CS170 ASSIGNMENT #1 REPORT

SAIKRISHNA REDDY

CHALLENGES

There were a few challenges I faced during this project. The main ones were adapting what we learned in class into a running program code. Understanding Uniform Cost Search, A* with the Misplaced Tile heuristic, and A* with the Euclidean Distance heuristic took some time to implement as code. Once I was able to understand these concepts thoroughly, the rest was transforming it into code and making sure it works. A lot of debugging took place as well for issues I had such as an ongoing infinite loop, improper storage of data, simple coding errors, and using wrong algorithms, just to name a few. Once all of these were dealt with, the program is able to run with no issues. I was able to properly organize the code and make it readable and clear for the users/audience.

HOW TO RUN

To run the program please note that the driver code is located in **8puzzle.cpp** and to execute:

g++ 8puzzle.cpp

./a.out

IMPLEMENTATION

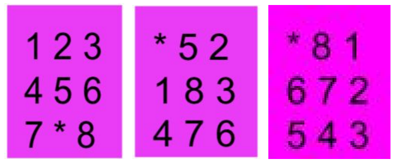
This 8 Puzzle program was solved with the **graph search algorithm** as discussed in the lectures. We know this method interfaces with a graph search so, an **explored set** is used to maintain and contain the states as we have seen prior.

The main data structure I have used is C++'s **PriorityQueue.** Below is the list of functions and variables implemented.

```
public:
    Problem();
    int parent[puzzleSize];
    vector<Problem> children;
    vector<Problem> predessor;
    int depth;
    int heuristic;
    void moveUp(int x, Problem root, int choice);
    void moveDown(int x, Problem root, int choice);
    void moveLeft(int x, Problem root, int choice);
    void moveRight(int x, Problem root, int choice);
    int findZero();
    void vctrOutput();
    void vctrOutput(int origpuzzle[puzzleSize]);
    bool goalCheck();
    void duplicateBoard(int destBoard[puzzleSize], int origBoard[puzzleSize]);
    void expansion(Problem node, int x);
    bool ifListContainsNode(Problem node);
    bool exact_puzzle(Problem node);
    void uni_search(Problem node, int x);
    void pathtrace(Problem node, int);
    bool originalMatrixCheck(Problem node);
    int checkMisplacedTile(Problem node);
    void misplaced_tile_search(Problem node, int x);
    void Euclidean();
    void euclid_dist_search(Problem node, int x);
    void new_node_size_check();
    void priority_queue_size_check();
```

COMPARISONS/CONTRASTING

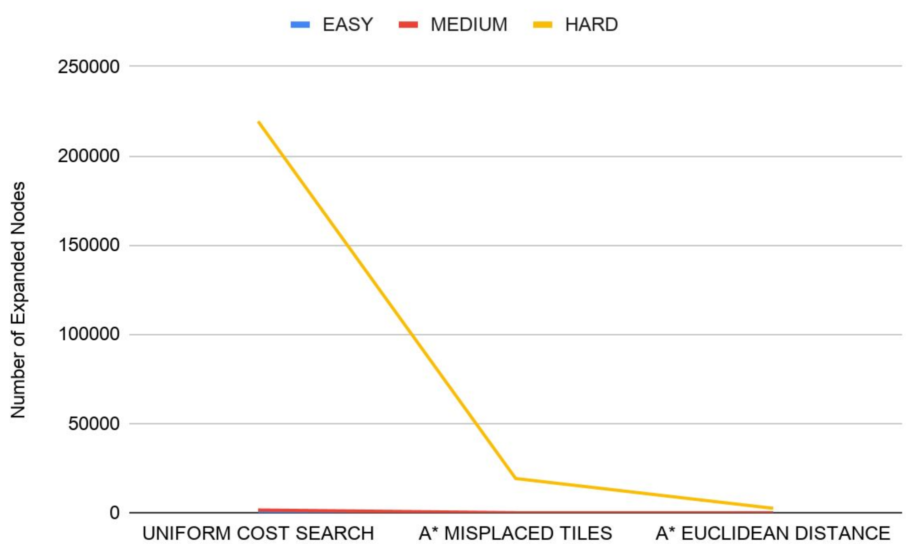
Three various levels were analyzed in regard to each of the given search algorithms. Here are the results:



EASY MEDIUM HARD

NUMBER OF EXPANDED NODES

	EASY	MEDIUM	LARGE
UNIFORM COST SEARCH	1	1531	219320
A* MISPLACED TILES	1	8	19169
A* EUCLIDEAN DISTANCE	1	8	2463

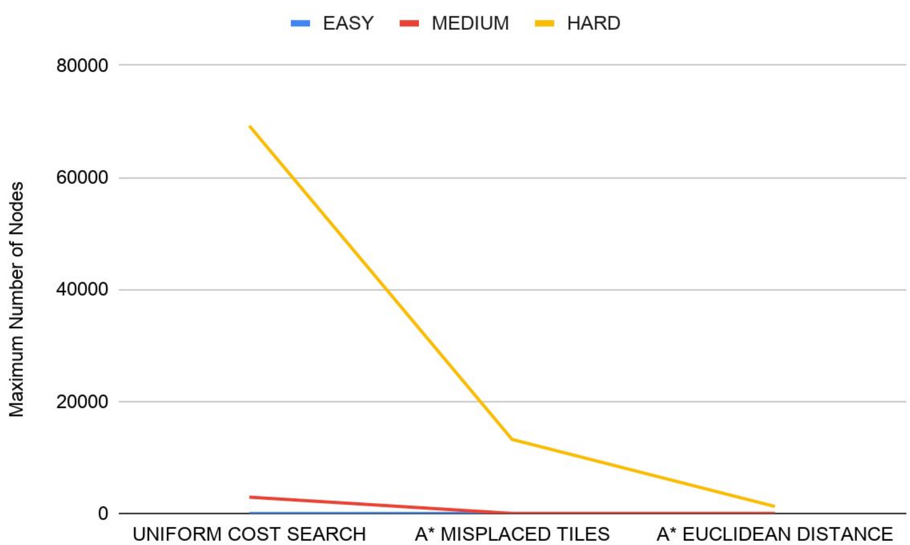


In these scenarios above, it is evident that Uniform Cost Search executes poorly in regards to the number of nodes it needs to expand. It is no surprise since g(n) is the complete path cost in order to reach a certain state. So, as the tree increases and expands the nodes that follow, it would deque from the frontier state as a sibling. The depth increase in trees will lead to new nodes growing at a massive rate.

We can see that the A* Search is not as impacted since a heuristic function is implemented with regards to g(n). So, if we want to use the Euclidean Distance, we know that g(n), due to
$f(n) = g(n) + h(n)$, the value of h(n) assists in the findings of the closest state to the goal state.

MAXIMUM NUMBER OF NODES IN QUEUE

	EASY	MEDIUM	LARGE
UNIFORM COST SEARCH	1	2915	69212
A* MISPLACED TILES	1	14	13224
A* EUCLIDEAN DISTANCE	1	14	1251



We are able to see that the Euclidean Distance works well in this scenario by having a smaller value of max nodes in the frontier. If we were to compare this to the Misplaced Tiles, the Euclidean Distance is able to calculate the distance of tiles from their goal state. If we were to dequeue from the frontier then it will return a state that is closer in proximity in regards to the goal state. Also, Uniform Cost Search is once again not that effective or efficient, as we see, due to dequeuing small g(n) nodes, and so on lead to more nodes (i.e siblings). However, for Misplaced Tiles even if there is a small number it does not mean the tiles are closer, from what we have learned in the lecture.

RESOURCES

Lectures regarding Heuristic Searching,

C++ PriorityQueue Documentation
http://www.cplusplus.com/reference/queue/priority_queue/