Graph Searching Algorithms: N-Puzzle Problem

Deo Fetalvero

April 27, 2019

1 Introduction

The N-Puzzle Problem is a sliding puzzle that consists of N moving square tiles in arranged to make make up a larger puzzle that has equal rows and columns of those tiles. In the puzzle, a tile that along with all other tiles complete all the spaces in the rows and columns is deliberately removed so that all other tiles can be rearranged or disarranged by moving adjacent tiles to the space of the supposed tile. Each tile is labeled by its original position in the grid (top to bottom, left to right). To solve the puzzle, the tiles should be moved one by one such that the original position of each tile correspond with it current position in the grid (top to bottom, left to right).[1]

The puzzle was invented by a postmaster in New York as early as 1874. Since then, the puzzle has become popular by the spread of copies thus, making its way to the manufacturers and sellers. Since its invention, the puzzle started with labels using numbers from 1 to N where 1 is the position at the top left corner and N is at the bottom right (top to bottom, left to right). With the position labels, it was relatively easy to find where a out of place tile should go. Further improvement of the puzzle was done by utilizing parts of images such that when arranged correctly would result to a complete cohesive image. This way increases the difficulty of the puzzle depending on how recognizable each image part of every tile that could connect to other image parts to form the bigger image.[6]

2 Methodology

The current discovered and proposed solutions to this puzzle problem are graph searching algorithm. Each node in the graph is an arrangement of the tiles labeled by its original position. The node would then have children nodes of another arrangement with the connecting edge as the move made to achieve another arrangement. Each move is done by sliding adjacent tiles to the empty tile space or by swapping the empty tile's position to one of the adjacent tiles in the arrangement. Of course, the tiles can't go outside the borders of the puzzle reducing the adjacent tiles that could move (or slide) when the empty tile space is beside the border of the puzzle.

Every move is done by sliding a square tile to the empty square tile space. Since the puzzle is composed of square tiles, the maximum moves that can be done by sliding to the empty tile space on a specific tile arrangement is 4. These moves that are possible with the puzzle can be labeled as the left move, right move, top move and down move. Each movement is based on which adjacent tile will move to the empty tile space in the arrangement. For example, the left move is done by moving the left adjacent-to-the-blank-space tile to the blank space.

To find the solution from a shuffled arrangement, where not all tiles are in their original or supposed position, to the original arrangement, where all tiles are in their supposed position, arrangement nodes are traversed from the shuffled arrangement node to search for the original arrangement node with each move possible from the previous arrangement as the connecting edge of the previous arrangement (parent) node to the next arrangement (child) node. The traversing or search only stops when the original arrangement node is found

connected using a specific order of edges or moves done from the said shuffled arrangement.

When using a graph searching algorithm, an increased search space composed of all nodes also increases the computational difficulty of finding nodes in the space. This is also happens to the graph searching algorithm when the number of tiles increases by adding rows and/or columns of square tiles to the puzzle. Since adding more tiles add more possible disarrangements that can be done by moving the tiles anywhere in the grid. It can be put simply that since there are more tiles, there would be more permutations from an increased tile count and more permutations would result to a larger search space.

2.1 Graph Searching Algorithms: Breadth First Search

One of the very basic and easily implementable solution of the N-Puzzle Problem is the Breadth First Search (BFS). A BFS is done by checking every node by each level of node depth. Since the search starts from a specific node, node depth describes the number of edges that are required to reach a another node from that specific node. This only means that BFS checks the nodes with the least amount of edges traversed from the specific starting node (or the "nearest" nodes) first before checking other nodes.[2]

This can computationally done by queueing nodes by their level of node depth. A basic implementation of BFS is:

a. Label the tree node or starting node as the current node.

b. Create a queue that would contain nodes to be checked.

c. Check if the current node is the node being searched for.

- If it is, skip to step 7.
- If it isn't, continue to step 4.

d. Get all the possible children nodes or nodes that can be traversed by an edge from the current node.

e. Add all the nodes from the previous step to the end of the queue.

f. Remove the node from the start of the queue, set this node as the current node and go back to step 3.

g. Collect the edges that in-order traverses the starting node to the node being searched for.

