A Comparison of Different Search Algorithms for the 8-Puzzle Problem

Elia Karimi Sisan <ekarimis@sfu.ca>,

S M Shahjiban Munjoreen <smunjore@sfu.ca>,

Mandeepa Mashhura <mmashhur@sfu.ca>

CMPT417 – Dr. Hang Ma
Simon Fraser University, Burnaby BC V5A 1S6, Canada

Short Project Name: PuzzleProblemSearchAlgorithms

Abstract:

On a large scale, a problem solving agent is an intelligent agent that decides what actions and states to consider in order to achieve the goal objective. The 8 Puzzle is a special kind of problem solving agent. The 8-puzzle is a game that consists of 9 tiles structured on a 3x3 grid, with each of the tiles having a value between 0 to 8 and the 0 tile being the "blank" tile. At the start of the game, we are given an "Initial State" that consists of the tiles misplaced from their intended locations, and our objective is to move the tiles around into different positions utilizing the blank square in order to achieve the "Goal State" given by the program, as shown in *Figure One*. The underlying purpose of the implementation of the 8-puzzle game is to solve any given Initial Configuration by making the least number of moves in order to get to the Goal Configuration.

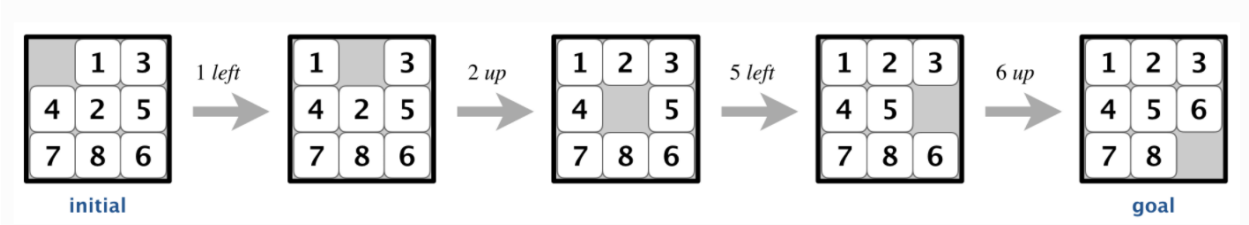


Figure One

Keywords:
Artificial Intelligence, Eight Puzzle, Search Algorithms, A-Star Search (*A**), Bidirectional Search , Dijkstra's Algorithm (*Dijkstra*), Depth-First Search (*DFS)*, Iterative-Deepening Depth-First Search (*IDDFS*), and Breadth First Search (*BFS*).

1. Introduction

In this project, our goal was to implement and test various instances of the 8 puzzle problem with different search algorithms. To formulate our problem, we considered four different actions in our algorithms, namely: Moving blank tile up, moving blank tile down, moving blank tile left, and moving blank tile right. Each state, or alternatively, node in our search tree, represents the arrangement of the tiles in its current configuration as a result of the action taken from the predecessor state. For example, *Figure Two* shows an 8-puzzle instance consisting of an initial state of [1,2,5,3,4,0,6,7,8], and its possible action it has from its current state in order to make the best possible unit-cost move in order to get to its desired goal state.

To achieve our goal of moving from the Initial State to Goal State, we implemented 6 different search algorithms, namely: A-Star Search (*A**), Bidirectional Search (*Bidirectional*), Dijkstra's Algorithm (*Dijkstra*), Depth-First Search (*DFS)*, Iterative-Deepening Depth-First Search (*IDDFS*), and Breadth First Search (*BFS*). In order to experiment with the algorithms, we will present the time and space complexity of the individual algorithms along with their respective benchmarked running times for completing the respective search algorithms. Additionally, we will compare the algorithms by determining how many moves are required to get to the goal state along with reporting the number of expanded nodes (nodes popped off the queue). Another aspect we will report is the optimality and completeness of the algorithms. Lastly, we will compare and discuss our results and the particular benefits and drawbacks from the pseudocodes of our proposed algorithms.

