COMS 4701 Artificial Intelligence

Homework 1: Coding - Search

Due Date: September 28, 2023

Please read carefully:

- You must name your file `puzzle.py`. Any submission that does not follow this naming will not be graded.

- A skeleton of each function has been provided to you in `puzzle.py`. You are expected to ONLY write code in the functions and blocks specified. In any case, DO NOT modify the function signatures, or any code that is not specified to be modifiable for any reason. This will be run by our autograder, so any unexpected modifications that make it malfunction will receive a zero. **You can, however, modify the main function** as necessary to make the program run as specified in the assignment Test cases have been provided in the main function in the skeleton code.

- Only Python 3.x versions will be graded.

- You may import the Numpy library if you would like, but the homework can be completed without it. No other library may be imported

- To receive points, make sure your code runs. We recommend using Spyder, Pycharm or Google Colab. They all allow you to download .py files. Be aware that if you write your code in some platforms like Codio and copy and paste it in a text file, there may be spurious characters in your code, and your code will not compile. **Always ensure that your .py compiles.**

- After your submission on Gradescope you will receive feedback in 5 minutes on whether your code has the proper filename, output format and execution time. Please address any issue and resubmit before the deadline.

In this assignment you will create an agent to solve the N-puzzle game. You will implement and compare several search algorithms, and collect some statistics related to their performances. Visit `mypuzzle.org/sliding` for the game's rules. Please read all sections carefully:
I. Introduction
II. Algorithm Review
III. What You Need To Submit
IV. What Your Program Outputs
V. Implementation and Testing
VI. Before You Finish

I. Introduction

The N-puzzle game consists of a board holding $N = m^2 - 1$ distinct movable tiles, plus one empty space. There is one tile for each number in the set $\{0, 1,..., m^2 - 1\}$. In this assignment, we will represent the blank space with the number 0 and focus on the m = 3 case (8-puzzle).

In this combinatorial search problem, the aim is to get from any initial board state to the configuration with all tiles arranged in ascending order $\{0, 1,..., m^2 - 1\}$ – this is your goal state. The search space is the set of all possible states reachable from the initial state. Each move consists of swapping the empty space with a component in one of the four directions {'Up', 'Down', 'Left', 'Right'}. Give each move a cost of one. Thus, the total cost of a path will be equal to the number of moves made.

II. Algorithm Review

Recall from lecture that search begins by visiting the root node of the search tree, given by the initial state. Three main events occur when visiting a node:

- First, we remove a node from the frontier set.

- Second, we check if this node matches the goal state.

- If not, we then expand the node. To expand a node, we generate all of its immediate successors and add them to the frontier, if they (i) are not yet already in the frontier, and (ii) have not been visited yet.

This describes the life cycle of a visit, and is the basic order of operations for search agents in this assignment–(1) remove, (2) check, and (3) expand. We will implement the assignment algorithms as described here. Please refer to lecture notes for further details, and review the lecture pseudo-code before you begin.

IMPORTANT: You may encounter implementations that attempt to short-circuit this order by performing the goal-check on successor nodes immediately upon expansion of a parent node. For example, Russell & Norvig's implementation of BFS does precisely this. Doing so may lead to edge-case gains in efficiency, but do not alter the general characteristics of complexity and optimality for each method. For simplicity and grading purposes in this assignment, do not make such modifications to algorithms learned in lecture.

III. What You Need To Submit

Your job in this assignment is to write puzzle.py, which solves any 8-puzzle board when given an arbitrary starting configuration. The program will be executed as follows:

$ python3 puzzle.py <method> <board>

The method argument will be one of the following. You must implement all three of them:
bfs (Breadth-First Search)
dfs (Depth-First Search)
ast (A-Star Search)

The board argument will be a comma-separated list of integers containing no spaces. For example, to use the bread-first search strategy to solve the input board given by the starting configuration {0,8,7,6,5,4,3,2,1}, the program will be executed like so (with no spaces between commas):

$ python3 puzzle.py bfs 0,8,7,6,5,4,3,2,1

IV. What Your Program Outputs

Your program will create and/or write to a file called output.txt, containing the following statistics:

path_to_goal: the sequence of moves taken to reach the goal
cost_of_path: the number of moves taken to reach the goal
nodes_expanded: the number of nodes that have been expanded
search_depth: the depth within the search tree when the goal node is found
max_search_depth: the maximum depth of the search tree in the lifetime of the algorithm
running_time: the total running time of the search instance, reported in seconds
max_ram_usage: the maximum RAM usage in the lifetime of the process as measured by the **ru_maxrss** attribute in the **resource** module, reported in megabytes

