John Pitkin

CS 420

Professor Pfaffmann

September 28, 2022

Project 1: A* Puzzle Solver Report

Software organization:

The first step of creating my project was designing a Puzzle class that encapsulated all the attributes and methods needed for an individual Puzzle instance. Each Puzzle saves the puzzle board state in the form of two-dimensional python lists where each row is indexed to a list and each column is indexed within each element of a list. In other words, the number of columns is the number of elements in a list and the number of rows is equal to the number of inner lists. Each Puzzle instance saves the size of the puzzle equivalent to the number of rows or the number of columns, since each puzzle board is square. Lastly, each Puzzle instance saves the location of the empty tile within the puzzle board state in the form of a dictionary with two data values. The first value has the key string “row” and stores the row position of the empty tile, and the second value has the key string “column” and stores the column position of the empty tile. These three Puzzle attributes are all the data needed for describing a Puzzle instance.

The methods in my Puzzle class include a Python specific method for describing the string representation of a Puzzle instance; basic getters for returning the puzzle state, the location of the empty tile, the value of a tile at a specified location, and the puzzle size; setter methods for the location of the empty tile, and for setting the puzzle state of a Puzzle instance;

a method that turns a Puzzle instance into a one dimensional list; a method that searches a puzzle state for the location of the empty tile; a method that swaps tiles and updates the location of the empty tile if needed; a method that checks if two Puzzles are equal; a method that checks if a puzzle has been solved; a method that sets the puzzle board state of a Puzzle as solved; a method that tells if all of the Puzzle tiles are out of their original place; a method that counts the number of tiles out of place; and a method for generating all the neighboring puzzle states of a given Puzzle.

I created a file for unit testing some Puzzle methods to ensure that my code was working properly. This included using the unittest class.

The second step in developing my project was creating a file with all the functions needed for generating a csv file full of shuffled puzzle states. The first function generates a list of random integers between zero and three, inclusive. My next function uses this random list to shuffle a given Puzzle's puzzle state where zero represents moving the empty tile up, one represents moving the empty tile to the right, two represents moving the tile down, and three represents moving the tile to the left. This function only moves the empty tile in an available direction. My next function takes a Puzzle and a list of Puzzles and checks to see if the Puzzle's puzzle state exists within the list of Puzzles. I created a function that generates a list of shuffled Puzzles of a given size and quantity. This function ensures that every Puzzle put into the returned list is unique relative to the other Puzzles in the list and has all its tiles out of place relative to their correct positions. I had to create a similar function for generating partially shuffled boards that checks to see if at least eleven tiles are out of place and that the puzzle state is unique in its list. I used this function for generating 15-puzzles since fully shuffled puzzles of this size took too long to solve. Lastly, I have two functions for putting this list of Puzzles' states into a csv file, one which writes and the other which appends. I finish my generator file with some lines of Python code for generating shuffled Puzzles and putting their states into a csv file called config_file.csv.

The third step in my project was making the solution file that contained the classes and functions for solving Puzzles with A*. At the start of the file, I have a Node class that contains the attributes and methods needed for creating an A* Node. The constructor provides each Node instance with a reference to a Puzzle object, a value for the path cost of the Node (the g(n) value), and a reference to a parent Node. Next, I have a method for representing a Node as a string which prints out the puzzle state and the Node's path cost. Lastly, I have a method that takes a heuristic function as an argument and returns the f(n) value for a given Node based off its path cost (g(n)) and the heuristic (h(n)) value for the Node's puzzle state.

The next function in my solution file is a function that takes a list of values and creates an appropriately sized Puzzle with the tiles set according to the order of the values in the list. This function is used for taking a list of integers from the csv config file and creating a Puzzle from them. By taking the root of the list length, we can get the new Puzzle's size. Next, I created a function that takes in the name of a config file and creates a Puzzle list from all the puzzle states represented as comma separated values within that file by using the aforementioned function.

I have my first heuristic function that takes a Puzzle and returns the heuristic value for the number of tiles out of place. Next, I have my helper function for the second heuristic function that takes a tile value and a Puzzle size and returns the correct row and column location for that tile. Using this function, my second heuristic, the Manhattan distance, is defined in a function that takes a Puzzle as its argument. I have my third heuristic function that computes the number of swaps in the Maxsort of a one dimensional Puzzle state. And finally, I have my fourth heuristic function that computes the Euclidean distance of all the tiles of a Puzzle state.

