CSDS 310

Professor Lewicki

John Mays, jkm100

**P1 Extra Credit Writeup**

Submitted

**Introduction**

I left out the experiments and discussion section, since they were not asked for. I felt that, despite its inclusion in first writeup, it was worth keeping some of the discussion about my code and augmenting it to show what I changed and what I did not, since that was emphasized in the instructions.

**1. Code Design**

**Basic Organizational Details:**

I have written my code in python with the only nonstandard library being the approved numpy. My zip contains a main.py file with the bulk of the code in it, an exceptions.py file for my custom errors, an experiments.py file for part C, my test file: P1\_jkm100\_test\_file.txt, and a .pdf of this writeup.

Within main.py, there are only a few global variables: the goal state, the “current” state, and the maximum number of nodes allowed.

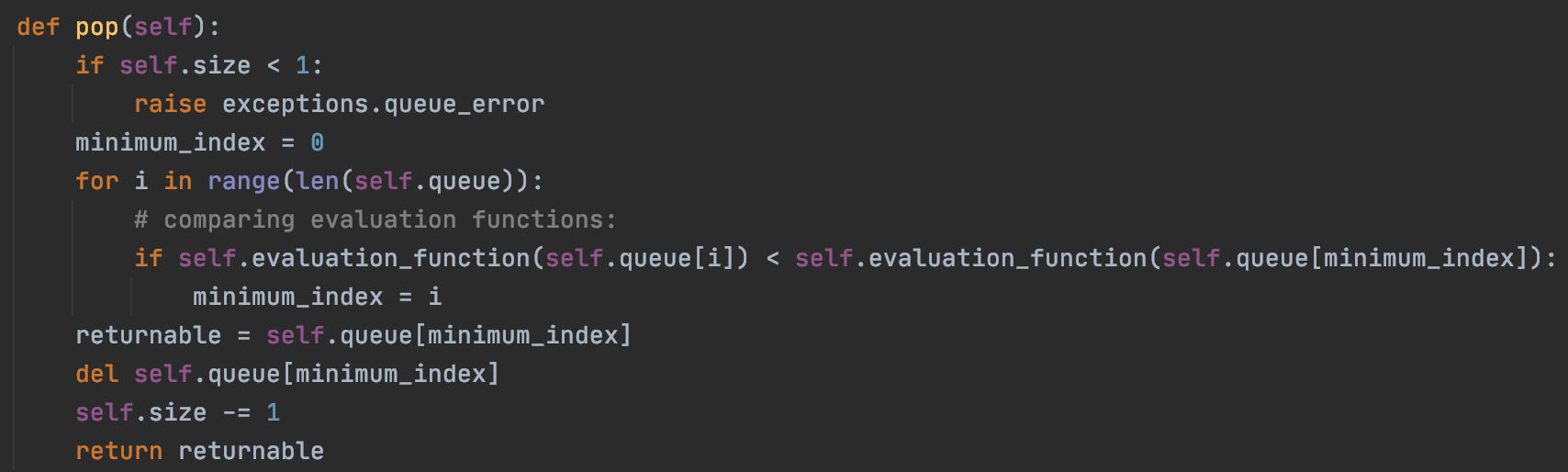
The \_\_main\_\_ function calls a function named interpreter(), that takes the txt with filename specified in the command line as an argument, and executes the proper python commands based on what it determines the file is saying with simple string processing.

**Running Code, Briefly:**

So, in order to run my code, I would open my terminal, navigate to the P1 directory, type this command:

Python3 main.py P1\_jkm100\_test\_file.txt

…and press enter. The output should show up in the terminal.



The implementation of the pop() function.

The third important data structure are is my beam\_priority\_queue. It is very similar to a\_star\_priority\_queue, except for the fact that it uses solely h2 as its evaluation function.

**The Functions:**

There are quite a few helper functions, all of which are listed after the primary functions. There are small helper functions such as string\_to\_int\_representation(), which converts a string such as “b12 345 678” to a 3x3 numpy matrix made of integers. But the important ones are as follows:

a\_star\_priority\_queue(heuristic) and beam\_priority\_queue(heuristic)

…discussed already.

heuristic\_one(node)

Given a node, observes its state, compares to goal\_state and returns the number of misplaced tiles.

heuristic\_two(node)

Given a node, observes its state and returns the sum of the Euclidean distances of tiles from their positions in the goal\_state.

check\_for\_success\_[search](node)

An often-run function from the searches that compares the node’s state to the goal\_state and, if equal, prints out the number of moves and the moves themselves by climbing back through the node.parent variables.

