**Solving8-puzzle problemby Searching**

**1.** **Team Members and Time Report:**

|  |  |  |  |
| --- | --- | --- | --- |
| **First Name** | **Last Name** | **Total Time** | **Contributions** |
| **Alex** | **Werner** | **11 hrs** | Wrote GUI, implemented breadth first search, and documentation |
| **Jordan** | **Renner** | **10.5 hrs** | A\* implementation and documentation |
| **Nate** | **Braukhoff** | **10 hrs** | Implemented Greedy Best First Search. |

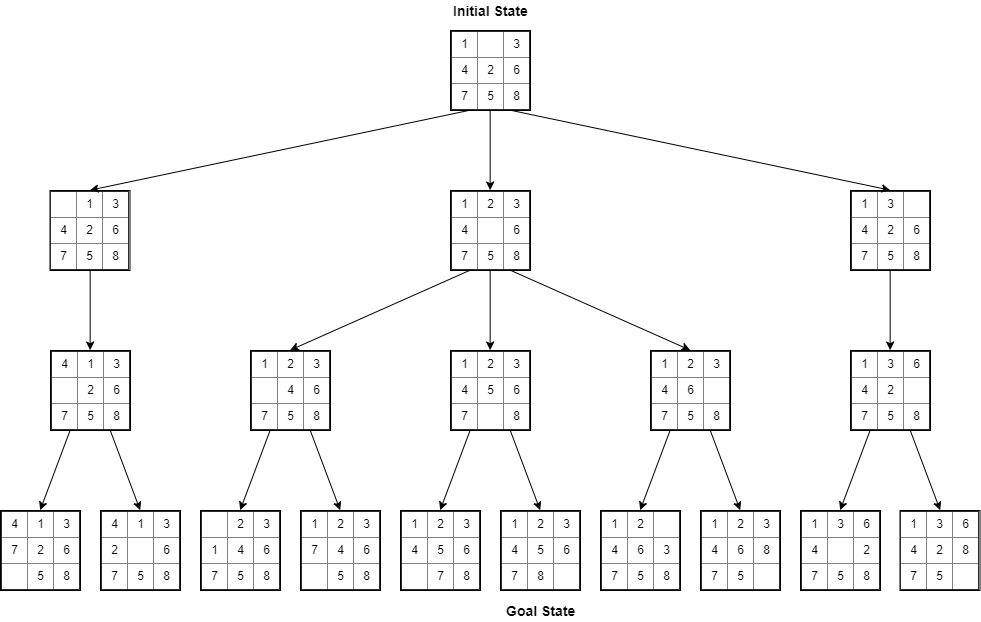
**2.** **Problem Description:**

The 8-puzzle problem consists of a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a single blank square. The objective is to maneuver the blocks so that they are in order. You are allowed to slide blocks up, down, left, and right into the blank space. Pieces are not able to be picked up or taken out of the puzzle in any way.

**3.** **Problem Modeling:**

1. Initial states
   1. Any solvable permutation of the 8 tiles and the empty space in the grid. If the parity of the initial state is the same as that of the goal state, then that state is solvable.
2. Actions and transition models (Operators)
   1. Possible operators to a given state include moving a tile left, right, up, or down into the location with the empty space
3. Goal state test
   1. If the location of each of the tiles in a given state is the same as the location of those tiles in the goal state, then that state is a goal state.
4. Path cost function
   1. The path cost of a state is equivalent to the number of moves required to reach a given state from the initial state.

Graph Example:



**4.** **Implementation**

Python was the language we chose to implement our solution. We chose this language because we all wanted to learn more about it. We used various data structures to implement the state space. All grids are stored in 2d lists. For the informed search algorithms, node objects are created and stored in lists and dictionaries.

**5.** **Uninformed Search Algorithm:** Breadth first search

**5.1.** **Algorithm Description**

Breadth first search traverses the search tree layer-by-layer and checks each node to see if it is the goal node. In order to actually implement this, nodes are put into a queue, and nodes are taken out of the queue, checked against the goal state, then that node’s children are added to the end of the queue. In an attempt to reduce the size of the tree, and therefore the number of nodes needing to be traversed, another list is used to track all previously visited nodes, and new nodes are only added to the queue if they have not already been visited. This also has the effect of removing the possibility of cycles.

