15780: GRADUATE AI (SPRING 2017)

Homework 0: A* Search
(Solutions)

Release: January 18, 2017,
Due: January 27, 2017, 11:59pm

1 Programming Component [50 points]

We will use scientific Python for the implementation portions in this course. If you have not used Python before, we recommend downloading the Anaconda distribution (`https://www.continuum.io/downloads`) and looking through introductory resources like Google's Python Class (`https://developers.google.com/edu/python/`) and the Python Beginner's Guide (`https://wiki.python.org/moin/BeginnersGuide`). If you have not used scientific Python before, we recommend following introductions to NumPy (`http://www.numpy.org/`) and matplotlib (`http://matplotlib.org/`).

All Python code submissions in this course use Python 3.

For this problem, you will implement the A^* graph search algorithm to discover the series of moves that transform a moving tile puzzle from an initial state into a desired goal state. For example, for a 3x3 puzzle, given the following initial state:

1	2	3
4		6
7	5	8

and the following goal:

1	2	3
4	5	6
7	8	

your program should return something like ["down", "right"], i.e., to solve this puzzle the blank has to be first moved down and then moved to the right.

We have provided a `search.py` Python module for you to get started with. When you are done, you will submit your completed module to Autolab for automatic grading. Do not rename the file or change the function names because our grader will import the module and functions by name. You are welcome to introduce new auxiliary functions and call them from the main functions we provide.

You should be able to execute the `search.py` we have provided without any modifications as a starting point. `search.py` contains a `main` function that shows expected outputs of every function we will grade.

1.1 Implementing Heuristics

The board state in Python will be a flattened tuple of the two-dimensional grid in row-major order where the blank tile is represented as 0. The example above is represented as (1, 2, 3, 4, 0, 6, 7, 5, 8). You can internally change the state to a format that's easier to work with if you prefer.

You should first implement the following two heuristics in the stubbed `heuristic_misplaced` and `heuristic_manhattan` functions we have provided.

1. **The number of misplaced tiles.** In the example above, there are two misplaced tiles, 5 and 8. The blank space is not tile and should not be included in your misplaced tile count.

2. **The sum of the Manhattan distances from the misplaced tiles to their correct positions.** In the example above, the distance from the misplaced tiles 5 and 6 to their correct positions are both 1, so the summed Manhattan distances is 2.

1.2 Implementing A^*

Next you will implement the A^* *graph search* algorithm in the stubbed `astar` function we have provided to find the shortest path using the heuristics above. Your function should return a string representing the moves needed to reach the goal and a list of states in the order they were visited. Use the characters 'r', 'l', 'u', and 'd' for 'right', 'left', 'up', 'nd 'down' directions, respectively. In the example above, your function should return the string 'dr'. If the grid is not valid, return `None` for the optimal path and the order of states visited.

You should try your program on a number of puzzles with different initial states. Because the goal state cannot be achieved from all possible states generated by randomly placing the tiles on the board, you should write a function that shuffles a puzzle from the goal state to an initial state by repeatedly moving the blank to a position randomly chosen from the possible moves. The depth of the solution for your shuffled puzzle will be no greater than the number of times the blank is moved.

Because states can repeat, you may also want to maintain a separate list of all explored states, and only add nodes to the list if they have not already been explored.

A^* is non-deterministic when selecting what node to explore next when they all have the same f value. To make A^* deterministic for grading, **we require that you break ties by selecting the lexicographically first node** based on the flattened state. For example, the flattened state `(1, 2, 3, 4, 0, 6, 7, 5, 8)` comes lexicographically before `(1, 2, 3, 4, 0, 6, 7, 8, 5)`.

We recommend that you use the `heapdict` package as a priority queue to find elements with minimum cost. The `heapdict` package is included by default in Anaconda distributions and the project page is available at `https://github.com/DanielStutzbach/heapdict`. To break ties, your should use a tuple with (f value, state, ¡other information you want to store¿) as the value in your heapdict.

2 Written Component [50 points]

Heuristics for n-puzzle

In class (lecture 2), we examined the 8-puzzle game and two heuristics. In this question, we generalize the puzzle and consider a different heuristic.