**A\* Search:**

The A\* search is fundamentally designed about the a\_star\_priority\_queue.

Here is the pseudocode for my algorithm:

solve\_a\_star(heuristic)

create root node

create frontier

place root node in frontier

if root node is the goal state:

return success

while frontier is full & number of nodes has not exceeded max\_nodes

cur\_node = pop from frontier

create 2-4 children

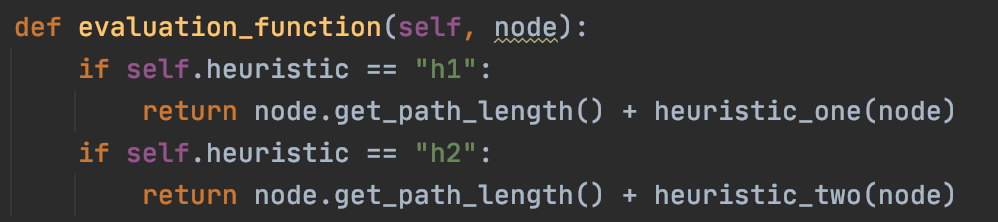
if children are the goal state:

return success

else: add them to the frontier

return failure: the number of nodes has exceeded max nodes

This leaves out arguably the most important part: comparing the evaluation functions. It occurs within the priority queue. When the pop() function is called, the priority queue iterates through every element it has, calling the evaluation\_function() function on all of them, and returns the one with the lowest value.

****

**Beam Search:**

My beam search is structurally similar in certain ways to my A\* search. It uses beam\_priority\_queue, which is almost the exact same as a\_star\_priority\_queue, except for the fact that it uses an evaluation function f(node) = h2(node), which is zero at the goal state and >2 everywhere else.

I use the queue in a mildly unconventional way: I scan through, generate all of the children, put the children and parents in, pop the k best, clear the rest of the queue, and then put them back in. In practice, what would have been more efficient would have been to just implement a queue that only accepts k nodes. However, this does work.

The pseudocode is as follows:

solve\_beam(k)

create root node

create frontier (with beam priority queue)

place root node in frontier

if root node is the goal state:

return

while frontier is full & number of nodes has not exceeded max\_nodes

for the elements currently in the frontier

cur\_node = pop from frontier

generate children for it

if they are the goal state:

return

add parents back in

refine queue:

pop k nodes from queue

clear the rest of the queue

add the k nodes back in

return failure: the number of nodes has exceeded max nodes

**2. Code Correctness**

**Demonstrating** move() **and** setState()**:**

Here I first set the state and print it. After, I do some moves, the last of which should fail, I print again:

|  |  |
| --- | --- |
| Text Input: | Output: |
|  |  |

**Demonstrating A\*:**

Given a fair number of maximum nodes, my A\* always works. Here is an example with h1.

|  |  |
| --- | --- |
| Text Input: | Output: |
|  |  |

Here it is again with a new input and h2.

|  |  |
| --- | --- |
| Text Input: | Output: |
|  |  |

**Demonstrating Beam Search:**

Here, I give beam a k-value of 14.

|  |  |
| --- | --- |
| Text Input: | Output: |
|  |  |

**FAILURES:**

Two cases in which these searches fail are:

1. When the maximum number of nodes is smaller than the problem is large:

Here, I set the maximum number of nodes to twenty, and A\* does not find a solution before hitting the max:

|  |  |
| --- | --- |
| Text Input: | Output: |
|  |  |

1. When the k for beam search is very large, it surpasses the node limit.

Here, I run the same search under the same conditions on the same state twice, except on the second time, I give k = 70. This is very large compared to 12. This leads to ineffective pruning, the proliferation of useless nodes, and a surpassing of the node limit much sooner than on the more limited version. Consequently, k = 12 succeeds and k = 70 does not:

|  |  |
| --- | --- |
| Text Input: | Output: |
|  |  |

**Note:** these examples are included in my test file, “P1\_jkm100\_test\_file.txt”, in the order they appeared. Additionally, my RNG seed is set when main.py is first run, so the same results should appear.