**5.2.** **Algorithm (pseudo code)**

Initialize *queue* with initial state

Initialize empty *visited* set

While *queue* is not empty:

Dequeue a node *x*

Add *x* to the visited set

If *x* is the goal state:

Return

For each possible child node *c* of *x*:

If *c* is not in *visited*:

Append *c* to *queue*

**5.3.** **Algorithm Properties**

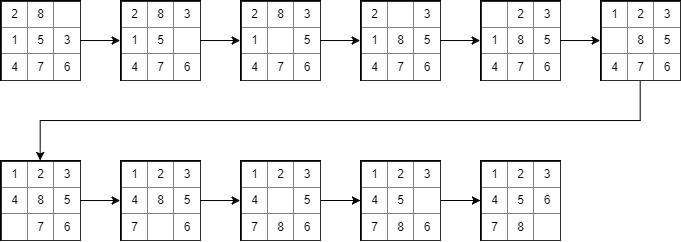
**Complexity:** Time and space complexity are both O(b^d), where ‘b’ is the branching factor of the tree, and ‘d’ is the depth of the shallowest goal state. For this problem, ‘b’ is around 2, and ‘d’ tends to be around 20.

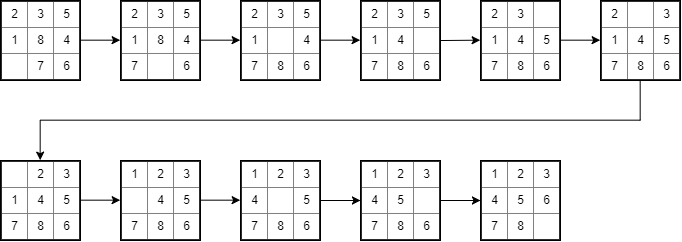
**Completeness:** Yes

**Admissibility:** Yes

**Irrevocability:** Yes, this implementation of BFS is irrevocable since the algorithm maintains a list of visited states, and does not go to any states that have already been visited.

**5.4.** **Results**

**

**

**6.** **Heuristic Search Algorithm 1:** A\* search

**6.1.** **Heuristic Description**

A\* is a widely used pathfinding algorithm typically used in map applications or gaming. It’s popularity derives from its performance and accuracy, but cannot compete with other traversal algorithms that can pre-process the graphs. It’s heuristic function, *h(n)*, is problem-specific, but this projects A\* heuristic is based on Manhattan Distances. In each iteration of the algorithm, each node stores the sum of Manhattan Distances for each misplaced tile. The evaluation function, *f(n)*, adds the previously summed distances to the cost of the path from the start node to n, *g(n)*. The algorithm evaluates the nodes with the lowest f-scores until the goal node is found.

**6.2.** **Algorithm (pseudo code)**

Initialize open set with initial node

Initialize closed set

While open set is not empty

bestNode = pop top node from open set

If bestNode is the goal node

Reconstruct the path from initial node to goal node

Else

Generate children of bestNode

Foreach child node

If the child node is not in the open set or closed set

Append child node to open set

Else if the child node is in the open set

If this child’s fscore is lower than child’s fscore in open set

Update the child node in open set

Append bestNode to the closed set

Sort the open set by lowest to highest fscore

**6.3.** **Algorithm Properties**

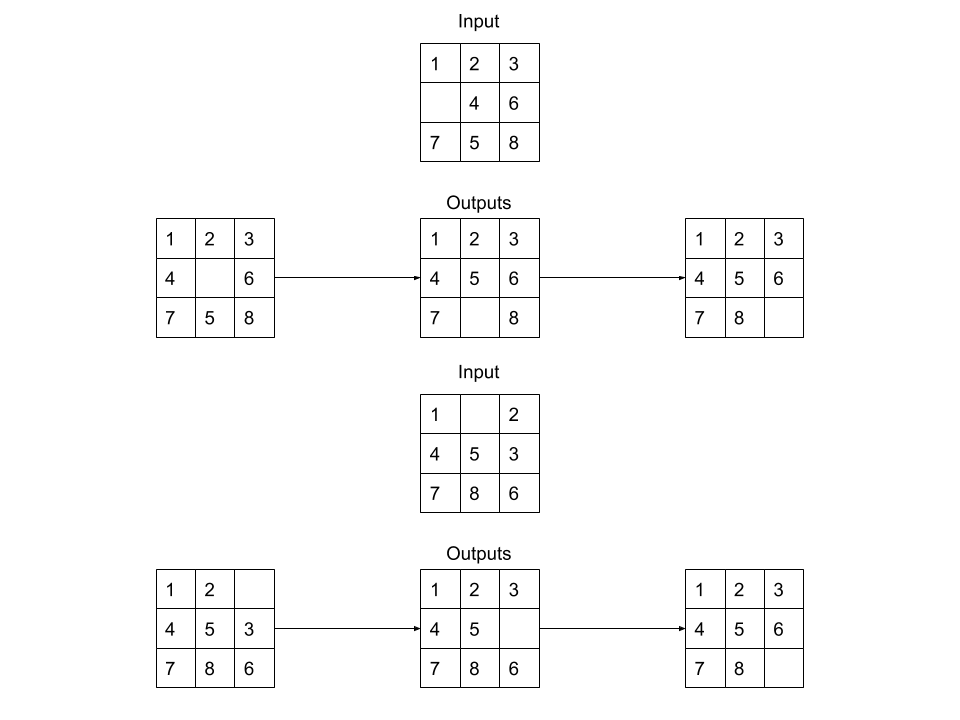
**Complexity:** O(log h(n))

