P1
Write Up
Jackson Jacobs

Design Choices

1. Wrote code in Python. Python has great data structures package support including numpy, pandas and queue. It is also faster to write code using Python than Java.

2. Implemented A* and Local Beam searches as their own classes

As algorithms, I expected A* and Local Beam to require a few helper methods. Additionally, both algorithms need to store important data as they run, so I figured to use fields to denote the more important variables.

3. Implemented LocalBeam as a class that contains a list of beams.

When the search algorithm is running, it loops through these beams sequentially (not in parallel) to expand each one.

```
# ----------INIT----------- #
def __init__(self, game_node: BoardNode, k, max_nodes):
    self.unique = Iterator(count())
    self.maxNodes = max_nodes

    self.root = game_node
    self.beams = []
    self.moves = []     # filled with arrays that contain the logged moves of their respective beam.
    self.all_children = PriorityQueue()

    for i in range(k):  # for each beam
        beam = BoardNode(game_node.get_node_state().copy(), parent=None, action=None, path_cost=0) # initialize each beam
        logged_moves = beam.randomize_state(10, reset_state=False)     # branch beam state
        self.moves.append(logged_moves)        # for beam state i, log its moves away from the root.
        self.beams.append(beam)        # append beam to beams array
```

4. Use numpy arrays to store the game state

Numpy arrays come with useful functionality such as .flatten(). I use this method because it makes it easier to examine the entire state using a single loop.

5. Use a hashtable to store “reached” states.

The easiest implementation I found for this was to use the default python dict object. Each time I would like to add a child to the “reached” dict I write:

```
self.get_reached()[key] = child
```

6. Used pandas to print gamestates. I just prefer the formatting.

7. Tested methods using unit testing.

When methods had outputs that were easy to assert in python's unittest module, I wrote test methods for them. Because python's unittest module does not natively support matching arrays, I wrote a function that checks for the equality of two 2D arrays. All unit tests displayed in the TestEightPuzzle.py file were passed.

```
Tests passed: 6 of 6 tests – 1 ms
/usr/bin/python3.8 /home/jackson/.local/share/JetBrains/Toolbox/apps/PyCharm-P/ch-0/212.5080.64/plugins/python/helpers/pycharm/_jb_unittest_runner.py --target
Testing started at 5:35 PM ...
Launching unittests with arguments python -m unittest TestEightPuzzle.MyTestCase in /home/jackson/cwru_2021/ai/projects/project_1


Ran 6 tests in 0.005s

OK

Process finished with exit code 0
```

```python
def states_equal(s1, s2):
    state1 = np.array(s1).flatten()
    state2 = np.array(s2).flatten()

    for i in range(len(state1)):
        if state1[i] != state2[i]:
            return False
    return True
```

8. Deliberately chose slower method find_blank()

A faster implementation would have been to store the blank spot as a field in the BoardNode class. I chose to search the entire array for the blank tile because the board is relatively small and it meant I didn't need to reassign the blank spot field every time I made a move.

```python
def find_blank(self):
    state = self.get_node_state()
    for r in range(state.shape[0]):
        for c in range(state.shape[1]):
            if state[r][c] == 0:
                return r, c
```

9. Used h2 as the heuristic function in local beam search. I like the manhattan search because I am from there.

Project Structure

Project1

```
├── P1_Write_Up.pdf
├── 391-P1.pdf
├── Commands.py
├── EightPuzzle.py
├── Searches.py
├── TestEightPuzzle.py
```

```
├── testing_a_star.py
└── testing_beam.py
```

Commands.py: Contains all of the commands necessary to demonstrate the code's correctness.

EightPuzzle.py

- Class BoardNode: contains fields associated with a single game state of EightPuzzle
  - get_node_state()
  - set_node_state()
  - get_parent()
  - get_action()
  - get_path_cost()
  - f()
  - print_state()
  - find_blank()
  - valid_directions()
  - randomize_state()
  - is_goal()
  - move()
  - h1()
  - h2()

Searches.py

- Class Iterator: A wrapper class for itertools.count(). Allows the user to get the current value of the iterator without changing the state of the iterator.
- Class AStar: A class that implements the A* search.
  - get_node()
  - get_frontier()
  - get_reached()
  - get_moves()
  - append_move()
  - is_reached()
  - get_max_nodes()
  - get_unique()
  - solve_a_star()
- Function expand()
- Class LocalBeam: A class that implements Local Beam Search with k beams.
  - get_beams()
  - get_moves()
  - get_root()
  - get_all_children()
  - get_max_nodes()
  - get_unique()

- solve_beam()