The second half of my solution file begins with a PriorityQueueHeap class that contains all the attributes and methods needed for implementing the frontier of the A* algorithm. Initially, I implemented the frontier using a Python list; however, I found that A* was taking far too long since it was linearly searching the whole Python frontier list for the Node with the Puzzle state with the minimum f value every time a Node was to be expanded from the frontier. The priority queue is stored as a minimum binary heap that speeds up search significantly to O(log n) instead of O(n) from the original list implementation. The constructor assigns the PriorityQueueHeap instance with an empty list of elements, the size of the queue starting at 0, and with a reference to the heuristic used for calculating f values in this heap. The first method outside of the constructor is a swap method that swaps two elements in the underlying list. Next, there is a function for generating the index of a parent node using the index of the current Node within the list. Additionally, there are methods for getting the indices of left and right children Nodes within the Python list as well. There is a very simple method that returns if the queue is empty. I created methods for bubbling up Nodes to their correct location within the heap and for bubbling down Nodes into their correct location. Using the bubble methods, I have a method for pushing a Node onto the priority queue/heap and a method for popping Nodes.

I complete my solution file with a method for expanding the frontier by taking a parent Node as an argument and returning a list of child Nodes, and with a method that implements A* and prints important information about the performance of the implementation. I created a file for testing some solution functions, test_solutions.py.

I made four files, run_h1.py, run_h2.py, run_h3.py, and run_h4.py for running A* with the four different heuristics and for printing experiments to different log files through the Makefile.

User manual:

Just simply use one of the seven different functions of the Makefile to control the different aspects of my project. “make generate” will generate Puzzles and populate the config file with puzzle states. Simply by changing the random seed at the top of the generator file, generator.py, you can get a completely different set of generated shuffled puzzle states. The generator will create five 2x2 Puzzles, twenty 3x3 Puzzles, and twenty 4x4 Puzzles. Next you can run A* with each of the four different heuristics by typing “make run_h1”, “make run_h2”, “make run_h3”, or “make run_h4”. Each of these commands will populate an individual log file that contains the initial and final states of each solved Puzzle, information about the number of frontier nodes and timings of each Puzzle solution, and totals and averages of timings and Node counts for all the Puzzles solved with each heuristic. The last two commands in the Makefile are ”make test_puzzle” and “make test_solution” that run the test files I created for my project.

Four-Phase Problem Solving process:

Goal formulation:

The goal of this A* agent is to find the optimal solution to 3-puzzles, 8-puzzles, and 15-puzzles.

Problem formulation:

This agent keeps track of Puzzle states and the legal moves between states by considering a simplified version of the problem where we act as though the empty space is the one tile that can “move”. We consider moving from one Puzzle state to the next, so the only fact about our world that will change is the location of the empty tile in each Puzzle, and the tile it swaps with.

Search:

The agent simulates sequences of actions, or empty tile moves, in its model, searching until it finds the optimal sequence of moves that solves the given Puzzle. This is the solution. The agent may consider parts of sequences that do not reach the goal, or are not optimal, but eventually it will find the best solution for any given initial shuffled Puzzle state or return no solution if the Puzzle is unsolvable.

Execution:

By starting at the initial Puzzle state in the root Node and following the shortest path to a solved Puzzle in the search tree, we can execute the least number of moves needed to solve a given Puzzle. The agent can now execute the actions in the solution, one at a time.

Heuristics:

Misplaced tiles:

The misplaced tiles heuristic iterates through a given Puzzle state and counts the number of tiles that are not in their solution state; but ignores the empty tile, which in my case is zero. My implementation of this heuristic compares each current tile to the calculated correct value at each location by using the iterator variables i and j. It returns the number of misplaced tiles.

Manhattan distance:

The Manhattan distance heuristic iterates through the Puzzle state and sums the total number of moves needed for each tile to move from its current location to its correct location if no other tiles could get in the way and block the movement of a tile; but ignores the empty tile. My implementation of this heuristic includes a function that returns the correct row and column location given a tile's value and uses this function to get the Manhattan distance for each tile. I, again, use the i and j iterator variables to calculate this heuristic efficiently for each tile and sum the distances for each tile and return this value. This heuristic performed the best.

Maxsort swaps:

The Maxsort swaps heuristic counts the number of swaps needed to sort a one dimensional version of the Puzzle state by putting the maximum tile into its correct location in the given list and then restricting the list to only consider the unsorted elements in the next iteration. This heuristic sums the number of swaps and required some additional debugging beyond the given pseudocode. This heuristic performed the worst.

