HW #1 – Solving N-Puzzle Problem Using the BFS, ID and A\* Algorithms

Due date: October 11 via Github

If you haven’t yet added your Github information, please do so at:

<https://docs.google.com/spreadsheets/d/109YNIcUSgZJxlq3S_hPsTJJJylvMWw8wXHWP1HiFbTo/edit?usp=sharing>

Part 1— BFS and ID Search. The files are at:

<https://drive.google.com/drive/folders/1PDSPDnUecrDZa1Zou9apIJsYFj5AUt99?usp=sharing>

Exercise #1-Part 1:

Attached are the following files:

1. stack.py – Implementation the data structure for ID.
2. frontier.py – Implements the data structure for storing states for the ID and BFS search. For BFS, this data structure implements a queue. For ID this is an "extended" stack - a stack that stores states up to depth of search d. When it is empty it reloads the initial state and increases d by 1.
3. state.py – Implements an N-Puzzle.
4. search.py – Implements the search pseudocode from class.

You need to go through the attached files, understand them, and perform the following tasks:

1. Understand how search.py implements the search pseudocode taught in class.
2. Modify the data structure to save the number of states stored in the structure (stored in the open list). As a result, the frontier will include 5 items and not 4:

 [#stack, max. depth, init. state, try next level?, total items pushed]

1. Write a program that prints the maximum depth and number of states inserted (added to the open list) and removed (popped) as averaged from 100 runs.
2. Benchmark the time it took for the search. An example for this type of timing can be found at:

<https://colab.research.google.com/drive/1933G-UpY8rO34sl1B_ZKRi432SVHdHAc?usp=sharing>

1. Check that the code works for simple cases. For example, the output for search(2) can be:

Average depth: 1.83

Average number inserted: 6.39

Average number removed: 4.56

Average runtime: 0.02 seconds

1. Run search (2), search (3) and search (4) 100 times and print the averages of the depth, inserts, removes and times. What do you see?  For purposes of completeness, I know that some of these searches will timeout.  To get around this, I suggest that any time a search takes more than 5 seconds (e.g. check: 1000\*(toc-tic)), you give up and you add large numbers for the values to the counters (e.g. Average depth = 20, Average number inserted / removed = 10000, runtime=100 seconds).

What can you say about these searches?  Which one seems to work better? Note: we will show that A\* does much better than both of these, but that is for part-2 😊 Nothing to submit yet – I will have you submit parts 1 and 2 together, and we will discuss part two next class.

\* Reminder from the lecture:

search(n)

f←frontier.create(state.create(n))

while not frontier.isEmpty(f) do

    s←frontier.delNext(f)

    if state.isTarget(s)

    then return s

    ns←state.getNext(s)

    for i←1 to length(ns) do

      frontier.insert(f,ns[i])

return null

Part 2: A\*

Attached are the following files:

1. frontier.py – Implementing frontier with a priority queue.  Will be used to prioritize states based on a value function, state.hdistance(s)+state.path\_len(s).  You will need to write two versions of state.hdistance(s) ☺
2. state.py – Implements an N-Puzzle, except for the heuristics.  Your assignment!
3. search.py – Implements the search pseudocode from class.

You need to go through the attached files, understand them, and perform the following tasks:

1. Run the code “as-is” and note that this is uniform search (why?).  Run the code 100 times for the 2X2, 3X3, and 4X4 puzzles. Save/print the output of the number of states inserted, checked (removed), solution depth and time as averaged from 100 (still not always possible for the 4X4 puzzle).  What do you see?  For purposes of completeness, I know that some of these searches will timeout.  To get around this, I suggest that any time a search takes more than 5 seconds (e.g. check: 1000\*(toc-tic)), you give up and you add large numbers for the values to the counters (e.g. Average depth = 20, Average number inserted / removed = 10000, runtime=100 seconds).
2. Implement the hdistance1 function to accurately return the number of tiles out of place and change the code in frontier to return state.hdistance1(s)+state.path\_len(s).  Run the code 100 times and and save/print the output of the number of states inserted, checked (removed), solution depth and time as averaged from 100 runs for the 2X2, 3X3, and 4X4 puzzles (should now be almost always possible in all cases).
3. Implement the hdistance2 function to accurately return the Manhattan distance of the tiles from their target position and change the code in frontier to return state.hdistance2(s)+state.path\_len(s). Run the code 100 times and save the output of the number of states checked (total items inserted and deleted) as averaged from 100 runs in a 2X2, 3X3, and 4X4 puzzles.
4. Compare these values to those you achieved in the first part and print a summary table that looks like this:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Depth | Inserts | Removes | Time |
| BFS |  |  |  |  |
| DFS |  |  |  |  |
| Uniform |  |  |  |  |
| A\*-h1 |  |  |  |  |
| A\*-h2 |  |  |  |  |

\* Reminder from the lecture:

