CSDS 310

Professor Lewicki

John Mays, jkm100

**P1 Writeup**

Due on 09/30/21, Submitted 10/02/21

**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, my test file: P1\_jkm100\_test\_file.txt, and a .pdf of this writeup.

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 called 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

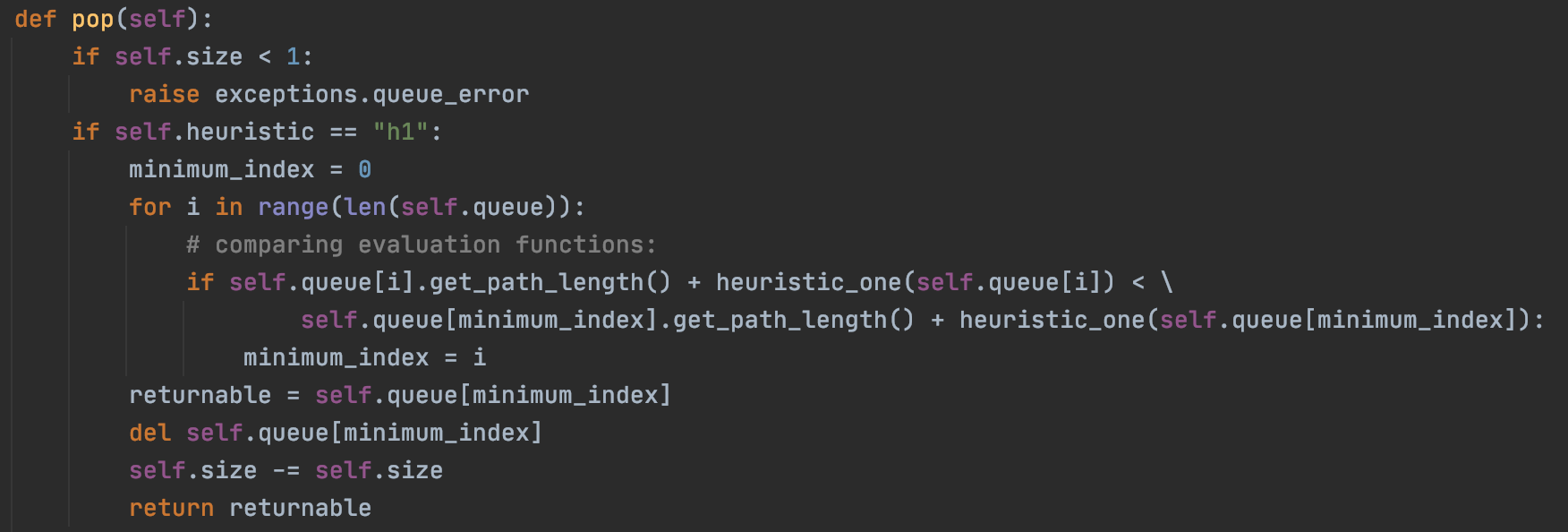
…and press enter. The output should show up in the command line.

**The Data Structures:**

There are two important data structures that I created.

The first one is a node, which is essentially a container object for a 3x3 state matrix that has a pointer for a parent node, a path\_length variable which is calculated on demand by tracing back to the root, and a move variable, which stores which move was taken to get to it.

The second important data structure is my priority\_queue. It stores individual nodes, has a very simple insert() function, and mildly complicated pop() function. The pop() function goes through all of the nodes it currently has and finds the one with the lowest evaluation function, which is a sum of the node’s path\_length variable and a heuristic function that the priority queue is instantiated with.



The implementation of the pop() function for h1.

**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:

priority\_queue(heuristic)

…discussed already.

heuristicOne(node)

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

heuristicTwo(node)

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

check\_for\_success(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 priority\_queue.

Here is the pseudocode for my algorithm:

solve\_a\_star(heuristic)

create root node

create frontier

place root node in frontier

check is root node the goal?