Euclidean distance:

The Euclidean distance heuristic iterates through the Puzzle state and sums the total distance for each tile from its current location to its correct location by using the distance formula if no other tiles could get in the way and block the movement of a tile; but ignores the empty tile. My implementation of this heuristic includes a function that returns the correct row and column location given a tile's value and uses this function to get the Euclidean distance for each tile. I, again, use the i and j iterator variables to calculate this heuristic efficiently for each tile and sum the distances for each tile and return this value.

Program performance:

Seed = 0	Total Time (seconds)	Average Time (seconds)	Total Nodes	Average Nodes	Max Number of Nodes
Heuristic 1	12.422635313	0.27605856251	73356	1630.133334	18469
Heuristic 2	0.676094642	0.015024325378	6135	136.3333334	1493
Heuristic 3	40.408267833	0.897961507399	148023	3289.4	49228
Heuristic 4	3.181369673	0.070697103844	11582	257.3777778	2802

Seed = 20	Total Time (seconds)	Average Time (seconds)	Total Nodes	Average Nodes	Max Number of Nodes
Heuristic 1	31.69413552	0.704314122667	162316	3607.022222	50253
Heuristic 2	1.759595576	0.039102123911	14407	320.1555555	2970
Heuristic 3	72.94430217	1.620984492844	272229	6049.533333	63529

Heuristic 4	11.133776008	0.247417244622	31687	704.1555556	9986

Seed = 40	Total Time (seconds)	Average Time (seconds)	Total Nodes	Average Nodes	Max Number of Nodes
Heuristic 1	20.863023231	0.46362273847	112158	2492.4	25352
Heuristic 2	0.9469664400	0.0210436987	8416	187.0222222	1652
Heuristic 3	50.750412610	1.1277869468	193392	4297.6	49228
Heuristic 4	4.970626006	0.1104583556	17227	382.8222222	3563

Using the same set of generated Puzzles for each table, here are the results from using my implementation of the A* algorithm on five 3-puzzles, twenty 8-puzzles, and twenty 15-puzzles per table. Heuristic 2, Manhattan distance, did the best in all categories, then heuristic 4, Euclidean distance, then heuristic 1, misplaced tiles, and lastly heuristic 3, the Maxsort swaps.

Summary:

The Manhattan distance heuristic performed the best of all the given heuristics for this A* informed search problem on different sized tile puzzles. As an *admissible* heuristic, one that never overestimates the cost to reach a goal, this heuristic allowed A* to be cost-optimal. If no tile is in the way between a tile's current location and its correct location, then the Manhattan distance will be equal to the actual path distance between that puzzle state and the solution state. Otherwise, the Manhattan distance will always be an underestimate, as other tiles will need to be moved; thus, generating more states and a longer solution path. Generally, it is better to use a heuristic function with higher values, provided it is consistent and that the computation time for the heuristic is not too long. The Manhattan distance is consistently equal or greater than all the other heuristics (unless ties are broken unluckily) and is relatively quick to compute. For these reasons, the Manhattan distance dominates all the other heuristics. We can see that for every experiment above, the Manhattan distance outperformed all other heuristics in every category. The data supports my claim.

The second-best heuristic is the Euclidean distance heuristic. Quite like the Manhattan distance heuristic, this heuristic is looking for the distance between a tile's current location and its correct location; however, instead of considering the number of row and column moves, we are considering the Euclidean distance. This heuristic is consistently equal to or smaller than the Manhattan distance, since it is computing a shorter distance that is a straight line as opposed to step like paths. The heuristic is admissible as well, since the number of row and column moves will always be greater than or equal to the straight-line distance. For these reasons, it is dominated by the Manhattan distance heuristic, but it dominates the other two heuristics. This heuristic is second best for all experiments and for every category. The data supports my claim.

The third best heuristic is the number of misplaced tiles heuristic. This heuristic is consistently less than or equal to the Manhattan distance since the number of misplaced tiles will always be equal to or less than the steps needed to move tiles to their correct locations, assuming nothing is in the way. In this same fashion, this heuristic is equal to or less than the Euclidean distance because the number of misplaced tiles will always be less than or equal to the distance misplaced tiles need to move to get to their correct location (the smallest distance for a misplaced tile is 1). Since this heuristic score third in all experiments in all categories, the data supports my claim.

Lastly, the Maxsort number of swaps is the worst heuristic out of the four. Firstly, this heuristic takes more processing than the misplaced tiles heuristic since the puzzle has to be turned into a one-dimensional list and then iterated through with nested loops to sort the whole list. We can see how much longer this heuristic takes in the data. The Maxsort will never swap more than the number of tiles that are out of place, and can swap less than the total number of tiles out of place when a swap incidentally puts a tile in place. The Maxsort has the worst performance for every category and in every experiment. The data supports my claim.