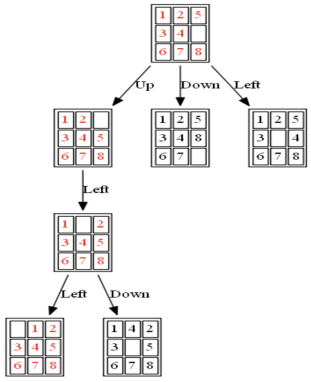


Figure Two

2. Implementation

With the notion of the 8-puzzle explained in the introduction, we will now proceed to highlight the key aspects of the six algorithms we have implemented, utilizing the AIMA repository for its helper functions. In general, all of the algorithms initiate from their initial state and generate a maximum of four states (up,down,left,right). Subsequently, they select one of the adjacent states to expand on iteratively, and they repeat the procedure until the respective goal state is found. However, choosing the state to expand on is the key difference that lies within the algorithms, which we will cover in detail.

2.1. A-Star Search (A*)

A* search takes into account the cost of the paths as well as the heuristic. In this case we used the number of misplaced tiles in a state to calculate the heuristic. We add that to the cost to reach the current state and that is our f(n). To find the shortest path, we start by expanding our first node. The neighboring nodes are stored into a priority queue according to their heuristic value. Then we repeatedly pop the first node of the priority queue and expand it until we reach our goal. To create the puzzles and their states, we used the AIMA repository, but the A* search algorithm itself was written from scratch. We followed the pseudo code in *Figure Four*. While testing we realized some puzzle instances were taking too long to run and were exhausting signing=ficant memory therefore we chose to optimize it by exiting if the algorithm had 20,000 nodes and still hadn't reached the goal. In such cases, we called the puzzle instance unsolvable.

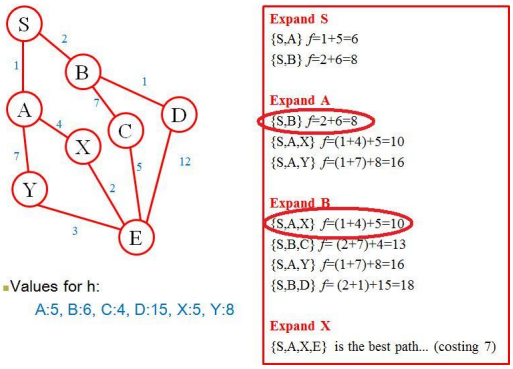


Figure Three

```
make an openlist containing only the starting node
make an empty closed list
while (the destination node has not been reached):
    consider the node with the lowest f score in the open list
    if (this node is our destination node) :
        we are finished
    if not:
        put the current node in the closed list and look at all of its neighbors
        for (each neighbor of the current node):
            if (neighbor has lower g value than current and is in the closed list) :
                replace the neighbor with the new, lower, g value
                current node is now the neighbor's parent
            else if (current g value is lower and this neighbor is in the open list ) :
                replace the neighbor with the new, lower, g value
                change the neighbor's parent to our current node

            else if this neighbor is not in both lists:
                add it to the open list and set its g
```

Figure Four

2.2. Dijkstra's Algorithm (DIJKSTRA)

Dijkstra's algorithm finds the shortest path to the destination by following the paths with the smallest cost. It has no way of knowing if it is getting any closer to the goal, therefore in our case, is very suboptimal, as shown in the illustration in *Figure Five*. We roughly followed the pseudocode from *Figure Six*. This algorithm starts by expanding the first node then choosing the neighboring node with the smallest path cost. It repeats this process until reaching the goal. Another huge drawback in our case is the fact that all actions (up, down, left, right) cost the same. So there is no need for a priority queue. This further slows the process down, making most puzzle instances unsolvable.