The 8-puzzle game is a single-player board game which consists of a 3×3 board with 8 tiles and 1 blank slot. A tile can move horizontally or vertically to its adjacent blank slot (if it neighbors it). The objective of the game is to start from a given arrangement of tiles and move tiles to achieve a given goal arrangement. Now, consider the n-puzzle game which consists of an $n \times n$ board with $n^2 - 1$ tiles and 1 blank slot. The rules for n-puzzle are exactly the same as those for 8-puzzle. The following questions are about the n-puzzle game.

Here, we discuss some heuristics to be used with A* graph search. Recall that heuristic $h_1(\cdot)$ returns the number of tiles that are in the wrong position and $h_2(\cdot)$ returns the sum of Manhattan distances of tiles from their goal positions. We introduce a third heuristic, $h_3(\cdot)$, that is the *minimum* number of moves necessary to get to the goal state if each action could move any tile to the blank slot. This is another *relaxed problem* heuristic.

1. (15 points) Prove that h_3 is consistent.

   Solution. Let x and y represent any two states of the problem. Define $H(x, y)$ to be the minimum number of moves necessary to go from x to y where each action could move any tile to the blanks slot. Note that this is the same as value of $h_3(x)$ when y is the goal state.

   Sliding a tile is the same as switching a tile with its adjacent blank space (if it neighbors the blank space). Since H allows any tile to be switched with the blank slot regardless of where it is located, H is a relaxation of the n-puzzle game. So, $H(x, y) \leq \mathrm{c}(x, y)$.

   Now consider any x and y. One way to go from x to t using the moves defined by H (and h_3) is to first use $H(x, y)$ moves to go from x to y, and then use $h_3(y)$ moves to go from y to t. Using the inequality from the previous paragraph, we have

   $$h_3(x) \leq h_3(y) + H(x, y) \leq h_3(y) + \mathrm{c}(x, y)$$

   So, h_3 is consistent. □

2. (5 points) Prove or disprove: h_3 dominates h_1.

   Solution. Yes. h_3 can only place a tile on a blank slot without changing the order of the other tiles. So, for any misplaced tile, h_3 takes at least one move. In fact, if the blank slot is in its correct position, then h_3 makes two moves to take a tile to its correct location. Since, h_1 is the number of misplaced tiles, h_3 is at least as large as h_1, so it dominates h_1. □

3. (5 points) Prove or disprove: h_3 dominates h_2.

   Solution. No. Consider a goal state where the blank space is in the bottom right corner and the tiles increase from left to right and top to bottom. Now consider a start state where the blank space and tile 1 are switched. h_3 can move tile 1 to its location in 1 move, whereas h_2 computes the Manhattan distance between the current location of tile 1 and its desired location, which is 4. So h_3 does not dominate $h1$. □

4. (5 points) Prove or disprove: the heuristic $h = \max(h_2, h_3)$ is consistent.

Solution. We shall show that any heuristic defined as the maximum of two other consistent heuristics is consistent. We know by definition that

$$
\begin{aligned}
h_1(s) &\leq c(s, s') + h_1(s') \\
h_2(s) &\leq c(s, s') + h_2(s')
\end{aligned}
$$

and would like to show that the heuristic function

$$h(s) = \max(h_1(s), h_2(s))$$

is also consistent. WLOG, we can assume that $h_1(s) \geq h_2(s)$ by symmetry. Then, we have two cases. Case 1: $h_1(s') \geq h_2(s')$. Then we have

$$
\begin{aligned}
c(s, s') + h(s') &= c(s, s') + h_1(s') \\
&\geq h_1(s) = h(s) \\
\rightarrow h(s) &\leq c(s, s') + h(s').
\end{aligned}
$$

Case 2: $h_1(s') < h_2(s')$. Then we have

$$
\begin{aligned}
c(s, s') + h(s') &= c(s, s') + h_2(s') \\
&> c(s, s') + h_1(s') \\
&\geq h_1(s) \\
&= h(s) \\
\rightarrow h(s) &\leq c(s, s') + h(s').
\end{aligned}
$$

Therefore, this is a consistent heuristic function. □

5. (20 points) An important feature of any heuristic is that it can be computed efficiently. Give a polynomial-time algorithm in the number of tiles to compute h_3 for a given state. Prove its correctness and running time.

