Project #1 Report

Van Truong, SID: 861279550

In completing this assignment I consulted:

- The Blind Search and Heuristic Search Lecture slides
- The TA's hints during discussion sessions
- Python Documentations and this is the URL to the Table of Contents of 3.94: https://docs.python.org/3/tutorial/index.html

Challenges

There were several challenges that I experienced during this project.

First challenge was designing the classes for this project. Inorder to implement the project in an object-oriented manner, the Problem class and Node class are necessary. However, it was hard to determine what internal variables each class should have. The Problem class should represent the whole problem, and the Node class should represent each state. The most difficult part was deciding which class the operator methods should belong to. An operator is applied to each state or node, but they are also needed for the initial state/node, which belongs to the Problem class. The design process took a few days but ultimately, the operator methods made its way to the Problem class because a Node object can contain a Problem object, giving access to the operator methods while keeping the initial state in the Problem class.

Continuing from the design, another issue arises - how to keep track of the parent node? Since Python is used for the project, pointers cannot be used. In addition, I find the concept of keeping track of the parents is harder to grasp than the concept of keeping track of the children. Thus, I decided to create a list to keep track of children but ended up not using it anyways because there was not enough time to implement the trace.

Lastly, after the implementation of the create_child function to make a copy of the "current state/node", the program prints out the wrong state. After a few days of debugging, an idea comes to mind. Could there be something wrong during the assignment process when creating a new state/node? Indeed. When creating a new child state/node, instead of simply assigning the parent state to a new variable, a deep copy needs to be created. Simply assigning the parent state to a new variable does not create a separate set of values. Both the parent and child variables "point" to the same array/matrix. Thus, modifying the parent changes the child, and modifying the child

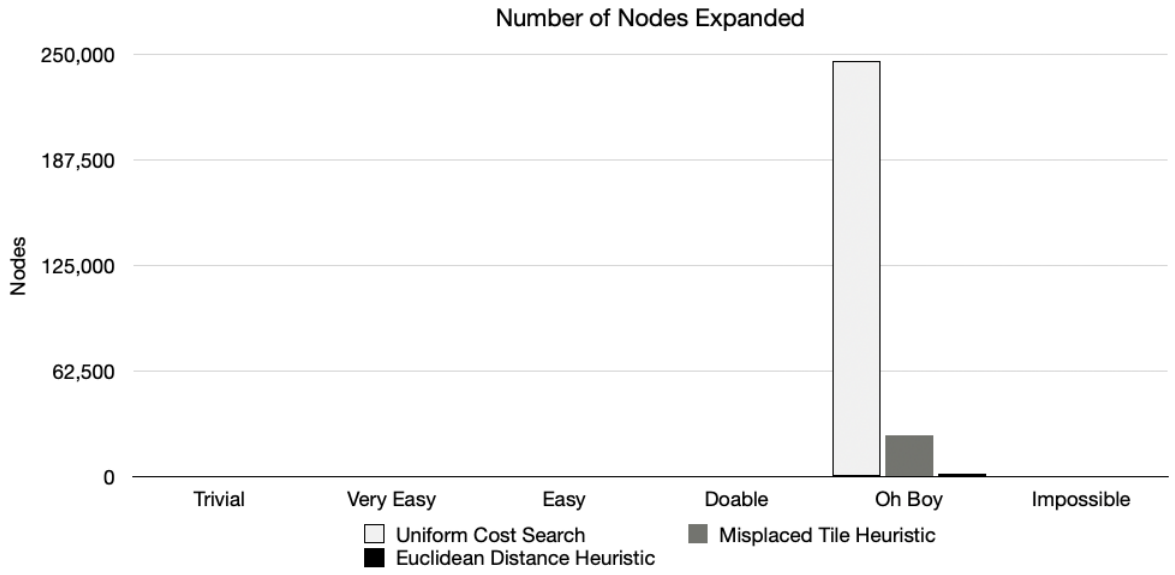
changes the parent. To address that, the library copy needs to be imported, and a new child needs to be created using the deepcopy function.