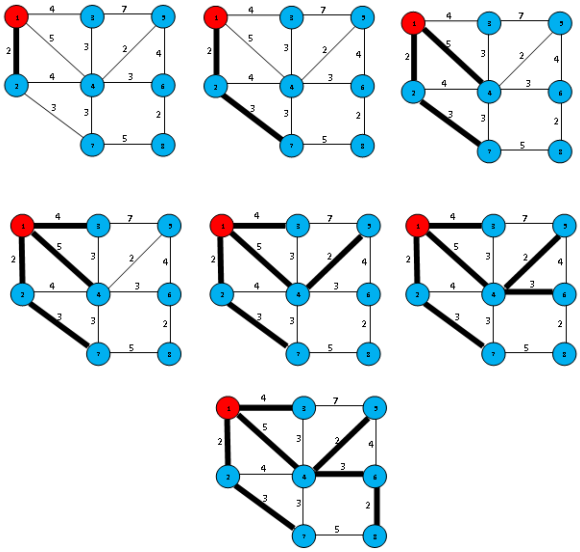


Figure Five

```
function Dijkstra(Graph, source):
       dist[source]  := 0
       for each vertex v in Graph:
           if v ≠ source
               dist[v]  := infinity
           add v to Q

      while Q is not empty:
          v := vertex in Q with min dist[v]
          remove v from Q

          for each neighbor u of v:
              alt := dist[v] + length(v, u)
              if alt < dist[u]:
                  dist[u]  := alt

      return dist[]
  end function
```

Figure Six

2.3. Depth-First Search (DFS)

With the nature of Depth-First Search being an uninformed search algorithm as it does not use any domain specific knowledge, its main purpose is to continuously expand the deepest node in a search tree until it cannot expand and then backtrack. To see an example of Depth-First Search, refer to *Figure Seven* where it portrays a search initiating from the root node (1) and progressing as far as it can in the left-most node in the left subtree (4) before backtracking and progressing the search from there. The frontier of our Depth-First Search was implemented using a Stack in a Last-In-First-Out manner. Originally, we implemented Depth-First-Search utilizing the EightPuzzle class in the AIMA Repository, but failed to gain any significant results as the time to complete the search was exhausted (as DFS was time-limited in that implementation). To alleviate this issue, we restructured the 8-puzzle class into its respective actions that permissible at each state (Action Up, Action Down, Action Left, Action Right) and implemented the algorithm utilizing Numpy Arrays, thus giving us a major improvement in terms of execution time as the access in reading and writing items in faster in NumPy rather than with plain Python Lists. In order to initiate the search, we instantiated the top most node in our search tree, namely the root node, and ensured the nodes are searched in a depth-first manner, resembling the pseudocode found in *Figure Eight.* The goal test condition for our Depth-First Search was to check whether the current state is the goal state, and if it is, the algorithm backtracks and prints the required results for the purposes of the search.

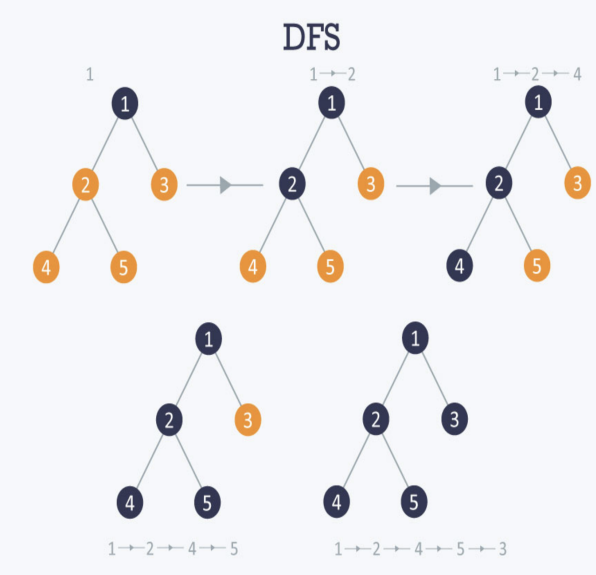


Figure Seven

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Figure Eight

2.4. Iterative-Deepening Depth-First Search (IDDFS)

As Iterative-Deepening Depth-First Search is merely an extension of Depth-First Search, it attains many of the properties that are present in Depth-First Search such as being an uninformed search algorithm as it does not use any domain specific knowledge. In order to implement the IDDFS algorithm, we first had to implement the Depth-Limited Search Algorithm. The Depth-Limited Search algorithm was simple to implement as it is merely an implementation of Depth-First search with an addition of an extra parameter *l* which is the threshold for the search's depth. As a result of imposing this threshold, only those nodes with depth less than or equal to *l* are expanded. However, one problem with this search is that we do not know the depth limit apriori. A solution to this problem is to make the search adaptive by iterating through each level of the search tree. This is done via a variation of the Depth-Limited Search algorithm, namely, Iterative-Deepening Depth-First Search. Its main purpose is to perform Depth-First Search for each level of the search tree until a good is found, which then the algorithm will backtrack and display the result. To see an example of Iterative-Deepening Depth-First Search, refer to *Figure Nine* where it portrays a search initiating from the root node and progressing level-by-level and consequently backtracking when the goal node is found. Similar to DFS, the frontier of our Depth-First Search was implemented using a Stack in a Last-In-First-Out manner. In order to initiate the search, we instantiated the top most node in our search tree, namely the root node, and ensured the nodes are searched in a depth-first manner, resembling the pseudocode found in *Figure Ten.* We used the same goal test condition for our Iterative-Deepening Depth-First Search to check whether the current state is the goal state, and if it is, the algorithm backtracks and prints the required results for the purposes of the search.

