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, 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 Data Structures:

There are three 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 are is my `a_star_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.

```
def pop(self):
    if self.size < 1:
        raise exceptions.queue_error
    minimum_index = 0
    for i in range(len(self.queue)):
        # comparing evaluation functions:
        if self.evaluation_function(self.queue[i]) < self.evaluation_function(self.queue[minimum_index]):
            minimum_index = i
    returnable = self.queue[minimum_index]
    del self.queue[minimum_index]
    self.size -= 1
    return returnable
```

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.

```
def evaluation_function(self, node):
    if self.heuristic == "h1":
        return node.get_path_length() + heuristic_one(node)
    if self.heuristic == "h2":
        return node.get_path_length() + heuristic_two(node)
```

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:

`setState 1b2 345 678`
`printState`
`move down`
`move right`
`move down`
`move left`
`move left`
`move left`
`printState`
	`Current State:`
	`[['1' 'b' '2']`
	`['3' '4' '5']`
	`['6' '7' '8']]`
	
	`this move was not accepted`
	`Current State:`
	`[['1' '4' '2']`
	`['3' '5' '8']`
	`['b' '6' '7']]`

Demonstrating A*:

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

Text Input:	Output:

`maxNodes 1024`
 	
`randomizeState 11`
`printState`
`solve A-star h1`
	`Current State:`
	`[['3' 'b' '1']`
	`['4' '5' '2']`
	`['6' '7' '8']]`
	
	`SUCCESS:`
	`Number of moves needed to find solution: 5`
	`Solution:`
	`right`
	`down`
	`left`
	`left`
	`up`

Here it is again with a new input and h2.

Text Input:	Output:

`randomizeState 10`
`printState`
`solve A-star h2`
	`Current State:`
	`[['1' '5' 'b']`
	` ['3' '2' '4']`
	` ['6' '7' '8']]`
	
	`SUCCESS:`
	`Number of moves needed to find solution: 6`
	`Solution:`
	`left`
	`down`
	`right`
	`up`
	`left`
	`left`

Demonstrating Beam Search:

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

Text Input:	Output:

`randomizeState 9`
`printState`
`solve beam 14`
	`Current State:`
	`[['6' '3' '2']`
	` ['b' '1' '5']`
	` ['7' '4' '8']]`
	
	`SUCCESS:`
	`Number of moves needed to find solution: 7`
	`Solution:`
	`up`
	`right`
	`down`
	`down`
	`left`
	`up`
	`up`

FAILURES:

Two cases in which these searches fail are:

a) 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:
maxNodes 20 randomizeState 11 printState solve A-star h1	Current State: [['4' 'b' '2'] ['1' '3' '5'] ['6' '7' '8']] A* Failure: Maximum number of nodes surpassed

b) 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:
maxNodes 70 randomizeState 12 printState solve beam 12 printState solve beam 70	Current State: [['3' '1' '2'] ['6' 'b' '5'] ['7' '4' '8']] SUCCESS: Number of moves needed to find solution: 4 Solution: down left up up Current State: [['3' '1' '2'] ['6' 'b' '5'] ['7' '4' '8']] BEAM FAILURE: Maximum number of nodes surpassed

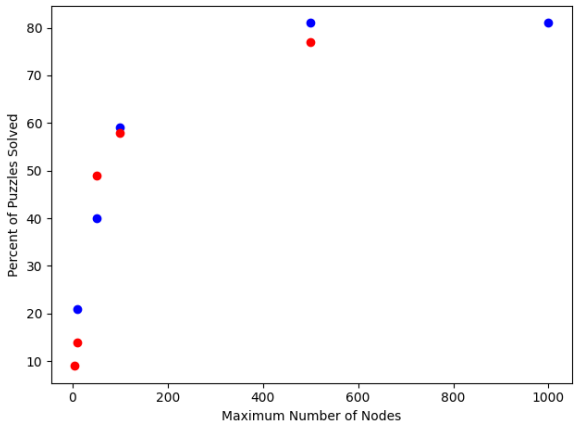
Note: these examples are included in my test file, "`P1_jkm100_test_file.txt`", in the order they appeared. Additionally, my RNG seed is set when `main.py` is first run, so the same results should appear.

