CS170 - Project 1: 8-Puzzle Solver

Pajaka Lakshmin, Adelyn Lampley, Parnika Nammi, Anisha Siva

5 May 2024

1 Project Design

For this 8 puzzle project and designing our A* Puzzle Solver, we chose Python as the programming language to take advantage of its libraries which optimize the management of different data types and structures, such as matrices and trees. Additionally, we adopted an object-oriented design approach, creating objects for menus, puzzle boards, states, operators, and our search space.

In our menus, we implemented a simple console UI to help the user navigate through different features of our program, such as providing their own test cases or choosing a default, along with choosing the heuristic they would like to implement to solve their chosen puzzle. With this, we have included some general input validations, ensuring that inputs given from the user create a valid puzzle board, and that valid heuristic options are chosen. For example, once they have chosen to make a puzzle or use a given one, depending on their input of 1,2, or 3, the test would be done in UCS, missing tile, or Euclidean.

To run the search algorithms, we developed a State class, which stored 3x3 matrices as NumPy arrays, allowing us to easily manipulate them between states when we would later need to hash them onto maps. This class contains simple mutator functions, as well as all the operators we used in this problem space, which includes moving the blank tile space in all four cardinal directions.

Lastly, we implemented an Algorithms class to encapsulate all search algorithms for our puzzle solver, accompanied by essential helper functions. Each algorithm, based on the generic search algorithm using a priority queue, was designed to be extensible and adaptable to different puzzle initial states, goal states, and even sizes. This adaptability is achieved by parameterizing the initial and goal states, and not hard-coding dependencies on its 3x3 shape into the generic search algorithm. A key feature of these algorithms is the implementation of a parent map, which records the relationship between states during the search process. This mapping is important in efficiently tracing back the optimal path from the initial state to the goal state, allowing us to output a single path from the initial state to the goal state, even if the nodes expanded are in the thousands.

2 Challenges Presented and Solutions Found

While developing the program we ran into some issues and complications throughout multiple steps of the project. Firstly, many of us are fairly new in Python, so there was some trouble in first thinking how we wanted to implement the program. After getting comfortable with the libraries, we were able to make some substantial progress with the program, having a working generic algorithm running.

Our initial attempt at printing the order of the nodes was unsuccessful, and we were actually printing out the trace of the search, which was not our intention. This is when we started to implement the parent map to help connect the states to each other. Luckily, this was easier to implement than we thought it would be, but led us to the next challenge we ran into: keeping track of the g(n) and h(n) values at each node. At this point, we were printing the search path in a separate function from the search, and we didn't have access to the depth from outside the function. So we changed the location of the prints to be inside the search functions. Ideally, to make the code cleaner, we would have a helper function to print the paths so each function has a single responsibility.

Lastly, we ran into some issues with the heuristics, before we realized that both of our heuristics were containing the blank tile in its calculation of the missing tiles and the sum of Euclidean distances. When we tweaked the heuristics to remove the blank tile, things were working as expected.

3 Data Structures Utilized

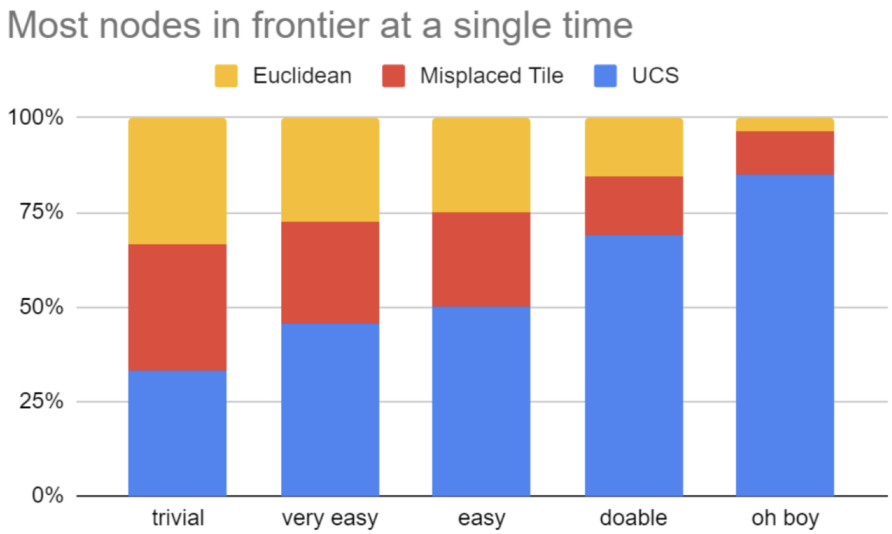
To complete our puzzle solver, we used a number of data structures, including NumPy arrays, priorities queues, and hash-maps to create a rudimentary tree structure to keep track of the depths, explored nodes, and parent/child connections.