Step 7 in the implementation can be easily done by alternatively storing the edges that have been traversed to reach the node (in step 4) inside that same node and then accessing that information in step 7 also.

It is also important to note that some graphs have node cycles such that blindly traversing nodes can make the traversal to loop back to a previously checked node (which can also cause infinite loops). One example of such graph is from the N-Puzzle Problem since each arrangement can be redone using a few number of moves and one of these few number of moves is by simply doing a set moves on an arrangement and then reversing those moves such that the left move reverses right move v.v. and up move reverses down move v.v.

Preventing looping checks described previously is also important to not waste compute power on traversing from a previously searched node for its children and "grand"-children which are also assumed to be already checked. The best way to mitigate this is by allowing the nodes to be serialized uniquely by how each node can be distinguished from each other (note: this can't be

traversed edge information) and each checked serial should be included in a hash set (fast set similar to a hash map which stores a unique item only once). For the N-Puzzle Problem this should be the original positions of each tile in order of how they are currently arranged spaced out by a separator. This serialization method also can be used to identify the node being searched for in step 3.[3]

For solving the N-Puzzle Problem, the implementation of BFS should allow: nodes to be serialized, node's children be identified and instantiated, edge traversal information be recorded in nodes and a hash set to be utilized for containing information of nodes already checked. These requirements are already described in the previous paragraphs.

2.2 Graph Searching Algorithms: Branch and Bound using Manhattan Distances

Branch and Bound (BnB) is a step up from the BFS Graph Searching Algorithm. This is because it identifies the best node to branch out at a certain time. Every node is ranked for their potential of reaching the solution in just a few traversals. Unlike BFS, BnB uses a cost function that allows it to evaluate every node available for traversal and find the current best node to branch out.[1]

Implementing BnB to find a solution for N-Puzzle will require a cost function. A cost function needs to be clear and should evaluate a node without relying on contextual data such as surrounding nodes to be fast. It should determine the best node using information that the node only provides. Such information for solving N-Puzzle would use move set length and tile arrangement.

The Manhattan Distance (Taxicab Geometry) is the sum of the absolute differences between the cartesian coordinates of two vectors. Unlike Euclidean distance being the shortest unique path between two vectors, Manhattan distance's shortest path is not unique and can be measured diagonally by alternating measurements of 1 unit between each coordinate from one vector to reach the other vector. [5]

For the implementation of the N-Puzzle solution using BnB, the sum of the Manhattan Distances of each tile from its original position in the arrangement is used. The cost function for such implementation[1][4] is:

$$\begin{aligned}\text{Cost of Puzzle Arrangement Node (state)} s &= f(s) \\ &= \text{NodeMoveListLength}(s) \\ &+ \text{SumOfTilesManhattanDistances}(s)\end{aligned}$$

The length of the current moves (edge traversals) allows the implementation to minimize the move set taking the shortest possible while the sum of manhattan distances allows it to predict which nodes are more likely to reach the original arrangement node first.

This implementation is an improvement to the BFS implementation already described. The following is the implementation of BnB for solving the N-Puzzle:

a. Label the tree node or starting node as the current node.

b. Create a queue that would contain nodes to be checked.

c. Check if the current node is the node being searched for.

   - If it is, skip to step 7.
   - If it isn't, continue to step 4.

d. Get all the possible children nodes or nodes that can be traversed by an edge from the current node.

e. Add unchecked nodes (using hash set) from the previous step to the end of the queue.

f. Evaluate each node in queue using cost function and take last least cost node, set this node as the current node and go back to step 3.

g. Collect the edges that in-order traverses the starting node to the node being searched for.

In the implementation, the last least cost node is used because it means that the nodes with more established moves are at the end of the queue and their sum of manhattan distances is minimized despite there are nodes that are of the same cost at the start of the queue. This only means that the nodes that are at the start of the queue have a greater sum of manhattan distance as part of the cost than the length of established moves while the nodes at the end of the queue tend to minimize the sum of manhattan distance while also having more moves that are concrete or already established.