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;

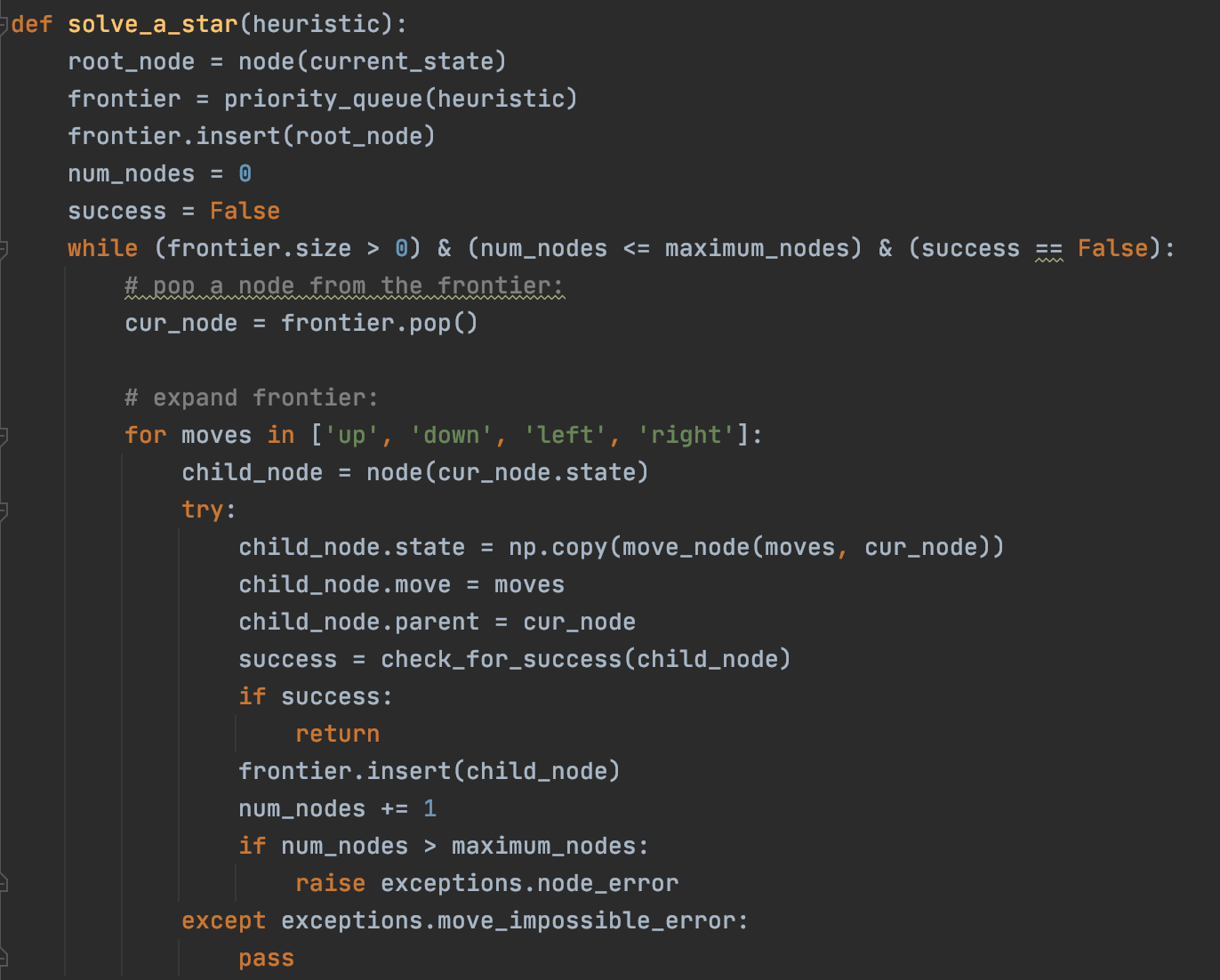
if f(child) < f(frontier.pop)

return & print success messages

else:

return child to frontier and keep going

As well as the code itself:

****

**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 priority\_queue, except for the fact that it uses an evaluation function f(node) = h1(node), which is zero at the goal state and >2 everywhere else.

