Code Design

My code is mainly split into multiple files by their class, the Board state, A-star search algorithm, and the File/Command-line Interface. When the program starts, it checks for any command line arguments pointing to a file to execute. If the argument exists, it then parses the file and executes each line.

The Command executor class handles the command parsing and function calling. When this class is instantiated, it is provided with a context of type dictionary and a dictionary of possible functions. The context is needed in order to maintain a state across commands, whilst keeping it separate from the global variable space in python. When a function is executed, it is tokenized then the matching function to the first token is matched and called. If no matching function is found, the help message is displayed instead.

The dictionary of possible functions are provided from outside the Command executor class. I chose to place these in the init file as they were not complicated enough to justify their own class, whilst being too different from the purpose of the line executor to include it there. All these functions wrap functions from the other two classes (A-star and Board) to play nicely with that execution's own state. The state includes the current board, node-limit, and the random seed. Before a function gets executed, these details are printed so that it is easier to understand the flow of the program as things run.

```
def solve(context, type, arg):
    """(algorithm, arg): Solver Wrapper and heuristic picker for A-star and
Beam Search"""
    if type == "A-star":
        F = AStar.Heuristics.h0 if arg == "h0" else AStar.Heuristics.h1 if arg
== "h1" else AStar.Heuristics.h2
        B = AStar(F, context["puzzle"], Board(),
nodeLimit=context["node_limit"])
    elif type == "beam":
        B = AStar(AStar.Heuristics.h2, context["puzzle"], Board(),
beamLimit=int(arg), nodeLimit=context["node_limit"])

    B.solve()
    print(B.getSolution())

FUNCTIONS = {
    "setState": setState,
    "printState": printState,
    "move": move,
    "randomizeState": randomizeState,
    "solve": solve,
    "maxNodes": maxNodes,
    "setSeed": setSeed,
    "readFile": readFile,
```

```
    "help": help
}
```

The Board class handles movements done to the state, creating random states, and parsing state strings. There are also other minor methods to define how it should be printed and copied, as well as internal methods like findValue, which finds the position of a number on the board.

```
@staticmethod
def randomState(n:int = 1000, seed:int = 12497989):
        """Constructs a random Board"""
        R = random.Random(seed)
        r = Board()
        [r.move(Board.directions[i]) for i in [R.randint(0, 3) for j in
range(n)]]
        return r
```

The A-star class handles all the solving of the state. It contains its internally used Node class as well as the static heuristics class to be selected from. The node class acts as the mediator between the A-star solver and the Board class, both containing it and adding information relevant to A-star like cost, parent, previous action, and a method to get possible actions. The main A-star class parses many parameters to solve the state according to specifications, by heuristic, beam limit, node limit, and goal state. Then there are two main methods: iterate and solve. Iterate explores one node according to A-star, and solve continuously iterates until a solution is found. I chose to split the solve and iterate methods to easier understand the structure of A-star without greatly compromising on performance.

```
def iterate(self):
        """Discover one tile according to the A-Star heuristics, and add it to
the queue"""
        _, node = heapq.heappop(self.queue)
        if node.state.value == self.goalState.value:
                self.solution = node
                return True

        for neighbor in node.getNeighbors():
                if neighbor.state.value in self.visited:
                        continue

                self.visited.append(neighbor.state.value)

                heapq.heappush(self.queue, (neighbor.cost +
self.heuristic(neighbor.state, self.goalState), neighbor))

        if self.beamLimit > 0:
                self.queue = self.queue[:self.beamLimit]
```

Code Correctness

My implementation of A* initially creates the priority queue - with the initial node in it at cost 0, then the empty visited list. After that, all other parameters of the solver are saved (goal state, heuristic, beam limit, node limit, initial state)

```
self.queue:list[AStar.AStarNode] = []
self.visited = []
self.visited.append(self.initState.value)
heapq.heappush(self.queue, (0, AStar.AStarNode(self.initState)))
```

Then, whenever iterate is called,

1. The lowest cost node is popped off the queue
2. The node is compared against the goal state and exits if matches
3. Possible actions are generated
    I. They are then filtered by which have already been explored
    II. If not, add them to the priority queue along with their cost added to the heuristic function
    III. Add them to the visited list
    I chose to add them to the priority queue so that new nodes discovered will not duplicate states with items stuck in queue already
4. If is beam-search, reduce the priority queue to fit the limit

```
def iterate(self):
        """Discover one tile according to the A-Star heuristics, and add it to
the queue"""
        _, node = heapq.heappop(self.queue)
        if node.state.value == self.goalState.value:
                self.solution = node
                return True

        for neighbor in node.getNeighbors():
                if neighbor.state.value in self.visited:
                        continue

                self.visited.append(neighbor.state.value)

                heapq.heappush(self.queue, (neighbor.cost +
self.heuristic(neighbor.state, self.goalState), neighbor))

        if self.beamLimit > 0:
                self.queue = self.queue[:self.beamLimit]
```

Using the command line feature of my program (run python file without arguments), I test the randomize feature along with the A* solver

```
0.0s > randomizeState 5000
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['randomizeState', '5000']
```

```
0.004s > printState
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['printState']
340
268
715
0.0s > solve A-star h2
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h2']
26: right right up left left up right down down left up up right right down
left down right up left left down right up right down
4.573s >
```

I then cut out pieces of paper and validated that the result solves the problem.