TestEightPuzzle.py

- function states equal
- class MyTestCase
  - test_set_and_get_state()
  - test_move()
  - test_h1()
  - test_h2()
  - test_is_goal()
  - test_find_blank()

testing_beam.py
initial command draft for testing beam

testing_a_star.py
initial command draft for testing beam

Code Correctness
NOTE: For evaluating lower-level methods for correctness, see the unittest document, TestEightPuzzle.py

1. Set-up commands
   a. Import statements **(see requirements.txt for required modules)**

```
from EightPuzzle import BoardNode
from Searches import AStar, LocalBeam
import numpy as np
```

   b. Set random seed

```
np.random.seed(2021)
```

   c. define some easy initial problem states (b1,b2,b3). b1 is already a solved state, b2 is 1 move away from being solved, b3 is 2 moves away.

```
b1 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]
b2 = [[1, 0, 2], [3, 4, 5], [6, 7, 8]]
b3 = [[1, 4, 2], [3, 0, 5], [6, 7, 8]]
```

2. Demonstrate the correctness of A* search.
   a. initialize BoardNode from one of these states

```
b = BoardNode(b3, parent=None, action=None, path_cost=0)
```

   b. Initialize AStar. Then run solve_a_star(). Either ‘h1’ or ‘h2’ may be specified for the heuristic function. max_nodes may also be played with.

```
search = AStar(b, 'h1', max_nodes=10000)
search.solve_a_star()
```

The output should be:

```
initial board state:
   0  1  2
0  1  4  2
1  3  0  5
2  6  7  8
solved!
list of moves: [None, 'up', 'left']
number of moves: 3
   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8
```

c. Now try with a randomized problem state. Play with the n, the argument of .randomize_state()

```
b_hard = BoardNode(b1, parent=None, action=None, path_cost=0)
b_hard.randomize_state(30)

search = AStar(b_hard, 'h2', max_nodes=10000)
search.solve_a_star()
```

Output should be:

```
initial board state:
   0  1  2
0  2  8  4
1  1  0  5
2  3  6  7
solved!
list of moves: [None, 'left', 'up', 'down', 'left', 'right', 'right', 'down', 'right', 'down', 'right', 'right', 'up', 'right', 'down', 'up', 'down', 'up', 'right', 'up', 'up', 'left', 'right', 'up', 'up', 'right', 'left', 'up', 'up', 'left', 'up', 'up', 'left', 'right', 'left', 'left', 'up', 'left', 'right', 'left', 'left', 'left', 'left', 'down', 'left', 'left', 'down', 'left', 'left', 'down', 'right', 'down', 'right', 'down', 'right', 'down', 'up', 'down', 'right', 'up', 'right', 'right', 'up', 'right', 'up', 'up', 'left', 'left']
number of moves: 68
   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8
```

3. Demonstrate the correctness of LocalBeam search
   a. initialize BoardNode from one of the preset states (b1, b2, b3).

```
b = BoardNode(b3, None, None, 0)
```

   b. randomize state (this line may be omitted if you would like to run local beam on one of the default states.

```
b.randomize_state(n=20)
```

Output should be:
['right', 'right', 'left', 'down', 'up', 'left', 'down', 'right', 'right', 'up', 'left', 'down', 'up', 'right', 'left', 'right', 'down', 'down', 'up', 'down']

   c. initialize and run local beam search (max_nodes=10000, k=10)

```
search = LocalBeam(b, 10, 10000)
search.solve_beam()
```

Output should be:
initial board state:

```
   0  1  2
0  3  1  2
1  4  5  8
2  6  7  0
```

solved!
list of moves: ['up', 'up', 'down', 'down', 'left', 'right', 'up', 'down', 'up', 'down', 'up', 'left', 'left']
number of moves: 13

```
   0  1  2
0  0  1  2
1  3  4  5
2  6  7  8
```

4. Cases where algorithm fails:
   a. depending on the amount of randomization and the max_nodes setting, either of these algorithms may reach their limits of explored state space. In the event of this happening, the following statement will be reached:

```
if self.get_max_nodes() <= self.get_unique().current:
    raise IndexError('max nodes reached.')
```

   b. In LocalBeam, if k is set to be too large, python will attempt to create an array that does not fit in memory. This should be avoided depending on the amount of memory available.