3 Parameters of the experiment

Table 1 shows the 30 test cases of the experiment. Each test case will run 10 trials each using the BFS and BnB implementation. Each test case represent a 3 by 3 matrix with numbers from 0-8 indicating their original location in the grid. 0 represents an empty tile space which should originally be at (2,2) [zero-indexed]. The original or "solved" arrangement is as follows:

123456780

The objective of each algorithm is to rearrange each test case to the original arrangement using the basic moves. Each algorithm will record its move list, node traversals or checks and calculate the time duration the algorithm was running. Each test case will run 10 trials each resulting to 300 runs per algorithm.

Test Case	Shuffled Arrangement
1	2 0 8 6 3 5 1 4 7
2	7 5 8 4 6 0 3 2 1
3	2 6 7 5 3 4 1 0 8
4	4 1 2 0 8 3 7 6 5
5	5 0 1 2 4 8 7 3 6
6	1 7 8 0 4 3 6 5 2
7	6 2 3 1 4 5 8 0 7
8	2 8 1 7 5 0 4 3 6
9	7 1 2 5 8 6 4 0 3
10	7 5 4 3 1 0 8 6 2
11	4 1 8 3 2 7 6 0 5
12	6 8 5 0 7 2 3 4 1
13	1 0 3 5 8 2 4 7 6
14	6 1 3 0 7 4 8 2 5
15	1 0 7 6 4 5 3 2 8
16	7 2 8 0 5 4 6 3 1
17	5 2 4 7 8 1 6 0 3
18	3 1 5 8 7 4 2 0 6
19	7 2 4 3 5 0 8 6 1
20	8 7 6 5 4 3 2 1 0
21	7 4 6 2 3 1 8 5 0
22	8 6 7 4 0 2 1 5 3
23	1 5 6 4 0 2 7 8 3
24	5 3 7 1 0 2 6 8 4
25	7 5 1 6 8 0 2 3 4
26	2 3 0 1 6 8 5 4 7
27	8 2 6 7 4 5 1 0 3
28	0 8 6 4 5 1 2 3 7
29	0 2 7 5 8 1 6 4 3
30	3 8 4 0 7 6 1 5 2

Table 1: Test Cases

4 Results and Discussion

The test cases were run and recorded for the results. Table 2 shows the resulting move set found by each algorithm for each case.

Notice that solutions differ between the algorithms for some test cases. This is because the behavior of choosing last best (least cost) has some effect over the traversal of the algorithm compared to choosing first in queue like in BFS.

Another thing to notice is that some solutions found by BnB is not the most optimal (least moves) solution. This reflects how imperfect cost functions could be. Since complete step-by-step graph search like BFS are careful enough to find for the most optimal solution, most BnB cost functions use heuristics as its basis and even heuristics can't find the most optimal solution in every situation, every time. Sometimes deep and careful searching like BFS is good for solutions with high confidence.

Table 3 shows all the runs of each trial of each test case. As discussed earlier, sometimes BFS can also be good. The only downside to BFS is how long it would need to find that solution. It can be seen in the table that BnB has an upper hand over BFS since there is a large difference between node checks. This shows how fuzzy searching like BnB can be significantly faster than the traditional BFS.

It can be said that this speed difference between the two can be more evident in larger search space like the 15-puzzle problem. Since larger puzzles have more tiles, BFS would struggle in slowly searching every possible instance of arrangement unlike BnB that choose what looks like the most optimal at the

time.

From the experiment, it is evident that while Breadth First Search employs a deep and careful search for solutions of the N-Puzzle problem, Branch and Bound search is still significantly faster and would overall conserve compute power as an algorithm.