Next, I gave it a simple position (two left moves) and checked if our intuition lined up with the solution it comes up with.

```
0.0s > setState 120345678
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['setState', '120345678']
0.0s > printState
{'puzzle': 120 345 678, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['printState']
120
345
678
0.0s > solve A-star h2
{'puzzle': 120 345 678, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h2']
2: right right
0.0s >
```

It does in fact find the exact moves to reverse the scrambling (and in 0.000s)

I have made sure to use the default seed that I provided, which can also be changed with the `setSeed` function

Experiments

How does fraction of solvable puzzles from random inital states vary with the maxNodes limit?

As maxNodes decreases, the number of solvable puzzles decreases simultaneously as there was not enough time to find a solution

For example randomState 5000 fails with 6000 nodes, but succeeds with 7000

```
> randomizeState 5000
{'puzzle': 340 268 715, 'node_limit': 6000, 'seed': 12497989, 'show_time':
True} ['randomizeState', '5000']
0.004s > maxNodes 7000
{'puzzle': 340 268 715, 'node_limit': 6000, 'seed': 12497989, 'show_time':
True} ['maxNodes', '7000']
0.0s > solve A-star h2
{'puzzle': 340 268 715, 'node_limit': 7000, 'seed': 12497989, 'show_time':
True} ['solve', 'A-star', 'h2']
26: right right up left left up right down down left up up right right down
left down right up left left down right up right down
4.507s > maxNodes 6000
{'puzzle': 340 268 715, 'node_limit': 7000, 'seed': 12497989, 'show_time':
True} ['maxNodes', '6000']
0.0s > solve A-star h2
{'puzzle': 340 268 715, 'node_limit': 6000, 'seed': 12497989, 'show_time':
True} ['solve', 'A-star', 'h2']
None
4.143s >
```

This happens because the algorithm will never be able to explore that node without exploring the 6000 other ones before it. So if we limit it, then we will simply never find the goal state

For A* search, which heuristic is better?

h2 is definitely the best heuristic. Compared to both h1, and h0, which I defined and 1 if not matching the goal state and 0 if it is.

```
0.0s > randomizeState 5000
{'puzzle': 012 345 678, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['randomizeState', '5000']
0.005s > solve A-star h2
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h2']
26: right right up left left up right down down left up up right right down
left down right up left left down right up right down
4.359s > solve A-star h1
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h1']
26: up right right down left up up right down left left up right down left down
right right up up left left down right down right
221.607s >
```

Heuristic h2 took 4.359s to solve randomState 5000, whilst h1 took over 221s to solve it, without any decrease in the path length.

On a smaller sample randomState 250:

```
> randomizeState 250
{'puzzle': 571 432 068, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['randomizeState', '250']
0.0s > solve A-star h2
{'puzzle': 751 320 648, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h2']
17: right down left up right up right down down left up right up left down down
right
0.04s > solve A-star h1
{'puzzle': 751 320 648, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h1']
17: right down left up right up right down down left up right up left down down
right
0.134s > solve A-star h0
{'puzzle': 751 320 648, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h0']
17: right down left up right up right down down left up right up left down down
right
15.63s >
```

h2 takes 0.04s, h1 takes 0.134s, whilst h0 takes 15.63s and they all produce the exact same resulting sequence.

h2 is objectively the best heuristic out of all three, as it is the fastest while still remaining optimal.

How does the solution length vary across the three search methods?

Both the A-star heuristics produce the same optimal length for their solutions, but beam search may take produce a subobtimal solution while computing it much faster

```
0.0s > randomizeState 5000
{'puzzle': 751 320 648, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['randomizeState', '5000']
0.004s > solve A-star h2
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'A-star', 'h2']
26: right right up left left up right down down left up up right right down
left down right up left left down right up right down
4.613s > solve beam 500
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
```

```
'show_time': True} ['solve', 'beam', '500']
26: right right up left left up right down down left up up right right down
left down right up left left down right up right down
1.203s > solve beam 300
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'beam', '300']
26: up right up left down down right up right up left left down right right up
left down right down left left up right down right
0.641s > solve beam 200
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'beam', '200']
26: up right up left down down right up right up left left down right right up
left down right down left left up right down right
0.435s > solve beam 100
{'puzzle': 340 268 715, 'node_limit': 1000000000, 'seed': 12497989,
'show_time': True} ['solve', 'beam', '100']
32: right up right down left up up left down down right up left down right
right up left down right up left up right down down left left up right right
down
0.432s >
```

For the same sample randomSample 5000, h2 produces a 26 long solution in 4.6s, and the equivalent beam search with a limit of 500 produces the same result in only 1.2s, an even lower beam size of 200 still makes a 26 long solution but only in 0.435s, over a 10x speed up over regular A* without any subobtimalities yet. However, at a beam size of 100, it produced a result of length 32. h1 also also always produces an optimal result, like h2 but just takes longer.

For each of the three search methods, what fraction of your generated problems were solvable?

All problems were solvable with enough time, however I would consider the h1 solve of randomState 5000 at 221.607s to be unacceptable and not a feasible solve. With a node-limit the number of solvable problems significantly drops.

Discussion

Based on your experiments, which search algorithm is better suited for this problem?Which finds shorter solutions? Which algorithm seems superior in terms of time and space?

I think since this is such a simple puzzle, we can sacrifice an extra 2 seconds to get a fully optimized answer with A. *A* will always produce an optimal length solution, whilst beam search may or may not. Beam search is definitely more memory friendly and time efficient, however I would still argue for this small scale, a 2 second penalty is sufferable.

Discuss any other observations you made, such as the difficulty of implementing and testing each of these algorithms.

Initially I made a couple mistakes, like forgetting to subtract one from the list of nodes, or forgetting to reverse the move array, but the actual A* portion of the code worked first time and I was surprised as how simple it was to implement. Testing was not too bad, I just tested it on a real puzzle and checked it it worked.