Example 1: Breadth-First Search
Suppose the program is executed for breadth-first search as follows:

This should result in the solution path:

$$parent = \begin{array}{|c|c|c|} \hline 1 & 2 & 5 \\ \hline 3 & 4 & \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|} \hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|} \hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|} \hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|} \hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|} \hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

The output file will contain exactly the following lines:

path_to_goal: ['Up', 'Left', 'Left']
cost_of_path: 3
nodes_expanded: 10
search_depth: 3
max_search_depth: 4
running_time: 0.00188088
max_ram_usage: 0.07812500

Example 2: Depth-First Search

Suppose the program is executed for depth-first search as follows:

$ python3 puzzle.py dfs 1,2,5,3,4,0,6,7,8

This should result in the solution path:

$$parent = \begin{array}{|c|c|c|} \hline 1 & 2 & 5 \\ \hline 3 & 4 & \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|} \hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|} \hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|} \hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|} \hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|} \hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

The output file will contain exactly the following lines:

path_to_goal: ['Up', 'Left', 'Left']
cost_of_path: 3
nodes_expanded: 181437
search_depth: 3
max_search_depth: 66125
running_time: 5.01608433
max_ram_usage: 4.23940217

More test cases are provided in **the FAQs**.

Note on Correctness

All variables, except running_time and max_ram_usage, have **one and only one** correct answer when running BFS and DFS. A* nodes_expanded might vary depending on implementation details. You'll be fine as long as your algorithm follows all specifications listed in these instructions.

As running_time and max_ram_usage values vary greatly depending on your machine and implementation details, there is no "correct" value to look for. They are for you to monitor time and space complexity of your code, which we highly recommend. A good way to check the correctness of your program is to walk through small examples by hand, like the ones above. Use the following piece of code to calculate max_ram_usage:

```
import resource
dfs_start_ram = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
dfs_ram = (resource.getrusage(resource.RUSAGE_SELF).ru_maxrss - dfs_start_ram)/(2**20)
```

Our grading script is working on a linux environment. For windows users, please change you max_ram_usage calculation code so it is linux compatible during submission. You can test you code on linux platforrm using services such as Google Colab.

V. Implementation and Testing

For your first programming project, we are providing hints and explicit instructions. Before posting a question on the discussion board, make sure your question is not already answered here or in the FAQs.

1. Implementation
You will implement the following three algorithms as demonstrated in lecture. In particular:

- **Breadth-First Search**. Use an explicit queue, as shown in lecture.

- **Depth-First Search**. Use an explicit stack, as shown in lecture.

- **A-Star Search**. Use a priority queue, as shown in lecture. For the choice of heuristic, use the Manhattan priority function; that is, the sum of the distances of the tiles from their goal positions. Note that the blanks space is not considered an actual tile here.
  Note: It may be helpful here to modify the PuzzleState object to allow for comparison between 2 states. You can explore pythons __lt__ method or other such methods for this purpose

2. Order of Visits
In this assignment, where an arbitrary choice must be made, we always **visit** child nodes in the **"UDLR"** order; that is, ['Up', 'Down', 'Left', 'Right'] in that exact order. Specifically:

- **Breadth-First Search**. Enqueue in UDLR order; de-queuing results in UDLR order.

- **Depth-First Search**. Push onto the stack in reverse-UDLR order; popping off results in UDLR order.

- **A-Star Search**. Since you are using a priority queue, what happens with duplicate keys? How do you ensure nodes are retrieved from the priority queue in the desired order?

3. Submission Test Cases
Run all three of your algorithms on the following test cases:

Test Case 1
$python3 puzzle.py bfs 3,1,2,0,4,5,6,7,8
$python3 puzzle.py dfs 3,1,2,0,4,5,6,7,8
$python3 puzzle.py ast 3,1,2,0,4,5,6,7,8

Test Case 2
$python3 puzzle.py bfs 1,2,5,3,4,0,6,7,8
$python3 puzzle.py dfs 1,2,5,3,4,0,6,7,8
$python3 puzzle.py ast 1,2,5,3,4,0,6,7,8

Make sure your code passes at least these test cases and follows our formatting exactly. The results of each test are assessed by 8 items: 7 are listed in **Section IV. What Your Program Outputs.** The last point

is for code that executes and produces any output at all. Each item is worth 0.75 point.