**Completeness:** A\* is complete

**Admissibility:** A\* is admissible

**Irrevocability:** A\* is not irrevocable

**6.4.** **Results**



**7.** **Heuristic Search Algorithm 2: Greedy Search**

**7.1.** **Heuristic Description**

In Greedy Best First Search, will choose the child that has the better heuristic values without considering future values. This means that the solution won’t be the most optimal one. For the 8-puzzle problem the heuristic value that I use was the number of tiles that are in the right spot. So, 9 is the heuristic value that the Greedy will be working towards. Each node in the tree will have a value that will represent the number of tiles that are in the correct spot. Greedy will move to the node that has the higher heuristic value.

**7.2.** **Algorithm (pseudo code)**

Create an open list and a close list

Add the current state into the open list

While their is not open nodes

Sort list greatest to smallest h(x)

Next node = open.pop

If current node == goal

Add current node into close list

break

Get all possible moves

For every child

Make child of the current node

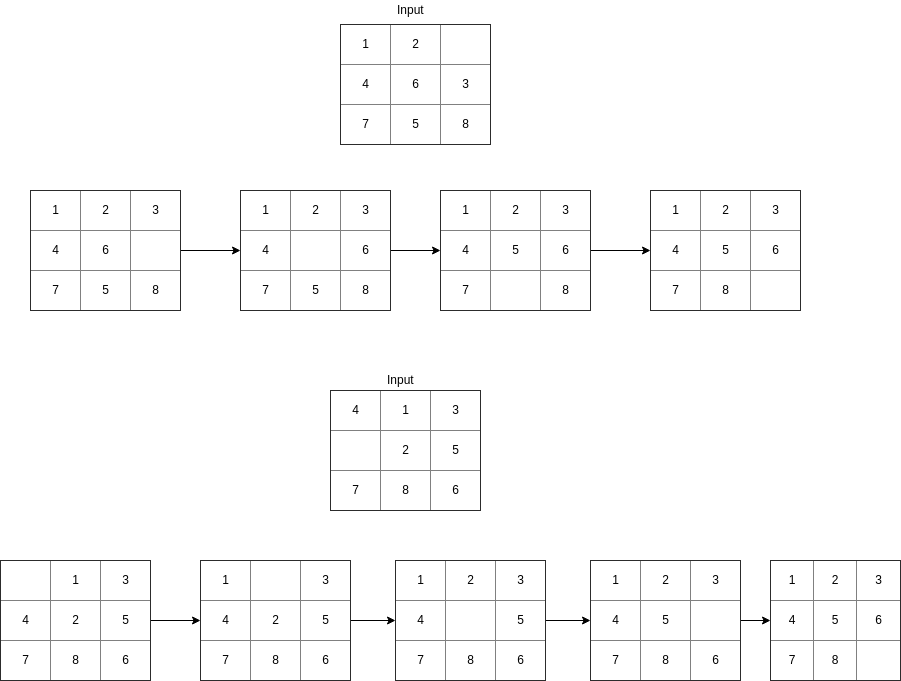
Add child to open list

Add the next node to close list

**7.3.** **Algorithm Properties**

* **Complete:** No, can get stuck in a loops
* **Complexity:** O(N \* logN), N = number of nodes
* **Time Complexity:** O(log N)
* **Admissibility:** Greedy Best first search is not admissible
* **Irrevocability:** Greedy Best first search is not irrevocable.

