The N-Puzzle With A* and Uniform Cost Search

Contributors: Jon Darius, Rohan Gujral, Jerry Li, Vaneesha Singh, Aditi Thanekar

External Resources:

https://brilliant.org/wiki/a-star-search/

https://www.scaler.com/topics/uniform-cost-search/

https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/

imports: math, numpy, pandas, seaborn, matplotlib, heapq, copy, sys

Our Design:

We settled on using Python for the project due to its readability, support for object oriented programming, as well as relevant libraries. From there, we implemented a Node class to represent the structure of each state of the puzzle. We also created a puzzleProblem class to store the essential parts of the problem like the frontier, initial and goal states (Nodes), as well as the states we have already seen.

The A* and Uniform Cost Search classes use the Node and puzzleProblem classes to implement their searches. The heuristics, cost, and total cost are stored as an attribute of each Node so that it can be reused for A* and UCS, with the heuristic for each Node of UCS being 0.

In both UCS and A*, the function relies on the frontier created in the puzzleProblem class to store each state. We use a priority queue for our frontier in order to have easy access to the cheapest cost node in the front of the queue each time. We then run a loop to iterate through the frontier until it's empty. For each iteration, we pop off the cheapest node from the front because that is what we want to expand next. The first thing we check is if this node is the goal state, in which case we return the node itself. If not, we ensure to add the node to the seen list to avoid repetition or infinite loops while searching.

After that, we call the operator functions in puzzleProblem to perform any valid operations on the blank tile and assign the new node to be the child node. puzzleProblem contains an expandNode function to check what states are actually valid, since any tiles on the edges of the

puzzle are limited to the number of moves they can make. After this, for UCS, we calculate the cost, which is simply incrementing by 1 for each move the tile makes.

For A*, we additionally calculate the heuristic, using either the Misplaced Tile Heuristic or Euclidean Distance heuristic. Each heuristic is calculated in a separate function to make the code more readable. For Misplaced Tile, we iterate through the puzzle to find how many tiles are not in the correct spot. We sum up the number of misplaced tiles and return this value for the heuristic. For Euclidean Distance, we use the Euclidean distance function to find the distance between where each tile should be located and where it is currently located. We sum up the euclidean distance for every tile and return this value for the heuristic.

Following the calculations for cost (UCS and A*) and heuristic (just A*), we add the child node into the frontier if it has not been explored yet. If any child node has a smaller cost, we update the child node to the new smaller cost. This is important because we want to expand nodes in order of lowest to highest cost for each iteration.

Challenges:

1) Using numPy to represent the actual N-puzzle, every state of the puzzle was stored as a matrix, or 2d array. While testing the functionality of our N-puzzle class, we discovered that some of our operators were not working properly. For example, even if a specific operation was deemed valid, such as moving right, a user was not able to. This issue was fixed after taking a closer look at our node class and detecting some inconsistencies between our uml model and software implementation.

2) Switching from a graph to a tree expansion of states in our actual A* algorithm. Our original plan involved expanding nodes as a graph and updating each node's g(n) value if a shorter path from the root was found. We soon realized that this would require more complicated data structures as well as substantially increase the running time of the algorithm, even if it meant potentially finding a more optimal solution.

3) When taking our high level design and implementing the code, we soon found that much of our design did not adhere to one or more of the SOLID principles of object oriented programming. For instance, we initially violated the single responsibility principle when combining the functionality of the puzzle problem class with the A* class, even though they served two different purposes. In addition, we also went against the open-closed principle when we were not able to add heuristics to our A* class without modifying the underlying code of the N-puzzle class.

4) When testing our UCS and A* algorithms, we ran into a major problem of infinite recursion occurring. This was because when assigning the child node its cost, we accidentally assigned it the parent's cost. This means the child cost was never being updated properly, causing the algorithm to never end. When running UCS on the Oh Boy case and any default case, we ran into the same problem. We realized that our code to update the child cost if we found a smaller cost was quite complex which made the code very slow.