4. Grading and Stress Tests

We will grade your project by running **additional test cases** on your code. In particular, there will be five test cases in total, each tested on all three of your algorithms, for a total of **15** distinct tests. Similar to the submission test cases, each test will be graded by 8 items, for a total of 90 points. **Plus, we give 10 points for code completing all 15 test cases within 10 minutes**. If you implement your code with reasonable designs of data structures, your code will solve all 15 test cases within a minute in total. We will be using a wide variety of inputs to stress-test your algorithms to check for correctness of implementation. So, we recommend that you test your own code extensively.

Don't worry about checking for malformed input boards, including boards of non-square dimensions, other sizes, or unsolvable boards.

You will not be graded on the absolute values of your running time or RAM usage statistics. The values of these statistics can vary widely depending on the machine. **However, we recommend that you take advantage of them in testing your code.** Try batch-running your algorithms on various inputs, and plotting your results on a graph to learn more about the space and time complexity characteristics of your code. Just because an algorithm provides the correct path to goal does not mean it has been implemented correctly.

5. Tips on Getting Started

Begin by writing a class to represent the **state** of the game at a given turn, including parent and child nodes. We suggest writing a separate **solver** class to work with the state class. Feel free to experiment with your design, for example including a **board** class to represent the low-level physical configuration of the tiles, delegating the high-level functionality to the state class.

You will not be graded on your design, so you are at a liberty to choose among your favorite programming paradigms. Students have successfully completed this project using an entirely object-oriented approach, and others have done so with a purely functional approach. Your submission will receive full credit as long as your puzzle program outputs the correct information.

VI. Before You Finish

- **Make sure** your code passes at least the submission test cases.

- **Make sure** your algorithms generate the correct solution for an arbitrary solvable problem instance of 8-puzzle.

- **Make sure** your program always terminates without error, and in a reasonable amount of time. **You will receive zero points from the grader if your program fails to terminate. Running times of more than a minute or two may indicate a problem with your implementation.** If your implementation exceeds the time limit allocated (20 minutes for all test cases), your grade may be incomplete.

- **Make sure** your program output follows the specified format exactly. In particular, for the path to goal, use square brackets to surround the list of items, use single quotes around each item, and capitalize the first letter of each item. Round floating-point numbers to 8 places after the decimal. You will not receive proper credit from the grader if your format differs from the provided examples above.

COMS 4701 Puzzle FAQs

A. Search algorithm is perhaps one of the best learning materials for computational complexity and Python's idiosyncrasies. There are four dos and don'ts:

1. Don't store possibly large data member such as solution path in search tree node class. Instead, rethink what operation should be fast.

   Explanation: Storing a path from the root node in each node class achieves O(1) lookup time at the expense of O(n) creation time. For example, if the current state is visited after 60,000 intermediate states, the current state has to allocate a list of 60,000 elements, and each of the children states have to allocate a list of 60,001 elements. This would soon use up physical memory, and typically your machine's Operating System kills the search process.

   A key observation is that in the case of search algorithm, path lookup operation is executed just once after search finishes. Thus, the lookup operation is fine to be slow. You might consider another data structure having O(n) lookup time for solution path but requiring O(1) operations during search.

2. Don't be satisfied by just using list as frontier. Instead, design your Frontier class which works faster.

   Explanation: one major bottleneck of list, dequeue, or queue class in Python is that their membership testing operation is O(n). The membership testing speed is critical for search algorithm because that operation is executed for every child state. Coming up with using such list-like data structures is a good first step, but for using it with reasonable execution time, you might need one more trick.

   Note that pseudocode in lecture slides does not necessarily reflect implementation details (i.e. time and space). Rather, it conceptually shows the algorithm's inputs, processing orders, and outputs. One of your missions in this assignment is to make the "frontier" thing into a reality by using Python's "low-level" primitives.

3. Don't use O(n) operation when you have another faster way to do the same thing.

   Explanation: Roughly speaking, if you set one O(n) operation under your for neighbor in neighbors loop, your code will be highly likely to exceed grading time limit. In other words, it happens that your code executes drastically faster when you fix just one line of your code.

   For example, merging two sets and checking an element is in the merged set is an expensive operation, while checking an element is in one of the two sets are O(1) operation.

4. Don't use copy.deepcopy() for list. Instead, use list1 = [5,6]; list2 = list(list1) or list2 = list1[:].

   Explanation: copy.deepcopy() handles very rare recursive edge cases and is slow. When simply copying a list or other data structures, you can construct a new list by list() constructor or : operator. Some people avoid the second notation due to readability, but it's frequently used in the real world.

A. You don't need to squeeze your code's performance by fancy optimization techniques such as bit shifting or reducing the number of function calls (i.e. putting every operation in one function for reducing overhead of function calls). Except copy.deepcopy(), most of your design choices are about choosing best data structures in terms of time/space complexity.