Algo(Test)	Solution Move Set
BFS(1)	**DLDRRUULLDDRRULDR**
BnB(1)	**DLDRRUULLDDRRULDR**
BFS(2)	**LLURDRDLLUURRDDLUURDDLULURRDD**
BnB(2)	**DLLUURRDDLUURDDLULURRDDLULDRR**
BFS(3)	**RUULDRDLULDRURULLDRDR**
BnB(3)	**RUULDDRULLDRRUULLDDRR**
BFS(4)	**URRDDLURD**
BnB(4)	**URRDDLURD**
BFS(5)	**LDRDRULDLUURRDLLURDLDRR**
BnB(5)	**DLDRRULDLUURRDLDRULLURRDD**
BFS(6)	**RULDRRDLLUURRDLURDDLURD**
BnB(6)	**RULDRRDLLUURDRULDDRULDR**
BFS(7)	**LUURDDRULLDRRULULDRDR**
BnB(7)	**LUURDDRULLDRRULULDRDR**
BFS(8)	**LDRULLURRDDLULURDLDRURD**
BnB(8)	**DLURDLULURRDDLULURDLDRURD**
BFS(9)	**URDLLUURRDLDLURDR**
BnB(9)	**URDLLUURRDLDLURDR**
BFS(10)	**ULDLURDRDLLURRULDRD**
BnB(10)	**ULDLURDRDLLURRULDRD**
BFS(11)	**RULLURDRULDDLURDRULLDRR**
BnB(11)	**RULLURDRULDDLURDRULLDRR**
BFS(12)	**RRULDDRULDLURRULLDRDRULURDLDR**
BnB(12)	**DRUULDDRURDLLURRULLDRRULLDDRR**
BFS(13)	**LDDRURDLLUURDLDRR**
BnB(13)	**LDDRURDLLUURDLDRR**
BFS(14)	**RDLUURDLDRURDLLURRD**
BnB(14)	**RDLUURDLDRURDLLURRD**
BFS(15)	**RDDLLURRDLLURURDDLULDRUURDD**
BnB(15)	**RDDLLURULDRRDLLUURRDDLUURDD**
BFS(16)	**RRULDDRULDLUURDDLUURRDLDR**
BnB(16)	**RRULDDRULDLUURDDLUURRDLDR**
BFS(17)	**LURURDLLURRDDLULURRDD**
BnB(17)	**LUURRDDLULURRDLLURDRD**
BFS(18)	**ULDRUULDRRULLDRRD**
BnB(18)	**ULDRUULDRRULLDRRD**
BFS(19)	**DLUURDLLURRDLDLUURRDD**
BnB(19)	**DLUURDLLURRDLDLUURRDD**
BFS(20)	**LLUURDLDRURDLLUURDRULDDLUURDDR**
BnB(20)	**ULLDRUURDDLUULDDRUURDDLUULDRRD**
BFS(21)	**LULURDRULDDRULDLUURRDD**
BnB(21)	**LULURDRULDDRULDLUURRDD**
BFS(22)	**LURRDLLDRUULDRRDLLURURDLDR**
BnB(22)	**ULDDRUURDDLUULDRURDDLLURDR**
BFS(23)	**URDDLUURDLDR**
BnB(23)	**URDDLUURDLDR**
BFS(24)	**URDLULDRDRULDLURRDLUURDD**
BnB(24)	**RULDDRULLDRRUULDLURRDLLDRR**
BFS(25)	**LDRULLURRDLLDRULURDLDRR**
BnB(25)	**DLURDLULURRDLLDRULURDLDRR**
BFS(26)	**LLDDRRULLDRR**
BnB(26)	**LLDDRRULLDRR**
BFS(27)	**ULURRDLLDRURDLUURDLULDDRR**
BnB(27)	**ULURDLDRUURDDLUURDLULDDRR**
BFS(28)	**RDRULDDLUURDDRUULDLDRURD**
BnB(28)	**DRURDLDRULURDLLDRULURDDR**
BFS(29)	**DDRURULDRDLULURRDLLDRR**
BnB(29)	**DDRURULDRDLULURRDLLDRR**
BFS(30)	**DRRUULLDRDRUULDDRULDLURDR**
BnB(30)	**DRRUULLDRDRUULDDRULDLURDR**

Table 2: Solution Move Sets of 30 different test cases on 2 graph search algorithms

