John McCormick
jcm258
CSDS 391: Programming Project 1 Writeup
Due: September 26th, 2023 18:00:00

Code Design:

The codebase is primarily split into 3 classes and 4 files:

- EightPuzzle (EightPuzzle.py) class
  - This class contains all of the essential eight puzzle functions.
  - Documentation for the entire class can be found at Appendix A.
  - This class holds the puzzle state, as well as processes the moves, the validity of moves, state changes, and heuristic evaluation
- GraphSearch class (search.py)
  - This class contains the main search functionality
  - Documentation for the entire class can be found at Appendix B
  - This class holds the methods for A* search and beam search, and does so independently, taking advantage of the modularity of the code, meaning with just a couple type changes it could be altered to process a different puzzle
- SearchNode class (EightSearchNode.py)
  - This class contains the node class for all the graph searches.
  - Documentation for the entire class can be found at Appendix C
  - This class holds the functionality for getting the moves and getting the path of nodes. It stores the puzzle with the current state, as well as the parent node, and the move that was taken to get to this node. It also holds the current cost (the number of levels from the root node) and the heuristic value for the current state.
  - It also includes overrides of the __eq__ method, the __lt__ method, and the __hash__ method to enable better hashing for the A* method.
- main.py
  - This python script is the main way of interacting with the program through the command line. It contains the main functionality to read the file as a command line arg with the relative file path, as well as the functionality to parse commands from text, and print the move path.
  - All txt file testing of the program should be done with the following format:

```
python main.py EXAMPLE.txt
```

The decision to split up the program into these 3 classes and 1 main script was to increase the modularity and the adaptability of the code base. The GraphSearch class can basically do graph search on any puzzle type with just a type change in the inputs for the puzzle, and a new node class that contains the new puzzle. So long as the new puzzle has the same functionality calls as EightPuzzle does, graph search should work on the new puzzle. This

modularity helps make the search functions more useful in broader applications. It also allows for efficient distribution of work by allowing the puzzle to do all the puzzle moves and calculation, the search class to do all of the searching, and the node class to hold the data, and find its path.

A unique quirk to my code base is that because I do all the actions between 3 classes, and because my code works for both b and 0 as the blank square, using the priority queue required me to override the __eq__, __lt__ and __hash__ functions within my node class, and create a separate method to get a standardized "hash" string that I could use to rate equality between puzzles.

The hash string method within EightPuzzle

```
def getHashString(self) -> str:
    """
    Returns a string representing the current state of the puzzle that is standardized to 0 as the blank square for hashing purposes
    args: None
    returns: a string representing the current state of the puzzle with 0 as the blank square
    """
    string = self.getStateString()
    string = string.replace("b", "0")
    return string
```

The overridden __eq__ __lt__ and __hash__ functions in Search Node:

```
def __eq__(self, other):
    """
    Checks if two nodes are equal (Overriding the default method for == so that the hash set works properly)
    args: other - the other node to compare to
    returns: True if the nodes are equal, False otherwise
    """
    if isinstance(other, SearchNode):
        return self.state.getHashString() == other.state.getHashString()
    else:
        return NotImplemented
```

```
def __lt__(self, other):
    """
    Checks if one node is less than another, it uses total cost as the comparison
    Used for the priority queue as it defaults to comparing the objects in the case of cost ties
    args: other - the other node to compare to
    returns: True if the node is less than the other, False otherwise
    """
    if isinstance(other, SearchNode):
        return self.totalCost < other.totalCost
    else:
        return NotImplemented
```

```
def __hash__(self):
    """
    Hashes the node (Overriding the default hash method so that the hash set works properly)
    args: None
    returns: the hash of the node
    """
    return hash(self.state.getHashString())
```

Furthermore I chose to implement beam search using h2 as the heuristic function. This was because I wanted to prioritize getting closer to the goal state in terms of number of moves.I didn't want the search to prioritize lessening misplaced tiles over lessening the overall number of moves away from the goal state the current state was.