3. Experiments

a.

```
Limit of 10 produced 21% success rate.
Limit of 50 produced 40% success rate.
Limit of 100 produced 59% success rate.
Limit of 500 produced 81% success rate.
Limit of 1000 produced 80% success rate.
```

I used A* search with h1 for this experiment. I randomized the conditions with 25 moves and let them run 100 times per maxNodes value. It seems as if the fraction of solvable puzzles varies with the maximum number of nodes logarithmically. Here are a few different runs on a scatter plot to illustrate this behavior:



b.

```
A* with h1 solved 47.6% of the puzzles.
A* with h2 solved 73.2% of the puzzles.
```

Heuristic two far outperforms heuristic one. I ran several trials of one versus the other, each on the same puzzle set of 500, shuffled randomly for 25 moves, and heuristic two consistently did around 25% better. Theoretically speaking, I believe this is because h2 contains more information relative to success than h1 does. h2 also describes the number of misplaced tiles, but it describes to what it extent they are misplaced. As an example, say we have two puzzle states, each with six misplaced tiles, but the first one has severely misplaced tiles, while in the second, every tile is one move away from its original position. h1 sees no difference between the two. However, h2 knows that the first is a worse option to pursue.

c.

A* with heuristic one and heuristic two always return a solution with the same length, the best solution. Beam search, however, must have a solution that is greater than or equal to the optimal solution, and returns a solution that is typically 2-5 moves greater, approximately 15% of the time, in cases where they all succeed. This is speaking in terms of randomization numbers and node limits that my computer can easily run tests with.

d.

To determine what fraction of my generated problems were solvable by the algorithms, I picked parameters that I typically used along the way: `maxNode` limits of 20, 25, 70, and 100, as well as number of moves for `randomize_state` at 7, 9, 12, 14 and 30. I generated 50 puzzles per parameter permutation, and tried each algorithm on all 600 puzzles. After that, I calculated the average number solved across all 1000:

```
A* with h1 solved 76.2% of problems.
A* with h2 solved 83.9% of problems.
Beam solved 76.2% of problems.
```

4. Discussion

a.

A* search is optimal, because it always checks nodes with the lowest evaluation first, which implies that if it found a solution that solution would have the lowest path cost of any solution (given that the heuristic is admissible). Meanwhile the beam search does not necessarily have to return the shortest path. It returns the first solution it runs into, which could easily be a nonoptimal one, considering the fact that a whole set of nodes are given children and checked upon every iteration of the while loop, not just the one with the shortest path. 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 ~4^d nodes, which becomes large very quickly. Meanwhile beam could only ever consider 4k nodes, which would remain quite small considering traditional k values. In conclusion, in this problem of the 8-puzzle, optimality is not the object of the game, simply finding a solution is. Therefore I would choose beam search based upon its ability to preserve computing resources and find a solution quickly.

b.

Implementing both of these algorithms presented a very big challenge for me. I would get to a point where I thought I wrote them properly, and then upon scrolling through random test cases, find a starting position that failed and made no sense. The most difficulty I had probably came from not having my heuristic 2 correct for a very long time. I initially had it calculating the Manhattan distance as opposed to the Euclidean distance. And not only that, but it also treated the blank tile as a tile for which to calculate the distance. Both of these errors contributed to its gross inadmissibility, which messed up both my beam and A*(h2) in quite a few instances.

Another thing that I found difficult was envisioning the proper way to organize my program. I wasted a lot of time on this assignment refactoring my code for silly constraints that I should have foreseen. I believe that if I had drawn out a modular diagram and written some pseudocode to begin with, I would have not made stupid mistakes such as designing several of my functions without the existence of a node object in mind. For projects as big as this, diving straight into the coding is definitely not the best approach, especially when certain parts of the algorithm are still unclear. So, for my next project, I definitely plan to allocate more time to pre-planning.