The pseudocode is as follows:

solve\_beam(k)

create root node

create frontier (with special priority queue)

place root node in frontier

check is root node the goal?

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

check if they are success

add parents and children back in

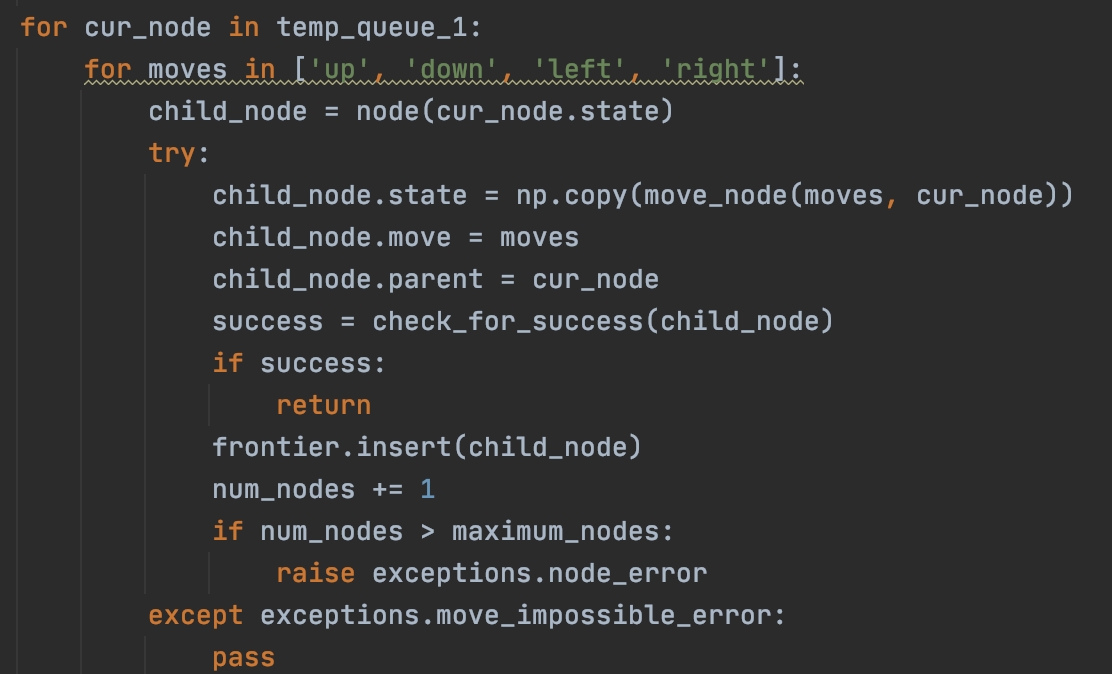
refine queue:

pop k nodes from queue

clear the rest of the queue

add the k nodes back in

I am going to refrain from including the entire function, because it is very long, but here is the central loop:



**2. Code Correctness**

Demo

**3. Experiments**

**4. Discussion**

**a.**

A\* search is optimal, because it will not give up searching until the priority queue has no better options to pursue, implying that it will only return a solution after it has discovered the shortest one. Meanwhile the beam search has nothing to check for this, and returns the first solution it encounters. Therefore, the A\* search finds the shorter solutions, in general. On the other hand, beam search is superior in terms of time and space in plenty of cases. There are some cases in which beam search could throw out routes to solutions before exploring them, where A\* would have gotten there beforehand, but because A\* is optimal, beam tends to quit first. Additionally, the worst-case space complexity for A\* is one in which the frontier could end up holding ~4d nodes, which becomes large very quickly. Meanwhile beam could only ever consider 4k nodes, which would remain quite small considering traditional k values. So for this problem, whe