5) It also took some time for some of us to familiarize ourselves with the intricacies of the python programming language. One of the problems we encountered was accidentally treating an object as a primitive type and trying to hash it. We also experienced issues when trying to use the != operator to check if a node was set to a null value instead of using the "None" keyword.

Optimization:

We optimized our code for the A* algorithm and Uniform Cost Search by utilizing a priority or heap queue to represent the frontier of nodes. By using a heap queue we are able to insert new states/nodes and sort them with a runtime of O(logn). This optimizes our runtime by using a regular list, because otherwise we would have to sort the entire array everytime we insert a new state/node which costs O(nlogn). Furthermore we used a hashmap for the seen or closed list. Since we need to store the Node object in seen, we converted the game puzzle into a single string that represents all the values in the gameboard and used this as the key for the hash. This helped

reduce our time complexity as by using a regular list we would have to iterate through it entirely just to check if it existed in seen which would be O(n) runtime. However with our hashmap method we're able to check if a node exists in seen with O(1) runtime.

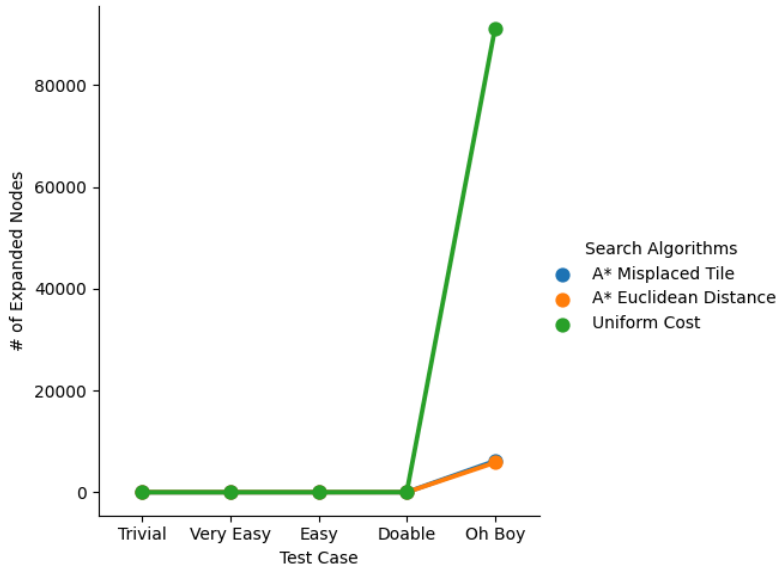
We also optimized the list of seen states, by using a hash of the string of the puzzle to quickly assess whether a state has been seen or not. In addition, we optimized our code when pushing the cheapest child node to the priority queue by using one if statement instead of multiple. This helps to reduce the amount of times we check if the child node is already in the frontier or the seen queue.

Search Implementation:

Rather than implementing a graph search, we implemented a tree search style with a parent and children nodes. We believe that this implementation method is more efficient than a graph one since we only need to keep track of the necessary nodes. Our "tree" structure is represented by our n_puzzle class which manages and contains all the nodes used in the search.

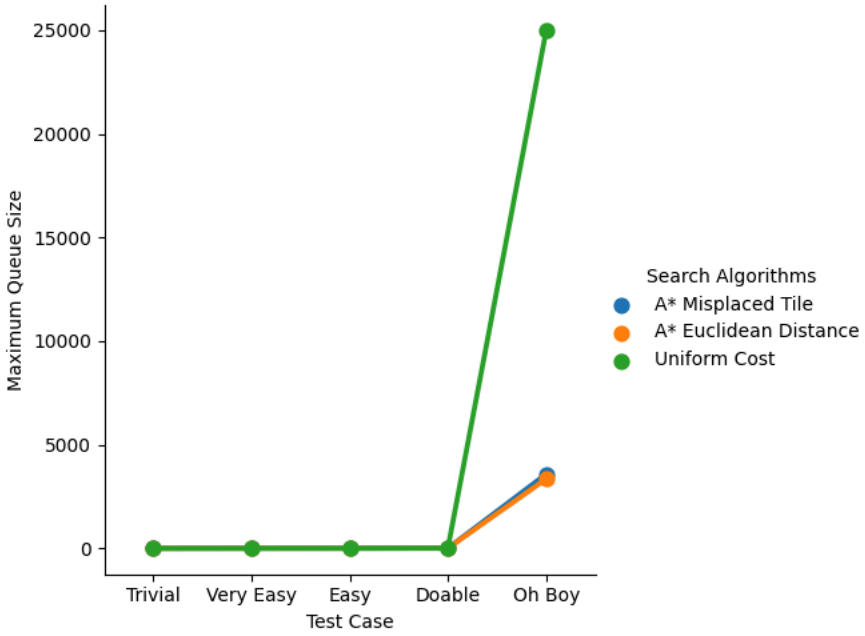
Results:

Number of Expanded Nodes for Increasingly Difficult Cases

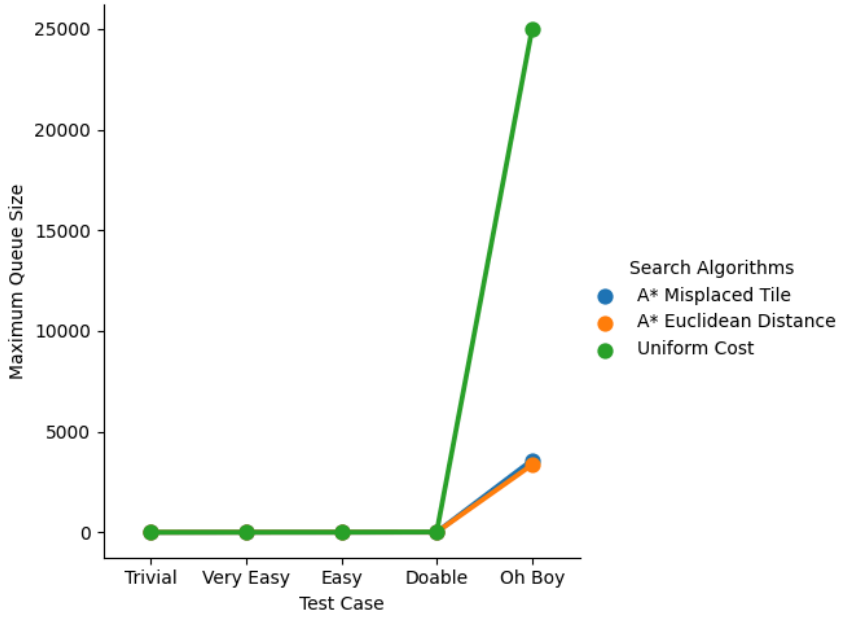


Test Case	A* Misplaced Tile	A* Euclidean Distance	Uniform Cost
Trivial	1	1	1
Very Easy	2	2	2
Easy	3	3	6
Doable	4	5	16
Oh Boy	6153	5832	91051

Maximum Queue Size



Test Case	A* Misplaced Tile	A* Euclidean Distance	Uniform Cost
Trivial	0	0	0
Very Easy	3	3	3
Easy	4	4	6
Doable	7	7	16
Oh Boy	3582	3351	24969



Test Case	A* Misplaced Tile	A* Euclidean Distance	Uniform Cost
Trivial	0	0	0
Very Easy	1	1	1
Easy	2	2	2
Doable	4	4	4
Oh Boy	22	22	22

Data Analysis:

After testing our A star algorithm on a wide variety of test inputs, we can see that the most useful heuristic was the euclidean distance heuristic. Although the exact heuristic used did not make any notable difference for trivial starting states, as the problem becomes more complex, the euclidean distance heuristic more effectively prioritizes states that were closer to the goal state. As can be seen in the graphs above, the amount of nodes expanded with the euclidean distance heuristic is slightly lower than the misplaced tile heuristic, and both are substantially lower than uniform cost search.

Contributions:

- Jon Darius: N_puzzle, creating graphs
- Rohan Gujral: A*, N_puzzle, euclidean distance
- Jerry Li: A*, misplaced tile heuristic
- Vaneesha Singh: Uniform Cost Search
- Aditi Thanekar: N_puzzle