` for all testing purposes only on CodePost (This is IMPORTANT! Otherwise it will not work)
2. For the section you wish to test on our grader, click on "Upload assignment" and upload
3. Refresh the page
4. Select "View feedback" after the submission have been processed
5. You may check your output for each test case of the corresponding section (number of moves, time taken) under `_tests.txt` option (this is also the default view)

You may submit your files as many times as you like. Note that the marks awarded by the autograder is not at all reflective of the actual marks you will get. We are merely using the platform as a means for you to be able to view the number of moves and timings when tested on our grader.

Marking Rubrics

Component	Requirements (Marks Allocated)	Total Marks
Uninformed Search Algorithm (BFS or UCS)	• Implement the BFS/UCS algorithm correctly, and works for all $n \times n$ puzzles, not just 3×3 (1) • Pass all 4 public test cases within time threshold (0.5) • Pass all 4 public test cases within moves threshold (0.5) • Pass all 2 private test cases within time threshold (0.5) • Pass all 2 private test cases within moves threshold (0.5)	3
A* Search Algorithm with Manhattan Distance heuristic	• Implement the A* Search algorithm with Manhattan Distance correctly, and works for all $n \times n$ puzzles (1) • Pass all 4 public test cases within time threshold (0.5) • Pass all 4 public test cases within moves threshold (0.5) • Pass all 2 private test cases within time threshold (0.5) • Pass all 2 private test cases within moves threshold (0.5)	3
A* Search Algorithm with one other heuristic dominating Manhattan Distance	• Implement the A* Search algorithm with chosen heuristic (and dominates Manhattan Distance heuristic) correctly, and works for all $n \times n$ puzzles (1) • Pass all 4 public test cases within time threshold (0.5) • Pass all 4 public test cases within moves threshold (0.5) • Pass all 2 private test cases within time threshold (0.5) • Pass all 2 private test cases within moves threshold (0.5) • Top 50% of the cohort in terms of time (0.5) • Top 50% of the cohort in terms of num of moves (0.5)	4

Test Cases and Thresholds

The first 4 cases in each component are public. The next 2 are private, but you will be able to view the timings on CodePost. Here are the timings for your reference.

Component	Test Cases	Thresholds
(1) Uninformed Search Algorithm (BFS or UCS)	• `input_1` (3×3): • `input_2` (3×3): • `input_3` (3×3): • `input_4` (3×3): • `input_5` (3×3): • `input_6` (3×3):	• UNSOLVABLE (& should fail in 1 move) • Time: $\leq 0.25s$, Num of Moves: ≤ 20 • Time: $\leq 0.42s$, Num of Moves: ≤ 20 • Time: $\leq 0.80s$, Num of Moves: ≤ 22 • Time: $\leq 3.2s$, Num of Moves: ≤ 25 • Time: $\leq 7.2s$, Num of Moves: ≤ 35
(2) A* Search Algorithm with Manhattan Distance heuristic	• `input_1` (4×4): • `input_2` (4×4): • `input_3` (4×4): • `input_4` (4×4): • `input_5` (4×4): • `input_6` (4×4):	• Time: $\leq 0.02s$, Num of Moves: ≤ 24 • Time: $\leq 3.5s$, Num of Moves: ≤ 28 • Time: $\leq 0.8s$, Num of Moves: ≤ 30 • Time: $\leq 12s$, Num of Moves: ≤ 30 • Time: $\leq 2s$, Num of Moves: ≤ 28 • Time: $\leq 15s$, Num of Moves: ≤ 30
(3) A* Search Algorithm with one other heuristic dominating Manhattan Distance	• `input_1` (4×4): • `input_2` (4×4): • `input_3` (4×4): • `input_4` (4×4): • `input_5` (4×4): • `input_6` (4×4):	• Time: $\leq 0.28s$, Num of Moves: ≤ 30 • Time: $\leq 0.015s$, Num of Moves: ≤ 25 • Time: $\leq 0.08s$, Num of Moves: ≤ 28 • Time: $\leq 0.85s$, Num of Moves: ≤ 32 • Time: $\leq 0.04s$, Num of Moves: ≤ 20 • Time: $\leq 18s$, Num of Moves: ≤ 45

As a guide, the above timings are obtained on Sunfire. Sunfire generally is much slower (approx. 3.5-4$\times$) compared CodePost and your local machine (MacBook Pro 2.8 GHz Intel Core i7 as a reference). The reported timings in the `.txt` file should be taken from CodePost. But for consistency, we will be running your code and compare timings on Sunfire. Obviously, as you do not have access to the private test cases to run on Sunfire, you may take the timings on CodePost as a rough estimate (compare the difference in timings for the public test cases and you should have an idea about whether your private test cases timings are way off or not). So do not compare the exact timings above with that on CodePost. In terms of the number of moves made, it should not differ by machine.