Optimization

Since Python is used for the project, I did not consider what special data structures I could use to make searching for and comparing states faster. Actually, one of the tough decisions I had to make for this project was which language to use, C++ or Python. I was told that C++ is faster, but I thought Python has easier syntax for comparing arrays, so I went with Python, which now I regret because I forgot that we can compare arrays in C++ by doing (*a1 == *a2).

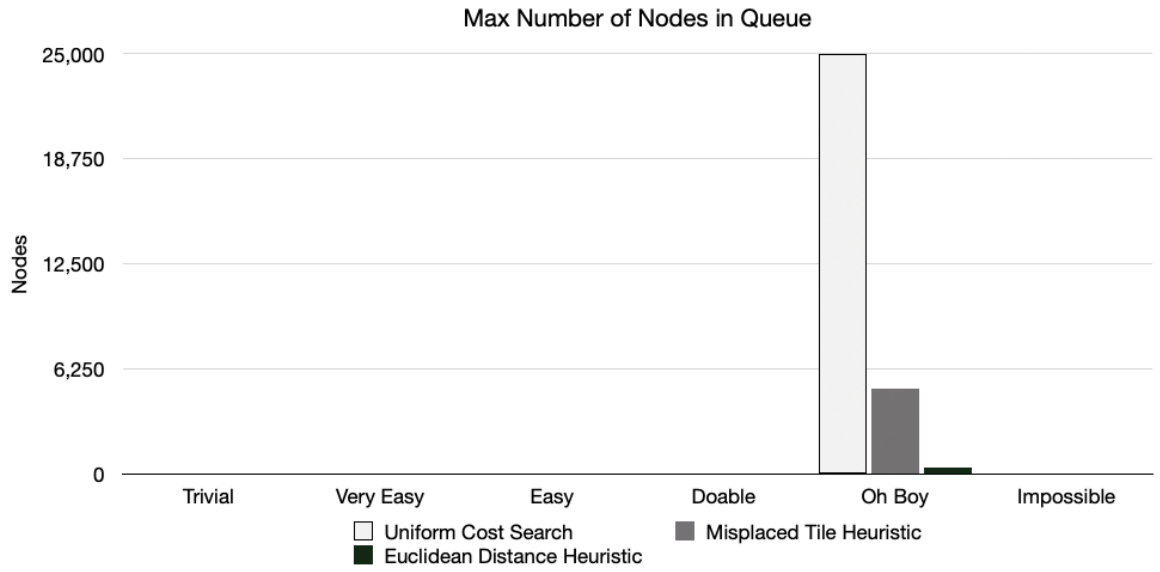
Sample Puzzles Comparison

Below are two charts that show the number of nodes expanded and the maximum number of nodes in the queue at any given time:



Number of Nodes Expanded

	Uniform Cost Search	Misplaced Tile Heuristic	Euclidean Distance Heuristic
Trivial	0	0	0
Very Easy	9	3	3
Easy	8	5	5
Doable	74	10	10
Oh Boy	246,170	24,642	1,814
Impossible			



Max Number of Nodes in Queue

	Uniform Cost Search	Misplaced Tile Heuristic	Euclidean Distance Heuristic
Trivial	1	1	1
Very Easy	5	3	3
Easy	4	3	3
Doable	18	4	4
Oh Boy	24,983	5,051	392
Impossible			

The number of nodes expanded for the "Oh Boy" test case seems abnormally large. And indeed, compared to the program executed by the TA during discussion, my algorithm resulted in a much larger number of expanded nodes.

The matrix used during discussion: [1,2,3]
[4,5,6]
[7,0,8]

	TA	Van
Expanded Nodes	3	9
Max Queue Size	6	5

Running the same algorithm on the same matrix with my program results in the differences highlighted in the table above.