Experimentation

1. variance in fraction of solvable states with respect to max_nodes

I ran the following method:

```
def frac_solvable_puzzles(search):
    max_nodes_domain = [1000, 10000, 100000, 1000000]
    frac_solvable_states = np.zeros(len(max_nodes_domain),
dtype=float)

    b = BoardNode([[],[],[]], None, None, 0)
    for n in range(len(max_nodes_domain)):
        max_nodes = max_nodes_domain[n]
        exceptions_caught = 0
        for i in range(50):
            b.randomize_state(n=100)

            if search == 'a_star':
                algorithm = AStar(b, 'h1', max_nodes=max_nodes)
                try:
                    algorithm.solve_a_star()
                except IndexError:
                    exceptions_caught += 1
            elif search == 'local_beam':
                algorithm = LocalBeam(b, 10, max_nodes=max_nodes)
                try:
                    algorithm.solve_beam()
                except IndexError:
                    exceptions_caught += 1

        frac_solvable_states[n] = float(50 -
exceptions_caught)/float(50)

    return frac_solvable_states
```

And got the following output:

array([0.22, 0.66, 1. , 1.])

for max_nodes values of (1000, 10000, 100000, 1000000), respectively.

2. h1 vs h2

I ran the following method:

```
def h1_vs_h2():
    num_moves = dict(h1=[], h2=[])

    b = BoardNode([], None, None, 0)
    for h in ['h1', 'h2']:
        for i in range(50):
            b.randomize_state(n=20)
            search = AStar(b, h, max_nodes=100000)
            search.solve_a_star()
```

```
                num_moves[h].append(len(search.get_moves()))

    return np.mean(num_moves['h1']), np.mean(num_moves['h2'])
```

Which returned h1_average = 32.62, h2_average=10.62

3. How does the solution length vary across the three methods?

I ran the following method

```
def var_sol_length():
    num_moves = dict(h1=[], h2=[], beam=[])

    b = BoardNode([], None, None, 0)
    for mode in ['h1', 'h2', 'beam']:
        for i in range(50):
            b.randomize_state(n=20)
            if mode == 'h1' or mode == 'h2':
                search = AStar(b, mode, max_nodes=100000)
                search.solve_a_star()
                num_moves[mode].append(len(search.get_moves()))
            elif mode == 'beam':
                search = LocalBeam(b, 10, 100000)
                search.solve_beam()
                num_moves[mode].append(len(search.get_moves()[0]))

    h1_var = np.std(num_moves['h1'])
    h2_var = np.std(num_moves['h2'])
    beam_var = np.std(num_moves['beam'])

    return h1_std, h2_std, beam_std
```

Which returned (44.61535610078665, 19.897738564972656, 2.330750951946604) respectively.

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

This depends on the value of max_nodes. Assuming that max_nodes is small enough to be reached regularly, say 1000, I ran the following method:

```
def compare_solvability(max_nodes_lim):
    frac_solvable_states = np.zeros(3)
    b = BoardNode([], None, None, 0)
    max_nodes = max_nodes_lim
    for n in range(3):
        exceptions_caught = 0
        for i in range(50):
            b.randomize_state(100)

            if n == 0:
                algorithm = AStar(b,'h1', max_nodes=max_nodes)
```

```
                try:
                    algorithm.solve_a_star()
                except IndexError:
                    exceptions_caught += 1
            elif n == 1:
                algorithm = AStar(b, 'h2', max_nodes=max_nodes)
                try:
                    algorithm.solve_a_star()
                except IndexError:
                    exceptions_caught += 1
            elif n == 3:
                algorithm = LocalBeam(b, 10, max_nodes=max_nodes)
                try:
                    algorithm.solve_beam()
                except IndexError:
                    exceptions_caught += 1
        frac_solvable_states[n] = float(50 -
exceptions_caught)/float(50)
    return frac_solvable_states
```

Which returned array([0.26, 0.64, 1.]) for h1, h2, and beam respectively.

Discussion

a. In conclusion from my experimentation, beam search is best-suited for this problem. Beam had the least standard deviation in its solution-lengths, likely due to the characteristic of being able to select from its 10 best children states. Because of this, it can better minimize its cost function, which was defined as the path-cost + the h1 heuristic. If Beam used the h2 heuristic, it might perform even better. Additionally, Beam reached max_nodes less frequently than the other two algorithms overall.
b. I found it difficult to hold a high enough standard of documentation. Due to the time commitments of writing, testing, evaluating methods, I was not able to apply docstring comments as frequently as I would have liked. I also found it difficult to navigate through the problem space following my decision to use classes and not methods for the search algorithms. In retrospect though, I think the choice to use classes was advantageous both organizationally and experimentally. Experimentally because I was able to pull field values from the search objects for statistical purposes. Had I used methods for these algorithms, I would have had to return many more values so that I would have access to all the relevant data.