Solution. If the blank space is in its final position, then switch any misplaced tile with the blank space. If the blank space is not in its final position, switch it with the tile whose final position is currently occupied by the blank space.

Runtime: This algorithm switches each misplaced tile at most twice, once into the blank space and once into its final position. So, if there are k misplaced tiles, the algorithm takes $O(k)$ steps. Since the number of misplaced tiles is at most the total number of tiles, the algorithm is polynomial (in fact, linear) in the number of tiles, assuming we have data structures where finding the index of nodes and finding misplaced tiles can be done in constant time. If we are using more naïve data structures, the algorithm could take quadratic time in the number of tiles.

Correctness: Let G be a graph with one vertex for each tile and an additional vertex, $*$, for the blank space. For any two vertices v and v', there is an edge going from v to v' iff v's final position is now occupied by vertex v'. For any state of the board, this graph is made of loops and non-trivial cycles (length > 1).

Define the following potential function.

$$\phi(G) = \left(\text{\small length of the cycle that includes the blank space} - 1 \right) + \sum_{\substack{\text{Non-trivial cycles} \\ \text{without the blank}}} (\text{size of cycle } + 1)$$

If a given tile is switched with the blank space, then we must be in one of the following five situations:

(a) If $*$ is in its place and is switched with a misplaced tile x, then cycle $(z, x, y, \ldots, z)$ of length n and $(*, *)$ merge to cycle $(z, *, x, y, \ldots, z)$. Before merging the potential was $n + 1$, after merging it is $(n + 1) - 1 = n$, so the potential function decreases by 1.

(b) If $*$ is out of place and it is switched with its predecessor (the tile that has to be moved to the blank space's position), cycle $(x, *, y, \ldots, x)$ of length n splits to cycles (x, x) and $(y, \ldots, x, y)$. Before the split, the potential was $n-1$, after the split it is $(n-1)-1$, so the potential decreases by 1.

(c) If $*$ is in its place and is switched with another tile, x, that is also in its final position, then cycle $(x, *, x)$ is formed from loops (x, x) and $(*, *)$ and the potential function increases by 1.

(d) If $*$ is out of place and is switched with a vertex b from a different cycle, then two cycles $(x, *, y, \ldots, x)$ of length n_1, and $(a, b, c, \ldots, a)$ of length n_2 merge to form a new cycle of length $n_1 + n_2$, $(a, *, y, \ldots, x, b, c, \ldots, a)$. The potential function goes from $(n_1 - 1) + (n_2 + 1)$ to $(n_1 + n_2) - 1$. So, the potential decreases by 1.

(e) If $*$ is out of place and it is switched with vertex b that is the same cycle as $*$ but it is not its predecessor, cycle $(x, *, y, \ldots, a, b, c, \ldots, x)$ of length n splits to two cycles $(x, b, c, \ldots, x)$ and $(*, y, \ldots, a, *)$ with lengths $n_1 + n_2 = n$. The potential goes from n to $(n_1 + 1) + (n_2 - 1) = n$, so it stays the same.

A graph that is represented by the goal state has potential value of 0. Our algorithm only takes actions that result in situations (a) or (b), both of which decrease the potential by 1, which is the maximum decrease possible at every round. So, our algorithm finds the minimum number of moves possible. Therefore, its computes $h_3(\cdot)$, correctly. Note that the value of the potential function is the value of h_3.

□

3 Submitting to Autolab

Create a tar file containing your writeup for the first problem and the completed `search.py` module for the second problem. Make sure that your tar has these files at the root and not in a subdirectory. Use the following commands from a directory with your files to create a `handin.tgz` file for submission.

```
$ ls
search.py  writeup.pdf
$ tar cvzf handin.tgz writeup.pdf search.py
a writeup.pdf
```

```
a search.py
$ ls
handin.tgz  search.py  writeup.pdf
```