Algo(Test)	Trial: Time [ss.sss] (Nodes Checked)									
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
BFS(1)	0.080 (17512)	0.054 (17512)	0.040 (17512)	0.042 (17512)	0.047 (17512)	0.031 (17512)	0.034 (17512)	0.035 (17512)	0.061 (17512)	0.046 (17512)
BnB(1)	0.001 (42)	0.000 (42)	0.000 (42)	0.000 (42)	0.000 (42)	0.000 (42)	0.000 (42)	0.000 (42)	0.000 (42)	0.000 (42)
BFS(2)	0.633 (180689)	0.651 (180689)	0.525 (180689)	0.695 (180689)	0.547 (180689)	0.527 (180689)	0.538 (180689)	0.528 (180689)	0.515 (180689)	0.520 (180689)
BnB(2)	0.034 (4241)	0.025 (4241)	0.025 (4241)	0.024 (4241)	0.024 (4241)	0.025 (4241)	0.025 (4241)	0.024 (4241)	0.025 (4241)	0.025 (4241)
BFS(3)	0.143 (70122)	0.141 (70122)	0.144 (70122)	0.148 (70122)	0.143 (70122)	0.142 (70122)	0.141 (70122)	0.141 (70122)	0.141 (70122)	0.142 (70122)
BnB(3)	0.000 (138)	0.000 (138)	0.000 (138)	0.000 (138)	0.000 (138)	0.000 (138)	0.000 (138)	0.000 (138)	0.000 (138)	0.000 (138)
BFS(4)	0.000 (408)	0.000 (408)	0.000 (408)	0.000 (408)	0.000 (408)	0.000 (408)	0.000 (408)	0.000 (408)	0.000 (408)	0.000 (408)
BnB(4)	0.000 (10)	0.000 (10)	0.000 (10)	0.000 (10)	0.000 (10)	0.000 (10)	0.000 (10)	0.000 (10)	0.000 (10)	0.000 (10)
BFS(5)	0.244 (104976)	0.235 (104976)	0.252 (104976)	0.238 (104976)	0.240 (104976)	0.241 (104976)	0.237 (104976)	0.245 (104976)	0.236 (104976)	0.251 (104976)
BnB(5)	0.009 (2241)	0.009 (2241)	0.009 (2241)	0.009 (2241)	0.009 (2241)	0.009 (2241)	0.009 (2241)	0.009 (2241)	0.009 (2241)	0.009 (2241)
BFS(6)	0.238 (106086)	0.238 (106086)	0.241 (106086)	0.244 (106086)	0.251 (106086)	0.238 (106086)	0.259 (106086)	0.238 (106086)	0.239 (106086)	0.242 (106086)
BnB(6)	0.002 (941)	0.003 (941)	0.003 (941)	0.002 (941)	0.003 (941)	0.003 (941)	0.003 (941)	0.003 (941)	0.003 (941)	0.003 (941)
BFS(7)	0.130 (64334)	0.131 (64334)	0.145 (64334)	0.132 (64334)	0.131 (64334)	0.130 (64334)	0.146 (64334)	0.131 (64334)	0.130 (64334)	0.131 (64334)
BnB(7)	0.003 (1063)	0.003 (1063)	0.003 (1063)	0.003 (1063)	0.003 (1063)	0.003 (1063)	0.003 (1063)	0.003 (1063)	0.003 (1063)	0.003 (1063)
BFS(8)	0.271 (112731)	0.259 (112731)	0.281 (112731)	0.271 (112731)	0.279 (112731)	0.257 (112731)	0.257 (112731)	0.260 (112731)	0.256 (112731)	0.263 (112731)
BnB(8)	0.009 (2203)	0.009 (2203)	0.009 (2203)	0.009 (2203)	0.009 (2203)	0.008 (2203)	0.009 (2203)	0.008 (2203)	0.008 (2203)	0.009 (2203)
BFS(9)	0.033 (18612)	0.033 (18612)	0.033 (18612)	0.033 (18612)	0.033 (18612)	0.033 (18612)	0.034 (18612)	0.033 (18612)	0.033 (18612)	0.033 (18612)
BnB(9)	0.000 (154)	0.000 (154)	0.000 (154)	0.000 (154)	0.000 (154)	0.000 (154)	0.000 (154)	0.000 (154)	0.000 (154)	0.000 (154)
BFS(10)	0.073 (37916)	0.072 (37916)	0.073 (37916)	0.072 (37916)	0.071 (37916)	0.071 (37916)	0.071 (37916)	0.078 (37916)	0.072 (37916)	0.071 (37916)
BnB(10)	0.000 (36)	0.000 (36)	0.000 (36)	0.000 (36)	0.000 (36)	0.000 (36)	0.000 (36)	0.000 (36)	0.000 (36)	0.000 (36)