Q. Is there any other test cases?

A. The following two test cases might help for your stat validation. Note that long path_to_goals are truncated and running_time/max_ram_usage are removed:

```
python puzzle.py dfs 6,1,8,4,0,2,7,3,5
```
path_to_goal: ['Up', 'Left', 'Down', ... , 'Up', 'Left', 'Up', 'Left']
cost_of_path: 46142
nodes_expanded: 51015
search_depth: 46142
max_search_depth: 46142


```
python puzzle.py bfs 6,1,8,4,0,2,7,3,5
```
path_to_goal: ['Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Left',
'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Left', 'Up', 'Up']
cost_of_path: 20
nodes_expanded: 54094
search_depth: 20
max_search_depth: 21


```
python puzzle.py ast 6,1,8,4,0,2,7,3,5
```
path_to_goal: ['Down', 'Right', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Left',
'Up', 'Right', 'Right', 'Down', 'Down', 'Left', 'Left', 'Up', 'Up']
cost_of_path: 20
nodes_expanded: 696
search_depth: 20
max_search_depth: 20


```
python puzzle.py dfs 8,6,4,2,1,3,5,7,0
```
path_to_goal: ['Up', 'Up', 'Left', ..., , 'Up', 'Up', 'Left']
cost_of_path: 9612
nodes_expanded: 9869
search_depth: 9612
max_search_depth: 9612


```
python puzzle.py bfs 8,6,4,2,1,3,5,7,0
```
path_to_goal: ['Left', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Right', 'Right',
'Up', 'Left', 'Left', 'Down', 'Right', 'Right', 'Up', 'Left', 'Down', 'Down', 'Right', 'Up', 'Left',
'Up', 'Left']
cost_of_path: 26
nodes_expanded: 166786
search_depth: 26
max_search_depth: 27


```
python puzzle.py ast 8,6,4,2,1,3,5,7,0
```
path_to_goal: ['Left', 'Up', 'Up', 'Left', 'Down', 'Right', 'Down', 'Left', 'Up', 'Right', 'Right',
'Up', 'Left', 'Left', 'Down', 'Right', 'Right', 'Up', 'Left', 'Down', 'Down', 'Right', 'Up', 'Left',
'Up', 'Left']
cost_of_path: 26
nodes_expanded: 1585
search_depth: 26
max_search_depth: 26

Q. Why does the example of python puzzle.py dfs 1,2,5,3,4,0,6,7,8 return ['Up', 'Left', 'Left'] instead of ['Up', 'Left', 'Down', ...] (a solution path with 31 moves)? We are using UDLR (Up, Down, Left, Right) order, and Down move should be executed before Left move. Isn't the ['Up', 'Left', 'Left'] solution resulted from optimization forbidden in project instruction?

No, ['Up', 'Left', 'Left'] solution does not use the forbidden optimization and is a correct answer. Compared to the simplicity of the 3-move solution path, you would notice that "nodes_expanded" statistic is extremely large (181437 states). In fact, the total reachable states in (solvable) 8-puzzle is 9!/2 = 181440 states, so this statistic suggests depth first search constantly overlooks the goal state and expands more than 99.9% of possible states in the search space.

Why is this happening?
Please think about the reason for one minute before reading the following explanations. There are two facts we can read from dfs pseudocode in class slides:

Fact 1. goalTest() function is only called for states which are just popped out from frontier. In other words, goalTest() is not applied to neighbor states even if the neighbor is actually the solution state (because we prohibited such an optimization).
Fact 2. A child state (neighbor) is only pushed into frontier when it's not already in frontier (and explored). More specifically, the goal state is only pushed into frontier when it's not already in frontier.

Combining those two facts reaches to the conclusion: since the goal state is already pushed into frontier (i.e. the state corresponding to ['Up', 'Left', 'Left'] move), any subsequent encounter to the solution state cannot execute frontier.push() or goalTest() and is effectively meaningless. Thus, the dfs algorithm extensively searches through numerous states in 8-puzzle (without putting the solution state into frontier again), gradually goes back to the previously pushed states, and finally find the solution state which is pushed at the very beginning of the search.

This question would arise when you remove/forget neighbor in frontier checking in your pseudocode. And if you add the checking, you will face the slowness of membership checking. In that case, please see the first question of this page ("Q. My search algorithm seems correct but is too slow. How can I reduce its running time?").

Q. Why the max_search_depth of python puzzle.py bfs 1,2,5,3,4,0,6,7,8 is 4 even though the goal state is at depth 3? The following figure would be useful (please ignore g and h values)