1 Introduction

Deep Thought, the city-sized computer tasked with the goal of coming up with the Answer to The Ultimate Question of Life, the Universe and Everything, has recently been having problems with its search algorithms. It appears to be afflicted with an unfortunate bug, which returns the same very puzzling answer to all of its questions! Once the pan-dimensional, hyper-intelligent beings who programmed Deep Thought realized they couldn’t figure out what to do, their first thought was to ask Deep Thought how to fix itself. Unfortunately, but somewhat predictably, Deep Thought merely returned the very same answer all over again, so instead they’ve decided to ask you to reimplement some of its core searching algorithms. The alien programmers have sent you Deep Thought’s code, disguised as a tile board solver file, so that you can help out.

In this assignment, you will be implementing breadth-first search, depth-first search, iterative deepening search, and A* search to solve tile board puzzles. First, however, you must prove your worth by answering a few simple questions.

2 Questions

1. In what situations would you use breadth-first search over depth-first search, and why? In what situations would you use depth-first search instead?

2. A* search requires a heuristic, and coming up with one demands domain knowledge plus a little thinking. You might therefore be tempted to cut corners and substitute in a trivial function instead.

   (a) What happens if you use a heuristic that returns zero everywhere?
   (b) What if it returned a fixed positive constant $c$? What if $c$ was larger than the maximum possible path cost?
(c) What happens if you always return a fixed negative constant?
(d) What happens your heuristic magically returns exactly the right answer?
(e) Which of the above cases are admissible, and what does that tell you?

3. Show that an inadmissible heuristic can lead to A* returning a suboptimal example, by giving an example search tree and heuristic where this occurs. Aim for as small a tree as possible, and explain your example clearly.

4. Give a detailed example (by drawing the states and labeling the edges with costs) where negative edge costs could lead to the following paradoxical result: a finite state space with an optimal solution that does not reach the goal in finite time. Be sure to explain why your example has this property.

5. Explain how you could modify iterative deepening search so that it finds the lowest cost solution, as opposed to the solution with the fewer number of steps. What information do you need? Prove that your algorithm finds the optimal solution.

3 The Tile Game

Now that you have proved yourself, the pan-dimensional, hyper-intelligent beings would very much like you to write some code for them. You are asked implement breadth-first search, depth-first search, iterative deepening search, and A* search to solve tile game puzzles.

The tile game is similar to the one you encountered in class. There is still a 3-by-3 grid, and each cell in the grid is still occupied by a number. Also, the win condition is the same; arrange the numbers in order from lowest in the upper left, to highest in the lower right.

However, the grid now has 9 numbers on it, instead of 8 numbers and an empty space. Additionally, movement is less restricted. In this game, you can swap any pair of adjacent numbers. The win condition—and your goal during search—is this board configuration:

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]

The code handout for the assignment contains several helper functions, which should be all that you need to implement search. For this assignment, you will represent the 3x3 board as a row-major list of lists.

Here’s a description of the helper functions you have been given:

- \texttt{boardSolver(board)}: takes in a board, runs BFS, DFS, and A* with a heuristic, records/prints the number of expansions per search, and then reports whether each search algorithm found the goal.
• generateBoard(): loads a random tile board and returns it.

• displayBoard(board): visualizes the given board.

• expandableStates(board): returns a list of boards that are all available expandable
states from a given board (i.e., it returns all unique valid tile swaps from board).

• isGoalState(board): returns a boolean value representing whether the input board
is in the goal configuration.

Please do not modify any helper function—we’ll be autograding your submissions by as-
suming the remainder of the code remains untouched, and if you change your copy of the
code—or change any global variables—your code won’t run in our tests!

One last thing: you must use the expandableStates method to get a list of all the reachable
states from a particular board; use it only to get a list of states to add to the horizon. We’ll
use our own implementation of that method to keep track of how many states you expanded
in your search.

Your task for the coding portion of this assignment is to fill in the following functions:

• BFS(board): Given an initial board state, perform Breadth First Search to find the
goal state. 

• DFS(board): As above, but use Depth First Search instead.

• IDS(board): As above, but use Iterative Deepening Search.

• Astar(board,heuristic): As above, use A* with an admissible heuristic.

• heuristic(board): Return an integer that represents an estimate of how close the
current board state is to the goal state. This heuristic must be both admissible and
consistent.

The top four functions should return a list of states, starting with board and ending with a
goal state. The heuristic function should return a single integer.

You are highly encouraged to read the remainder of the stencil code and understand how it
works. However, please only modify the functions listed above—they will be the
only parts of your code tested by our auto-graders. Do not modify any global
variables.
4  Install/Hand-in and Grading

You should hand in:

1. **A signed copy of the collaboration policy.**

2. A document containing:
   
   (a) Your answers for part 2.
   
   (b) A description of the heuristic that you have invented for the Tile Game, and a proof that it is admissible.
   
   (c) A graph of the found solution length for each of the four search types, averaged over 100 randomly generated boards.

3. Your Python code implementing the five functions listed above for the Tile Game.

To install the stencil code, make a cs1410 directory in your course directory: `[/CSlogin]/course/cs1410` and run the command:

```
        cp -r /course/cs1410/stencil/TileGame .
```

To hand in the assignment, go into `[/CSlogin]/course/cs1410/TileGame` and run:

```
        cs1410_handin TileGame
```

Please submit your printed answers (along with a signed copy of the collaboration policy) with your banner ID (not your name or CS login!) to the questions to our box #3 on the second floor of the CIT, directly outside the fishbowl.