BFS(11)	0.255 (109137)	0.257 (109137)	0.245 (109137)	0.264 (109137)	0.247 (109137)	0.254 (109137)	0.253 (109137)	0.249 (109137)	0.286 (109137)	0.248 (109137)
BnB(11)	0.001 (570)	0.001 (570)	0.001 (570)	0.001 (570)	0.001 (570)	0.001 (570)	0.001 (570)	0.001 (570)	0.001 (570)	0.001 (570)
BFS(12)	0.513 (180738)	0.528 (180738)	0.540 (180738)	0.553 (180738)	0.565 (180738)	0.530 (180738)	0.518 (180738)	0.521 (180738)	0.536 (180738)	0.531 (180738)
BnB(12)	0.018 (3850)	0.018 (3850)	0.018 (3850)	0.019 (3850)	0.018 (3850)	0.018 (3850)	0.019 (3850)	0.018 (3850)	0.018 (3850)	0.019 (3850)
BFS(13)	0.026 (14534)	0.026 (14534)	0.026 (14534)	0.026 (14534)	0.025 (14534)	0.025 (14534)	0.025 (14534)	0.026 (14534)	0.026 (14534)	0.026 (14534)
BnB(13)	0.000 (310)	0.000 (310)	0.000 (310)	0.000 (310)	0.000 (310)	0.000 (310)	0.000 (310)	0.000 (310)	0.000 (310)	0.000 (310)
BFS(14)	0.065 (34994)	0.066 (34994)	0.066 (34994)	0.075 (34994)	0.066 (34994)	0.066 (34994)	0.066 (34994)	0.066 (34994)	0.066 (34994)	0.065 (34994)
BnB(14)	0.000 (177)	0.000 (177)	0.000 (177)	0.000 (177)	0.000 (177)	0.000 (177)	0.000 (177)	0.000 (177)	0.000 (177)	0.000 (177)
BFS(15)	0.492 (176623)	0.486 (176623)	0.491 (176623)	0.492 (176623)	0.492 (176623)	0.489 (176623)	0.484 (176623)	0.486 (176623)	0.503 (176623)	0.493 (176623)
BnB(15)	0.019 (3924)	0.019 (3924)	0.018 (3924)	0.019 (3924)	0.019 (3924)	0.018 (3924)	0.018 (3924)	0.018 (3924)	0.019 (3924)	0.018 (3924)
BFS(16)	0.365 (145325)	0.380 (145325)	0.362 (145325)	0.368 (145325)	0.362 (145325)	0.364 (145325)	0.383 (145325)	0.362 (145325)	0.370 (145325)	0.362 (145325)
BnB(16)	0.001 (504)	0.001 (504)	0.001 (504)	0.001 (504)	0.001 (504)	0.001 (504)	0.001 (504)	0.001 (504)	0.001 (504)	0.001 (504)
BFS(17)	0.123 (60262)	0.120 (60262)	0.120 (60262)	0.120 (60262)	0.129 (60262)	0.124 (60262)	0.120 (60262)	0.120 (60262)	0.136 (60262)	0.122 (60262)
BnB(17)	0.000 (231)	0.000 (231)	0.000 (231)	0.000 (231)	0.000 (231)	0.000 (231)	0.000 (231)	0.000 (231)	0.000 (231)	0.000 (231)
BFS(18)	0.032 (17742)	0.032 (17742)	0.031 (17742)	0.031 (17742)	0.031 (17742)	0.031 (17742)	0.031 (17742)	0.031 (17742)	0.031 (17742)	0.031 (17742)
BnB(18)	0.000 (63)	0.000 (63)	0.000 (63)	0.000 (63)	0.000 (63)	0.000 (63)	0.000 (63)	0.000 (63)	0.000 (63)	0.000 (63)
BFS(19)	0.169 (78435)	0.162 (78435)	0.160 (78435)	0.177 (78435)	0.159 (78435)	0.160 (78435)	0.161 (78435)	0.165 (78435)	0.160 (78435)	0.161 (78435)
BnB(19)	0.000 (210)	0.000 (210)	0.000 (210)	0.000 (210)	0.000 (210)	0.000 (210)	0.000 (210)	0.000 (210)	0.000 (210)	0.000 (210)
BFS(20)	0.525 (181385)	0.524 (181385)	0.519 (181385)	0.519 (181385)	0.525 (181385)	0.528 (181385)	0.530 (181385)	0.525 (181385)	0.518 (181385)	0.516 (181385)
BnB(20)	0.055 (7031)	0.055 (7031)	0.055 (7031)	0.055 (7031)	0.055 (7031)	0.056 (7031)	0.055 (7031)	0.056 (7031)	0.055 (7031)	0.054 (7031)