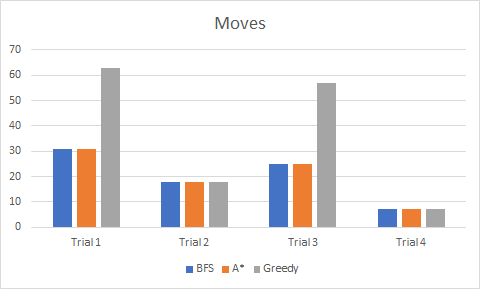
**7.4.** **Results**

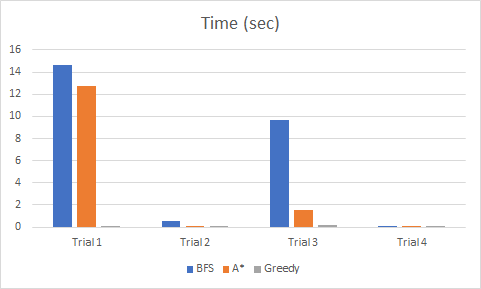
****

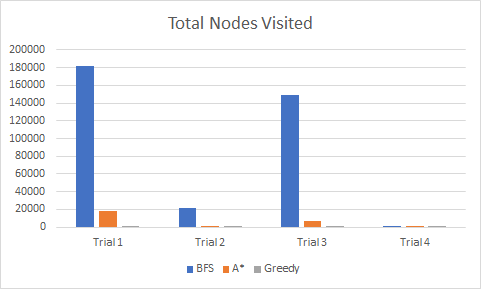
**8.** **Empirical Analysis**

**Data Comparison Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Trial | Starting Position | Algorithm | Number of Moves | Time | Nodes Visited |
| Trial 1 | 867254301 | BFS | 31 | 14.67 | 181439 |
| A\* | 31 | 12.71 | 18598 |
| Greedy | 63 | 0.0209 | 302 |
| Trial 2 | 378102645 | BFS | 18 | 0.55 | 21737 |
| A\* | 18 | 0.01 | 71 |
| Greedy | 18 | 0.00399 | 96 |
| Trial 3 | 278314506 | BFS | 25 | 9.63 | 149013 |
| A\* | 25 | 1.58 | 6658 |
| Greedy | 57 | 0.142 | 962 |
| Trial 4 | 103726548 | BFS | 7 | 0.002991 | 141 |
| A\* | 7 | 0.000998 | 19 |
| Greedy | 7 | 0.000997 | 11 |

****

****

****

The table and graphs above exhibit the differences between the three algorithms we used, Breadth First Search, A\* Search, and Greedy Best First Search. This data is a great representation of how informed search algorithms tend to perform better than uninformed search algorithms. It’s also worth noting that Greedy BFS consistently visited less states in a shorter amount of time than the other two algorithms, though for more difficult initial states, Greedy BFS also required significantly more moves than the other two algorithms.

**9.** **Conclusion**

We learned that Python is a versatile language that is easy to comprehend and adapt to. We have gained a better understanding of the differences between informed and uninformed search algorithms, as well as the effect of using more dominant heuristic functions in the same search algorithm. In the future, it would be interesting to take the 8 Puzzle and generalize the problem into a larger grid and see how much of a difference there is in algorithm performance for larger sized grids.

**10.** **Repository**

The repository for this project can be found on github at: <https://github.com/Nate96/8PiecePuzzle>

**11. Screenshots**

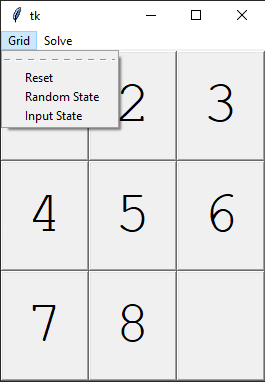


Figure 1 - Grid Submenu

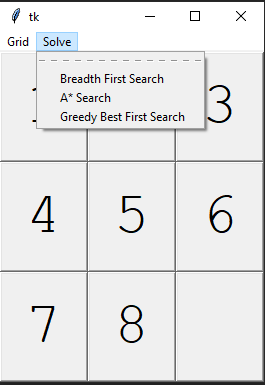


Figure 2 - Solve Submenu

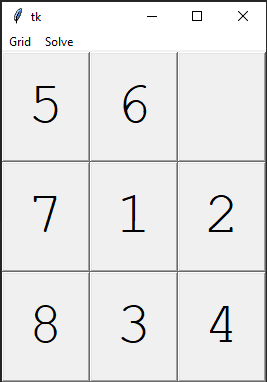


Figure 3 - Puzzle in solvable state

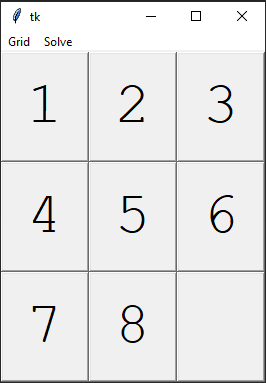


Figure 4 - Puzzle in goal state