CS 470 Programming Assignment: The 8-puzzle

Honor code note: You are welcome to discuss the assignment with other students, but you must individually and independently write the code and other deliverables that you submit for the assignment. Any copying of code from outside sources or other students will be considered an honor code violation.

This assignment involves solving 8-puzzles. You should first download the `PA8.zip` file from Canvas and unzip in on your computer. The provided code has been tested with Python 3, but should work with minor changes on other versions of Python.

The folder you are provided contains:

- `PA_eight_Handout.pdf` - This file that you are reading
- `eight.py` - This is the main file that you will be editing. To run the code you should type:

  ```
  python eight.py INPUT_FILE.txt OPTIONS
  ```

  from the command line, where `INPUT_FILE.txt` is one of the provided puzzle files, covered next. OPTIONS will specify the options that are possible and can be specified as follows:

  - Search strategy: This is specified with the -s option. The options are bfs (best-first search) or ids (iterative-deepening search). The default, if you don't specify, is ids.
  - Heuristic function: This is specified with the -f option. The options are top (tiles out of place), torc (tiles out of row column), and md (manhattan distance to goal). You will see more details about these functions later on.
  - Evaluation function type: This is specified with the -t option. The options are g (greedy), u (uniform cost), and a (A*).

  As an example, to run the best first search greedy code with the manhattan distance heuristic on the easy.txt file, you would type

  ```
  python eight.py easy.txt -s bfs -f md -t g
  ```

  This allows us to easily run the different parts of the code for testing.

- `easy.txt, medium.txt, hard.txt, worst.txt, random.txt` - These files provide mixed-up 8-puzzles for you to solve. You will be reporting your code's performance on each file. The mixed up puzzles were created using breadth-first search (see `create_puzzle.py` if you are interested) to target specific solution lengths, as follows:
  - `easy.txt` - puzzles with solutions of length 7
  - `medium.txt` - puzzles with solutions of length 15
  - `hard.txt` - puzzles with solutions of length 21
  - `worst.txt` - puzzles with solutions of length 30-31
  - `random.txt` - puzzles with solutions of length between 1 and 30 (randomly chosen)
- `create_puzzle.py` - A program that can be used to create more puzzle files. You shouldn't need this, but it is provided if you are interested.

In all of what follows you can assume the following goal state for the 8-puzzle:

	1	2
3	4	5
6	7	8

Part 1 [70 points]

The first part of this assignment is to compare different search algorithms and several heuristics for solving the 8-puzzle (iterative-deepening search, uniform-cost search, greedy best-first search, and A*search).

You have been provided starter code with a basic search framework implemented. In this task, you will be required to implement two heuristic functions and test how well these work with the various search algorithms. Starter code has been provided on Canvas.

- **Task 1.1** [**10 points**] Implement the goal test to determine if a puzzle's state matches the goal. This should return True or False. Look for the place in the code marked "TASK 1.1 BEGIN".
- **Task 1.2** [**5 points**] With the goal test implemented, the provided code should run successfully. The provided code already has iterative-deepening search (IDS) implemented. Run the code on the different puzzle files and report the average performance on number of puzzles from each file. (The default is set to 5, but you can change it. The IDS search code runs by default without any other options specified, so you shouldn't need any options specified, other than the puzzle file to solve) Note: this will be very slow for the `hard.txt` and even slower for the `worst.txt`. You can omit these if they are taking too long on your machine, but try to get timing and expansion information for at least 1 puzzle from each file.
- **Task 1.3** [**5 points**] This task requires you to look over the provided search code and modify the f-value calculations in the `SearchNode` class `compute_f_value` function. Look for the place in the code marked "TASK

1.3 BEGIN". When necessary, use the `heuristic` function result, which is provided and by default computes the tiles out of place heuristic, although it will run whichever heuristic function was specified in the options. Once you have implemented the f-value calculation for the following three algorithms, complete the following tasks.

1. Uniform-cost search (Command line option: -t u): Run the code on each of the provided files and report the average performance on 40 puzzles from each file.
2. Greedy best-first search (Command line option: -t g): Run the code on each of the provided files and report the average performance on 40 puzzles from each file.
3. A*search (Command line option: -t a): Run the code on each of the provided files and report the average performance on 40 puzzles from each file.

Additionally, make sure run the code with option (-s bfs) to run the best first search code that actually utilizes these different algorithms.

- **Task 1.4 [30 points]** This task requires you to implement two new heuristic functions, the and `tiles_out_of_row_column` and `manhattan_distance_to_goal` functions.