BFS(21)	0.164 (79344)	0.181 (79344)	0.165 (79344)	0.164 (79344)	0.166 (79344)	0.165 (79344)	0.165 (79344)	0.164 (79344)	0.176 (79344)	0.164 (79344)
BnB(21)	0.001 (594)	0.001 (594)	0.001 (594)	0.001 (594)	0.001 (594)	0.001 (594)	0.001 (594)	0.001 (594)	0.001 (594)	0.001 (594)
BFS(22)	0.433 (162747)	0.434 (162747)	0.434 (162747)	0.426 (162747)	0.442 (162747)	0.429 (162747)	0.429 (162747)	0.432 (162747)	0.427 (162747)	0.429 (162747)
BnB(22)	0.009 (2424)	0.009 (2424)	0.009 (2424)	0.009 (2424)	0.009 (2424)	0.009 (2424)	0.009 (2424)	0.009 (2424)	0.009 (2424)	0.009 (2424)
BFS(23)	0.003 (2155)	0.003 (2155)	0.003 (2155)	0.003 (2155)	0.003 (2155)	0.003 (2155)	0.003 (2155)	0.003 (2155)	0.003 (2155)	0.003 (2155)
BnB(23)	0.000 (55)	0.000 (55)	0.000 (55)	0.000 (55)	0.000 (55)	0.000 (55)	0.000 (55)	0.000 (55)	0.000 (55)	0.000 (55)
BFS(24)	0.359 (144707)	0.360 (144707)	0.360 (144707)	0.368 (144707)	0.368 (144707)	0.368 (144707)	0.356 (144707)	0.355 (144707)	0.368 (144707)	0.357 (144707)
BnB(24)	0.008 (2182)	0.008 (2182)	0.008 (2182)	0.008 (2182)	0.008 (2182)	0.008 (2182)	0.008 (2182)	0.008 (2182)	0.008 (2182)	0.008 (2182)
BFS(25)	0.269 (112664)	0.258 (112664)	0.267 (112664)	0.258 (112664)	0.275 (112664)	0.256 (112664)	0.256 (112664)	0.256 (112664)	0.255 (112664)	0.263 (112664)
BnB(25)	0.003 (1215)	0.003 (1215)	0.003 (1215)	0.003 (1215)	0.003 (1215)	0.003 (1215)	0.003 (1215)	0.003 (1215)	0.003 (1215)	0.004 (1215)
BFS(26)	0.002 (1193)	0.002 (1193)	0.002 (1193)	0.002 (1193)	0.002 (1193)	0.002 (1193)	0.002 (1193)	0.002 (1193)	0.002 (1193)	0.002 (1193)
BnB(26)	0.000 (13)	0.000 (13)	0.000 (13)	0.000 (13)	0.000 (13)	0.000 (13)	0.000 (13)	0.000 (13)	0.000 (13)	0.000 (13)
BFS(27)	0.432 (156154)	0.399 (156154)	0.399 (156154)	0.422 (156154)	0.399 (156154)	0.398 (156154)	0.424 (156154)	0.398 (156154)	0.401 (156154)	0.420 (156154)
BnB(27)	0.005 (1703)	0.005 (1703)	0.005 (1703)	0.006 (1703)	0.005 (1703)	0.005 (1703)	0.005 (1703)	0.005 (1703)	0.005 (1703)	0.006 (1703)
BFS(28)	0.302 (127646)	0.322 (127646)	0.301 (127646)	0.320 (127646)	0.302 (127646)	0.313 (127646)	0.312 (127646)	0.318 (127646)	0.302 (127646)	0.307 (127646)
BnB(28)	0.001 (465)	0.001 (465)	0.001 (465)	0.001 (465)	0.001 (465)	0.001 (465)	0.001 (465)	0.001 (465)	0.001 (465)	0.001 (465)
BFS(29)	0.203 (94779)	0.200 (94779)	0.204 (94779)	0.201 (94779)	0.219 (94779)	0.201 (94779)	0.202 (94779)	0.213 (94779)	0.201 (94779)	0.203 (94779)
BnB(29)	0.000 (184)	0.000 (184)	0.000 (184)	0.000 (184)	0.000 (184)	0.000 (184)	0.000 (184)	0.000 (184)	0.000 (184)	0.000 (184)
BFS(30)	0.415 (159480)	0.415 (159480)	0.412 (159480)	0.413 (159480)	0.415 (159480)	0.416 (159480)	0.412 (159480)	0.407 (159480)	0.431 (159480)	0.410 (159480)
BnB(30)	0.008 (2263)	0.008 (2263)	0.008 (2263)	0.008 (2263)	0.008 (2263)	0.008 (2263)	0.008 (2263)	0.008 (2263)	0.008 (2263)	0.008 (2263)