4 Test Cases Utilized

The test cases we utilized for testing and comparing our different search algorithms are the ones provided specifications file. When developing our code, we also included a variety of simpler test cases to make sure our operators were working as intended (which are not shown below).

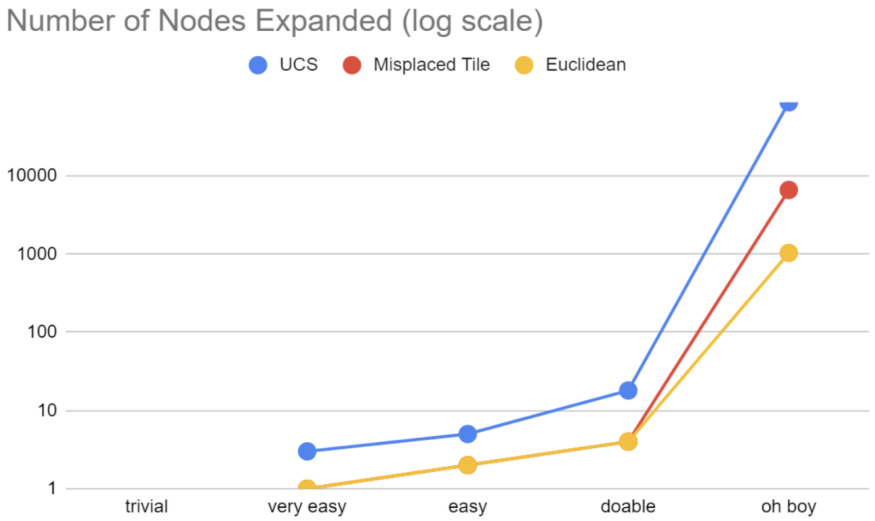
Trival	Easy	Oh Boy
1 2 3 4 5 6 7 8 *	1 2 * 4 5 3 7 8 6	8 7 1 6 * 2 5 4 3
		IMPOSSIBLE:
Very Easy	**doable**	
1 2 3 4 5 6 7 * 8	* 1 2 4 5 3 7 8 6	1 2 3 4 5 6 8 7 *

5 Comparisons Between Heuristic Functions



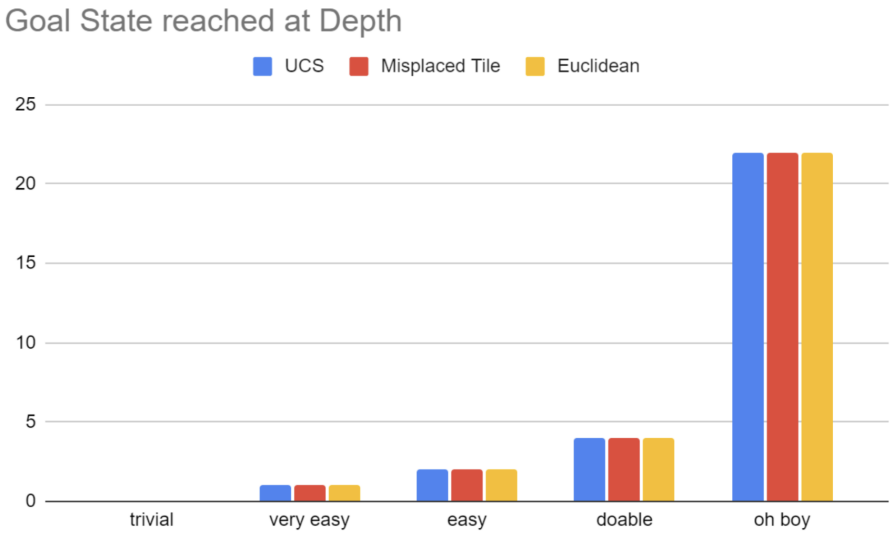
	most nodes in frontier at a single time				
	trivial	very easy	easy	doable	oh boy
UCS	1	5	6	18	24973
Misplaced Tile	1	3	3	4	3359
Euclidean	1	3	3	4	1033