Code Correctness:

To evaluate correctness I used two different methods. I tested using txt file input, and I used pytest unit tests to verify that key functionality was working within the individual classes. This unit testing helped me isolate where problems may or may not have been.

Note to grader: to use pytest, you must pip install pytest, and run the file with: "pytest test.py"

All tests passed in the pytest.

There were 2 tests for the txt files. The first test included many commands that tested the ability for the main.py to parse moves, printing commands, as well as handle both b and 0. It also randomized the state with a seed arg of 7. It then did various methods of solving the puzzle.

The second test included examples unsolvable to test that failed tests would not crash and would output that the test indeed failed.

```
~/githubFolder/8-PuzzleSolver git:(main) ±20 (0.438s)
python3 main.py test2.txt

No solution found.
Nodes expanded: 11003
No solution found.
Nodes expanded: 12
```

```
~/githubFolder/8-PuzzleSolver git:(main) ±20 (0.42s)
python3 main.py test1.txt
Moves: 0
Nodes expanded: 1

b 1 2
3 4 5
6 7 8


3 1 2
b 4 5
6 7 8


3 1 2
6 4 5
b 7 8


Moves: 2
Nodes expanded: 3
up, up
Moves: 2
Nodes expanded: 4
up, up
6 3 2
1 5 b
7 4 8


Moves: 9
Nodes expanded: 23
left, left, up, right, down, down, left, up, up
0 1 2
3 4 5
6 7 8


Moves: 0
Nodes expanded: 1

3 1 2
0 4 5
6 7 8


3 5 8
4 1 0
6 7 2


Moves: 15
Nodes expanded: 670
down, left, up, right, up, left, down, down, right, up, up, left, down, left, up
No solution found.
Nodes expanded: 10013
```

Experiments:
Data Collected from running 10 terminal instances in which for every variation below 100 random trials were conducted and then these 10 instances had their data aggregated into the tables below:

A Search using H1 heuristic:*

A* Search with h1 heuristic							
Max Nodes	total time	successes	failures	moves	success rate	time per iteration	avg moves per solution
1	0.1433353424	0	1000	0	0	0.0001433353424	#DIV/0!
10	0.8891336918	1	999	7	0.1	0.0008891336918	7
100	9.298126221	12	988	122	1.2	0.009298126221	10.16666667
1000	102.0878985	97	903	1469	9.7	0.1020878985	15.1443299
10000	831.712528	512	488	9990	51.2	0.831712528	19.51171875
max integer (2^63-1)	1739.808601	1000	0	22040	100	1.739808601	22.04

A Search using h2 heuristic:*

A* Search with h2 Heuristic							
Max Nodes	total time	successes	failures	moves	success rate	time per iteration	avg moves per solution
1	0.1691703796	0	1000	0	0	0.0001691703796	#DIV/0!
10	0.9832522869	2	998	17	0.2	0.0009832522869	8.5
100	9.740783691	64	936	981	6.4	0.009740783691	15.328125
1000	73.04917789	617	383	12532	61.7	0.07304917789	20.31118314
10000	131.6484439	997	3	21869	99.7	0.1316484439	21.93480441
max integer	137.60815	1000	0	22049	100	0.1376081	22.049

(2^63-1)	17					517	

Disclaimer: For beam search, there was an error in the implementation at the time of the parallel computing test, therefore once this implementation was fixed, to preserve computing power and time, only 1 batch was done, so the results are out of 100, although the trends continue to hold meaningful weight despite the smaller sample size.