Table 3: Results of 30 different test cases on 2 graph search algorithms with 10 trials each

5 Conclusion

In this research, two algorithms are implemented with solving the N-Puzzle problem in mind. Using different test cases, these algorithms are experimented and compared for results. The experiment tests the 8-puzzle version of the problem, uses 30 different test cases and run 10 trials for each test case.

One of these algorithms is the Breadth First Search algorithm (BFS). BFS is a simple graph search algorithm that finds nodes one node depth at a time. It involves carefully searching for solutions or nodes branch by branch. The downside to the way BFS works is that it is very slow and checks too many nodes when traversing.

Another of these algorithms is the Branch and Bound Graph Search algorithm. The cost function employed in the implementation uses edge traversal (move list) length and the sum of Manhattan distances (Taxicab geometry) of each tile from its original position. The advantage of using Branch and Bound is an increase in search speed (similar to fuzzy matching) and a decrease of node checks. It performs way faster than the BFS implementation. The disadvantage of this implementation to BFS is that it sometimes does not generate the most optimal solution for the N-Puzzle problem.

Further improvements to the research would entail having to improve the cost function of the BnB implementation, trying to do two-way BFS where traversal which would include solving from original to shuffled arrangement of tiles or increasing the tiles to solve 15 puzzle or 24 puzzle problems.

In conclusion, it can be said that the two types of algorithm differ to one another by their use case or how they are meant to be used. The BFS,

despite being slower than BnB, is more optimal and accurate than BnB. This implementation should be used for critical and accurate work where correctness is very important. On the other hand, the BnB, despite not generating the most optimal solution, works so much faster than BFS. This implementation should be used for work that do not require accuracy and correctness but to create a solution in the least amount of time.

References

[1] GeeksforGeeks 8 puzzle problem using branch and bound. https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/. Accessed: 2019-04-27.

[2] GeeksforGeeks breadth first search or bfs for a graph. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/. Accessed: 2019-04-27.

[3] GeeksforGeeks hashset in java. https://www.geeksforgeeks.org/hashset-in-java/. Accessed: 2019-04-27.

[4] Stanford Artificial Intelligence Laboratory : CS121 heuristic (informed) search. http://ai.stanford.edu/~latombe/cs121/2011/slides/D-heuristic-search.pdf. Accessed: 2019-04-27.

[5] Study.com taxicab geometry: History and formula. https://study.com/academy/lesson/taxicab-geometry-history-formula.html. Accessed: 2019-04-27.

[6] Jerry Slocum and Dic Sonneveld. The 15 puzzle. 2006.