Uniform Cost Search has the most nodes in the frontier at a single time because it does not have the heuristic information to find the goal. This search will find the most optimal path but it will explore more of the search space leading to a larger frontier. A* Search with Misplaced Tile and Euclidean Distance do not need to explore has many nodes so it has less nodes in the frontier.



	nodes expanded				
	trivial	very easy	easy	doable	oh boy
UCS	0	3	5	18	86824
Misplaced Tile	0	1	2	4	6581
Euclidean	0	1	2	4	1033

Our Uniform Cost Search expands the most nodes because it expands based on lowest path cost. This means that USC will continue to expand nodes with lower costs until moving to higher costs, which ensures optimality but can potentially lead to excess node expansion. A* Search with Misplaced Title Heuristic expands more nodes than A* Search with Euclidean Distance Heuristic because the misplaced tile does not always accurately estimate the distance of the goal. This leads to the function exploring sub-optimal paths as well. The Euclidean distance heuristic is more accurate with its estimation because it gives more information about the distances the tiles need to move to reach the goal state.



	goal reached at depth				
	trivial	very easy	easy	doable	oh boy
UCS	0	1	2	4	22
Misplaced Tile	0	1	2	4	22
Euclidean	0	1	2	4	22

Uniform Cost Search, A* Search with Misplaced Tile Heuristic, and A* Search with Euclidean Distance Heuristic all reach the goal state at the same depth: 0 for 'trivial', 1 for 'very easy', 2 for 'easy', 4 for 'doable', and 22 for 'oh boy'. This is due to the fact that A* search is both complete and optimal, with an admissible heuristic function, further implying that the problem space has uniform edge costs, and both choices for the heuristics were admissible.

6 Overall Findings

Overall, we have found that all three methods work well and efficiently to solve the easier test cases, with goal states of lower depths. Once they get more complicated, however, we found that the Euclidean distance heuristic for our A* function was the most efficient in finding the goal state.

7 Future Improvements

Though our program produced correct outputs, and we were able to implement all three search algorithms, we think that with more time we could've produced cleaner and more optimized code. Examples of this would be being able to utilize polymorphism to select a heuristic onto a single search function, instead of having multiple, dividing up tasks more to align itself better to single responsibility coding principles, and ensuring that the entirety of our program is open to extension easier.

Another improvement we think could be interesting to implement, now that we know Euclidean heuristic is the most efficient at more complex puzzles, is producing a way for the program to dynamically choose the best heuristic based on the estimated complexity of the puzzle received.

8 Contributions

Anisha: Implemented core menus, data structures, operators, and generic search algorithm (UCS)
Adelyn: Implemented optimal path tracing, missing tile heuristic, general debugging + test cases
Parnika: Implemented Euclidean distance, visualizations and analysis in report
Pajaka: Implemented Euclidean distance, visualizations and analysis in report

9 Resources

1) Discussions + guidance from TAs
2) https://numpy.org/doc/stable/user/basics.html – basics in NumPy (for syntax/other technicalities)
3) https://www.youtube.com/watch?v=dvWk0vgHijs – basics in A* searching for the puzzle solver
4) https://deniz.co/8-puzzle-solver/ – help for creating test cases and checking efficiency