Beam Search k = 1							
Max Nodes	total time	successes	failures	moves	success rate	time per iteration	avg moves per solution
1	0.005436182022	0	100	0	0	0.00005436182022	#DIV/0!
10	0.0290312767	0	100	0	0	0.00029031276 7	#DIV/0!
100	0.2801709175	14	86	480	14	0.00280170 9175	34.28571429
1000	2.238912106	27	73	1263	27	0.02238912106	46.77777778
10000	22.49982595	25	75	1207	25	0.2249982595	48.28
100000	229.4604411	25	75	1413	25	2.294604411	56.52

Beam Search k = 5							
Max Nodes	total time	successes	failures	moves	success rate	time per iteration	avg moves per solution
1	0.005150079727	0	100	0	0	0.00005150079727	#DIV/0!
10	0.0436248779296875`	0	100	0	0	#VALUE!	#DIV/0!
100	0.3103988171	1	99	13	1	0.00310398 8171	13
1000	2.032927036	48	52	1852	48	0.02032927036	38.58333333
10000	12.94967389	59	41	2241	59	0.1294967389	37.98305085

100000	149.1818409	51	49	2079	51	1.491818409	40.76470588

Beam Search k = 10							
Max Nodes	total time	successes	failures	moves	success rate	time per iteration	avg moves per solution
1	0.005163908005	0	100	0	0	0.00005163908005	#DIV/0!
10	0.04267120361	0	100	0	0	0.0004267120361	#DIV/0!
100	0.3182690144	0	100	0	0	0.003182690144	#DIV/0!
1000	2.164733171	55	45	1698	55	0.02164733171	30.87272727
10000	11.02683377	67	33	2200	67	0.1102683377	32.8358209
100000	101.0618649	67	33	2106	67	1.010618649	31.43283582

Beam Search k = 100							
Max Nodes	total time	successes	failures	moves	success rate	time per iteration	avg moves per solution
1	0.005101919174	0	100	0	0	0.00005101919174	#DIV/0!
10	0.04154610634	0	100	0	0	0.0004154610634	#DIV/0!
100	0.397217989	1	99	6	1	0.003972177989	6
1000	3.272692919	2	98	23	2	0.03272692919	11.5
10000	8.761699915	99	1	2300	99	0.08761699915	23.23232323
100000	9.258594036	100	0	2459	100	0.09258594036	24.59

1. How does the fraction of solvable puzzles from random initial states vary with the maxNodes limit?

For all the three methods, in general, as maxNodes increases as does the share of puzzles that can be solved. However this does seem to at times have a lesser effect with the beam search, as the k value seems to be a larger constraint. But for both heuristics in A* search, there was an increase in the percentage of puzzles solved as maxNodes increased. This makes sense as the longer A* is allowed to search for the more solutions it will find. This is seen with the increase in average moves per solution as the A* algorithm is able to dig deeper. Essentially maxNodes increasing increases the depth to which A* can search and longer solutions are then unlocked. However with beam search the opposite seems to be true. While yes as maxNodes increases so too does the percentage of puzzles solved, as k values increase average move number decreases along with the increase in problems solved. This suggests that k value is more important to the search algorithm as it is able to search down more branches thus having a better chance of finding the shortest path. So maxNodes helps beam search solve more, but ultimately the larger impact is on the efficacy of the k value for beam search.

2. For A* search, which heuristic is better?

The Manhattan distance heuristic (or h2) is better. This is seen as the average time per search is significantly less at higher maxNode values. But at the same maxNode values, bar the maximum and 1, h2 solves a larger proportion of the puzzles than h1 does, suggesting that the Manhattan distance heuristic is a better performing heuristic function for A* search.

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

For A* Search there is an increase in average solution length associated with an increase in maxNode value. This makes sense as A* getting access to more nodes allows A* to search deeper and find longer paths. However for beam search while that trend does exist, a much more significant trend is an increase in k leading to an overall decrease in average solution length. This too makes sense, as since beam search is based on BFS, increasing the number of nodes to evaluate at any given level increases the odds that the shortest path is found. That is because sometimes the shortest path includes short term negative heuristic gain to set up larger gains in later moves. Beam search cannot consider these as it will only expand the top k moves in terms of short term gain. Unlike A* which will expand the lowest total cost, so it will come back to ideal paths eventually.