  The three heuristic functions that you will test are as follows:

  - **Number of tiles out of place:** A simple count of the number of tiles not in their goal position. The code for this heuristic has been provided to you as the function `tiles_out_place` in `pa1.py`.
  - **Number of tiles out of row and column:** This heuristic simply counts the number of tiles that are not in their goal row and the number of tiles that are not in their goal column and returns the sum of these two numbers. This way a tile can contribute 0 (right row and column already), 1 (right row or column, but wrong on other) or 2 (wrong row **and** wrong column) to the heuristic value. You should be able to convince yourself that this is an admissible heuristic. You must write the code for this heuristic function, but the function definition is again provided as `tiles_out_of_row_column` in `pa1.py`.
  - **Manhattan distance:** The sum of the (horizontal and vertical) distances of each tile from its goal position, as discussed in class. You will be required to write this code, but the function definition is provided as `manhattan_distance_to_goal` in `pa1.py`.

  - **Task 1.4.1 [15 points]** You should fill out the `tiles_out_of_row_column` function. The place where you should write your code is clearly marked. You are free to write additional helper functions to make your code more readable.
  - **Task 1.4.2 [15 points]** You should fill out the `manhattan_distance_to_goal` function. The place where you should write your code is clearly marked. You are free to write additional helper functions to make your code more readable.

- **Task 1.5 [20 points]** Your task for this is evaluate each of the three heuristics and each of the three algorithms (Uniform-cost, greedy best-first, A*) on some of the input files provided. You are free to use whichever ones you want, but your results should effectively communicate how well each of the heuristics and search algorithms work. For example, you might want to plot out the number of nodes expanded as a function of solution length for each algorithm-heuristic combination. These results should be compared to your results for iterative-deepening as well.

You are required to hand in the following for Task 1:

1. Your code with the required parts implemented or changed.
2. Your evaluation of the heuristics and search algorithms from Tasks 1.1 and 1.3. Included whatever graphs or tables of results you produced.
3. At least a half-page write-up describing what you learned from these experiments.

Part 2 [30 points]

The second part of the assignment is a bit more open-ended: you should modify the provided iterative-deepening search code to implement the IDA*algorithm and use it to solve the 8 puzzle. To do this you will have to modify the existing code as you see fit. Make sure to specify the option to use the iterative-deepening code again.

You are required to hand in the following for part 2:

1. A commented copy of the code you wrote to modify Iterative-deepening and implement IDA*.
2. Using the same techniques as in Part 1, an analysis of how well your IDA*implementation worked with the different heuristics.
3. At least a half-page write-up describing what you learned from implementing IDA*.

1 Extra credit [25 points]

- [5 points] **Implement additional heuristic: Manhattan distance with linear conflicts:** This is the Manhattan distance heuristic with the following enhancement: if tiles are in the same correct row (or column) but must pass each other to get to their goal position, then one must move to get out of the other's way. When two tiles are in this situation we can add 2 to the Manhattan distance heuristic for each such occurrence. The reason you can add 2 while preserving admissibility is that the Manhattan distance heuristic does not account for the fact that one of the tiles must move out of the way to allow the other to get past. When three tiles are in the right row or column but in the incorrect order than it can be slightly more complicated to preserve admissibility. You can figure out the different cases.

- **[20 points] Solving the 15-puzzle** You will get up to 20 points extra credit if your implementation of IDA*also solves the 15-puzzle. The 15-puzzle is much more difficult to solve than the 8-puzzle, but IDA*can solve it easily if implemented efficiently. (By contrast, A*runs out of memory on most examples of the 15-puzzle, especially those that require long solution paths.)

  Note: the provided code only supports the 8-puzzle and so you must first modify it to handle the 15-puzzle. I recommend making a new copy of the code and working with it for the 15-puzzle, so you don't mess up the code for the 8-puzzle. You can modify and use `create_puzzle.py` to create new 15-puzzles for you to solve to test your algorithm (you might have to use hexadecimal notation for the tile numbers, as the current code assumes all tiles only take one character to represent). Since it is extra credit you are kind of on your own, but if you have any questions, feel free to ask Dr. Archibald.

  If you run your implementation of IDA*on the 15-puzzle, you should consider including the following enhancements in your code to make it more efficient.

  - **Parent checking:** You can make your implementation of IDA*more efficient by not generating a successor node that is the same as the parent of a state. Although IDA*will work correctly without this enhancement, adding this check will eliminate some duplicates from your search tree.
  - **Node ordering:** Your implementation of IDA*will run much faster if you expand the children of each node in increasing order of the heuristic estimate. This technique is called node ordering. Because IDA*stops as soon as it finds a solution, this enhancement allows it to visit many fewer nodes during the last iteration. It will not make earlier iterations any faster, but most of the running time of IDA*is spent in the last iteration.

  To pass off the extra credit turn in any code you wrote for it as well as evidence that it worked and a short paragraph write-up of what you learned.