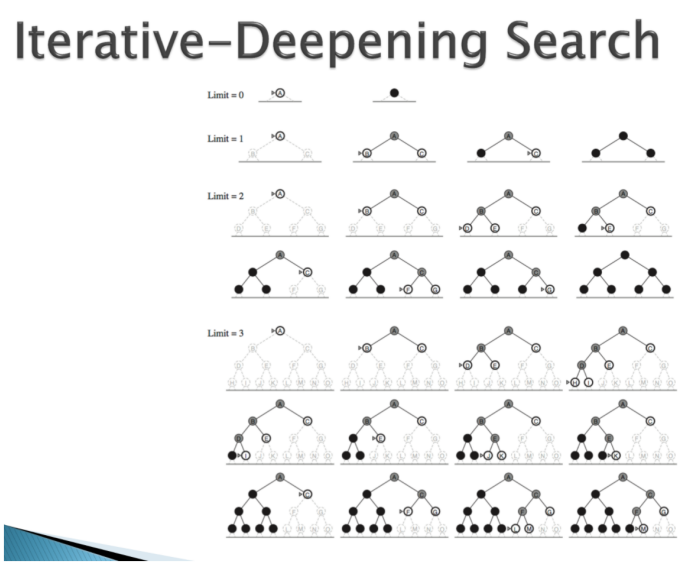


Figure Nine

```
function IDDFS(root) is
    for depth from 0 to ∞ do
        found, remaining ← DLS(root, depth)
        if found ≠ null then
            return found
        else if not remaining then
            return null

function DLS(node, depth) is
    if depth = 0 then
        if node is a goal then
            return (node, true)
        else
            return (null, true)    (Not found, but may have children)

    else if depth > 0 then
        any_remaining ← false
        foreach child of node do
            found, remaining ← DLS(child, depth-1)
            if found ≠ null then
                return (found, true)
            if remaining then
                any_remaining ← true    (At least one node found at depth, let IDDFS deepen)
        return (null, any_remaining)
```

Figure Ten

2.5. Bidirectional Search

Bidirectional search is a search algorithm that finds the shortest path from an initial vertex to a goal vertex in a directed graph. It runs two searches simultaneously as seen in *figure eleven*: one forward from the initial state, and one backward from the goal, stopping when the two meet. This approach in most cases is faster: for instance, in a simplified model of search problem complexity in which both searches expand a tree with branching factor b, and the distance from start to goal is d, each of the two searches has complexity $O(b^{d/2})$, and the sum of these two search times is much less than the $O(b^d)$ complexity that would result from a single search from the beginning to the goal. The implementation followed from the pseudocode in *Figure Twelve.*