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

For A* Search using h1, the average percent of problems that were solvable was 27%. For A* Search using h2, the average percent of problems that were solvable was 44.7%. For

beam search with k = 1, it was 15.2%, for k -5 it was 26.5%, for k = 10 it was 31.5%, and for k=100 it was 33.7%. Based on these results it can be collected that A* search with h2 as a heuristic was the best performing search method. Especially considering that unbounded searches took an average time of .13 seconds to complete while solving 100% of puzzles it makes sense that this solution would be best. Although 100,000 maxNodes beam search with k = 100 yielded similar results faster, although with longer average paths.

Discussion:

Based on my experiments I would say that A* search using h2 was the most optimal considering the smaller average solution length. However beam search using 100 as k and 100,000 as maxNodes, performed faster. Because of this I would probably be inclined to use beam search with 100 as k if I had spatial constraints. This is because the algorithm only saves 100 nodes at a time, compared to A* which holds all visited nodes and unexplored nodes. All the nodes do store history, so this isn't a differentiator between the two. But it does seem that A* always found the optimal solution when unbounded and using h2 as a heuristic.

I had difficulty implementing beam search especially without the visited hash set that A* has. I was for a long time not preventing infinite loops from forming, and it took me some time to figure out exactly how I should have done it. Furthermore testing was difficult because of the temporal and computational complexity. I ran 10 simultaneous tests of all methods, and it took 18 minutes each on 98% CPU usage just to do 100 iterations of each method and parameters. This major resource used in getting the metrics used to do the analysis was difficult to work with at times.

Appendix starts next page:

Appendix A: EightPuzzle

index
/Users/johnmccormick/githubFolder/8-PuzzleSolver/EightPuzzle.py

```
## John McCormick
## jcm258
## CSDS 391: Programming Assignment 1
## Due: September 26th, 2023 18:00:00
```

Modules

random

Classes

builtins.object

EightPuzzle

class **EightPuzzle**(builtins.object)

```
EightPuzzle(state=None)

A class representing an 8-puzzle
```

Methods defined here:

__init__(self, state=None)
```
Constructor for the EightPuzzle class
args: state - (An optional parameter) a string representing the initial state of the puzzle Must be of the same format as setState takes ("xxx xxx xxx")
returns: None
```

calculate_h1(self) -> int
```
Calculates the heuristic h1 value for a given state of the 8-puzzle (this calculates the current state)
h1 is the number of misplaced tiles
args: None
returns: an int representing the heuristic value of the state
```

calculate_h2(self) -> int
```
Calculates the heuristic h2 value for a given state of the 8-puzzle (this calculates the current state)
h2 is the manhattan distance, or the sum of the distances of each tile from its goal position
args: None
returns: an int representing the heuristic value of the state
```

findBlank(self) -> tuple
```
Method to find the blank square in the puzzle board
args: None
returns: tuple representing the location of the blank square
```

getHashString(self) -> str
```
Returns a string representing the current state of the puzzle that is standardized to 0 as the blank square for hashing purposes
args: None
returns: a string representing the current state of the puzzle with 0 as the blank square
```

getHeuristic(self, hueristic: str) -> int
```
Returns the value of a given state according to a heuristic function specified by the user for the 8-puzzle
args: heuristic - a string representing the heuristic function to use (in this case h1 or h2)
returns: a int representing the value of the state according to the heuristic function
```

getStateString(self) -> str
```
Returns a string representing the current state of the puzzle
args: None
returns: a string representing the current state of the puzzle (in the xxx xxx xxx format)
```

isGoal(self) -> bool
```
Checks if the current state is the goal state
args: None
returns: True if the current state is the goal state, False otherwise
```

move(self, direction: str)
```
Makes a move in the desired direction and updates the gameboard
args: direction - a string representing the direction to move the blank square
returns: None
```

printState(self)
```
Prints the current state of the puzzle.
args: None
returns: None
```

randomizeState(self, n: int, seed: int = None)
```
Randomizes the state of the puzzle by making n random moves
args: n - the number of random moves to make
        seed - (optional) a seed for the random number generator (for standardizing testing procedures)
returns: None
```

setState(self, state)
```
Sets the state of the puzzle to the given state.
args: state- a string representing the puzzle state (9 numbers in groups of 3, 0 for blank)
returns: None
```

validMoves(self) -> list
```
Returns a list of valid directional moves for the blank square
args: None
returns: list of strings representing valid moves
```