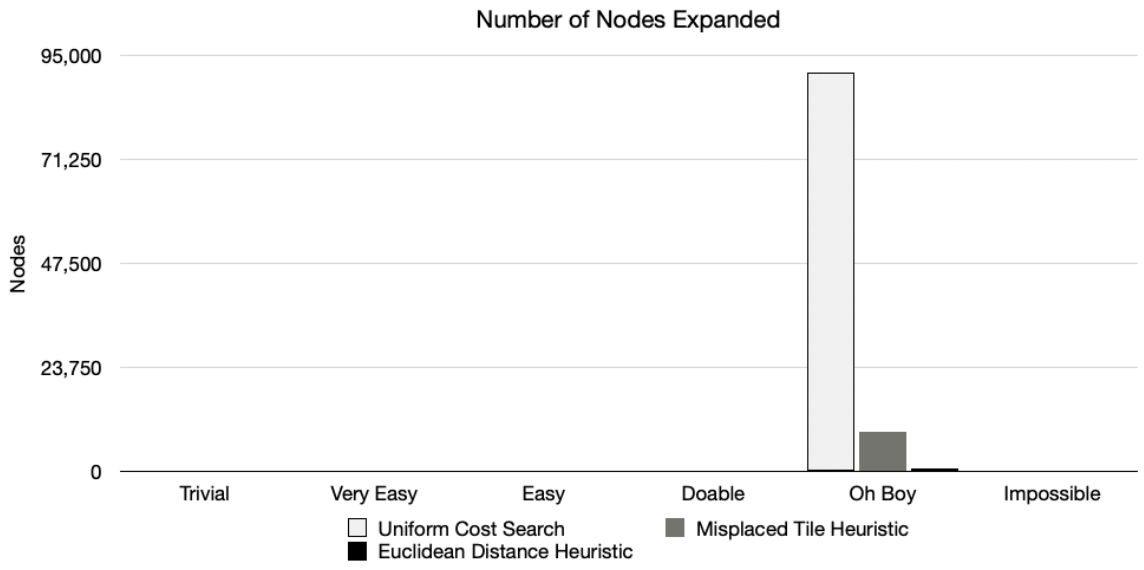
After revisiting the code, the issue found with this abnormality is that my program is keeping track of the number of children the expanded node generated rather than the expanded node itself, like so:

expanded_children += len(new_children)

Thus, I changed the code to:

expanded_children += 1

This line of code is now adding one to the counter every time a node is expanded, and the new result is shown in the updated bar graph below.



Number of Nodes Expanded

	Uniform Cost Search	Misplaced Tile Heuristic	Euclidean Distance Heuristic
Trivial	0	0	0
Very Easy	3	1	1
Easy	3	2	2
Doable	29	4	4
Oh Boy	91,120	9,103	683
Impossible			

As we can see, the number of nodes expanded reduced significantly after changing the code.

Note: Running the Uniform Cost Search on the "Oh Boy" test case took 3301 seconds (55 minutes) to complete.

A similar situation occurred for the "Impossible" test case. Supposedly, the program is to return no solution, yet the program never returned no solution even after running for more than an hour. I am not sure if I was supposed to run it longer, but the machine could not handle the stress, so I stopped the execution. Thus, not being able to get the results for the "Impossible" test case. An observation that I made for the "Impossible" test case is that the g(n) value fluctuates instead of increasing constantly.

Conclusion

From the graphs and data tables, we can conclude that the algorithm used does not matter for problems that are easy. Although Uniform Cost Search has a significantly larger size, the size for both expanded nodes and explored nodes are relatively small for the easy problems. The difference is apparent as we move on to harder problems, shown with the "Oh Boy" test case. The size of both expanded nodes and explored nodes explodes. This is where heuristics can help greatly reduce the sizes. Even so, there is a difference between a good heuristic and a weak heuristic, as shown with the Misplaced Tile heuristic and Euclidean DIstance Heuristic. And as mentioned during lecture, heuristics that underestimate (admissible) are more desirable than ones that overestimate because we want to be as close to the truth as possible. Euclidean Distance measures the hypotenuse of the path to the next state/node, which of course, is always closer than taking the L-shape path.