Data descriptors defined here:

__dict__
```
dictionary for instance variables (if defined)
```

__weakref__
```
list of weak references to the object (if defined)
```

Data and other attributes defined here:

goalState0 = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]

goalStateb = [['b', 1, 2], [3, 4, 5], [6, 7, 8]]

Appendix B: GraphSearch

index
/Users/johnmccormick/githubFolder/8-PuzzleSolver/search.py

```
## John McCormick
## jcm258
## CSDS 391: Programming Assignment 1
## Due: September 26th, 2023 18:00:00
```

Modules

heapq

Classes

builtins.object

GraphSearch

class **GraphSearch**(builtins.object)

```
This class holds all of the graph searches for any puzzles given certain methods within the puzzle class itself.
```

Methods defined here:

__init__(self)
```
Constructor for the GraphSearch class
args: None
returns: None
```

maxNodes(self, nodes: int)
```
Sets a maximum number of nodes to be considered during a search
args: nodes - the maximum number of nodes to be considered
returns: None
```

solveAStar(self, puzzle: EightPuzzle.EightPuzzle, heuristicFunction: str) -> tuple
```
Solves the given puzzle using A* search
args: puzzle - the puzzle to be solved, heuristicFunction - a string representing the heuristic function to be used (an arg to the heuristic evaluation function)
returns: a tuple with the number of moves needed, the path of moves, and the number of nodes expanded
```

solveBeam(self, puzzle: EightPuzzle.EightPuzzle, k: int) -> tuple
```
Solves the given puzzle using beam search
args: puzzle - the puzzle to be solved, k - the number of nodes to be considered at each level
returns: a tuple with the number of moves needed, the path of moves, and the number of nodes expanded
```

Data descriptors defined here:

__dict__
```
dictionary for instance variables (if defined)
```

__weakref__
```
list of weak references to the object (if defined)
```

Appendix C: EightSearchNode

index
/Users/johnmccormick/githubFolder/8-PuzzleSolver/EightSearchNode.py

```
## John McCormick
## jcm258
## CSDS 391: Programming Assignment 1
## Due: September 26th, 2023 18:00:00
```

Classes

builtins.object

SearchNode

class **SearchNode**(builtins.object)

```
SearchNode(parent=None, state: EightPuzzle.EightPuzzle = None, heuristicFunction: str = None, move=None)

Class for a search node that holds itself and its child nodes
```

Methods defined here:

__eq__(self, other)

```
Checks if two nodes are equal (Overriding the default method for == so that the hash set works properly)
args: other - the other node to compare to
returns: True if the nodes are equal, False otherwise
```

__hash__(self)

```
Hashes the node (Overriding the default hash method so that the hash set works properly)
args: None
returns: the hash of the node
```

__init__(self, parent=None, state: EightPuzzle.EightPuzzle = None, heuristicFunction: str = None, move=None)

```
Constructor for the SearchNode class
args: parent - the parent node of the node, state - A puzzle object with the board position of the node
returns: None
```

__lt__(self, other)

```
Checks if one node is less than another, it uses total cost as the comparison
Used for the priority queue as it defaults to comparing the objects in the case of cost ties
args: other - the other node to compare to
returns: True if the node is less than the other, False otherwise
```

getMove(self) -> str

```
Gets the move that was made to reach the current node
args: None
returns: a string representing the move that was made to reach the current node
```

getPath(self) -> list

```
Gets the path from the root node to the current node
args: None
returns: a list of the moves from the root node to the current node
```

Data descriptors defined here:

__dict__

```
dictionary for instance variables (if defined)
```

__weakref__

```
list of weak references to the object (if defined)
```