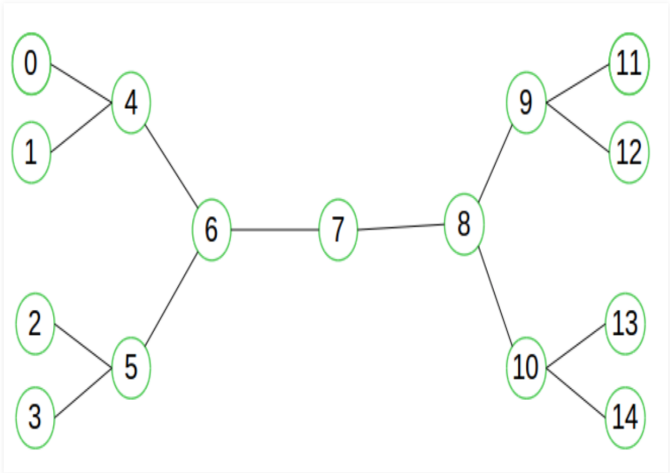


Figure Eleven

```
BIDIRECTIONAL_SEARCH
 1   Q_I.Insert(x_I) and mark x_I as visited
 2   Q_G.Insert(x_G) and mark x_G as visited
 3   while Q_I not empty and Q_G not empty do
 4       if Q_I not empty
 5           x ← Q_I.GetFirst()
 6           if x = x_G or x ∈ Q_G
 7               return SUCCESS
 8           forall u ∈ U(x)
 9               x' ← f(x, u)
10               if x' not visited
11                   Mark x' as visited
12                   Q_I.Insert(x')
13               else
14                   Resolve duplicate x'
15       if Q_G not empty
16           x' ← Q_G.GetFirst()
17           if x' = x_I or x' ∈ Q_I
18               return SUCCESS
19           forall u^-1 ∈ U^-1(x')
20               x ← f^-1(x', u^-1)
21               if x not visited
22                   Mark x as visited
23                   Q_G.Insert(x)
24               else
25                   Resolve duplicate x
26   return FAILURE
```

Figure Twelve

2.6. Breadth First Search (BFS)

Breadth first search (BFS) is an uninformed search which continuously searches all the nodes on a depth level before jumping to the next one. In *Figure Thirteen*, it can be seen that the BFS starts at the root and searches all its neighbouring nodes from left to right of the second depth level before advancing to the third depth level. In this way BFS continues searching all the nodes until it finds the goal node. We used a queue (First In First Out) to implement BFS which contained the frontier along which the algorithm will search. Then we checked whether a goal node has been discovered before enqueueing the node rather than delaying the check until the vertex is dequeued from the queue. The implementation of BFS was based on the pseudocode shown in *Figure Fourteen*.

Figure Thirteen

```
procedure BFS(G, root) is
    let Q be a queue
    label root as discovered
    Q.enqueue(root)
    while Q is not empty do
        v := Q.dequeue()
        if v is the goal then
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered then
                label w as discovered
                Q.enqueue(w)
```

Figure Fourteen

3. Methodology:

For this project, our primary area of investigation is the performance of the algorithms on 20 known 8-Puzzle instances. In particular, our methods of evaluating the respective search algorithms are by measuring their Completeness (determining if there is a solution, is the algorithm guaranteed to find it), Optimality (determining if the algorithms find a goal node with the lowest path cost), Time Complexity (the execution time of the respective searches) and the Space Complexity (determining the amount of memory needed to complete the search). The problem instance we used for the purposes of investigating our algorithms were taken from the published paper “Intelligent System Design Using Hyper-Heuristics” by N.Pillay, which consisted of both easy puzzles with 4 optimum moves to get to its goal state, to more challenging puzzles that required 31 optimum moves to get to its goal state, shown in *Figure 15*. Additionally, in our puzzle instances, we had both unscrambled goal states, for example [1,2,3,4,5,6,7,8,0] and scrambled goal states such as [2,8,1,4,6,3,0,7,5], which we later

determined upon execution of our algorithms that there is no real evidence that the scrambled goal instances is easier to solve than the unscrambled goal instances. Out of the six search algorithms we implemented, the only algorithm that was not complete as it did not find a solution for most puzzle instances was Dijkstra's Algorithm, as mentioned later in the report.

123804765	134862705
123804765	281043765
123804765	281463075
134805726	123804765
231708654	123804765
231804765	123804765
123804765	231804765
283104765	123804765
876105234	123804765
867254301	123456780
647850321	123456780
123804765	567408321
806547231	012345678
641302758	012345678
158327064	012345678
328451670	012345678
035428617	012345678
725310648	012345678
412087635	123456780
162573048	123456780

Figure Fifteen

Below is a table containing all of the known performance metrics of our algorithms, which will be used for comparison purposes in the *Conclusion* section of our paper:

Algorithm	**Completeness**	**Optimality**	**Time Complexity**	**Space Complexity**
DFS	Not complete as it may fail to find a goal node if it exists	Not optimal as it may first find goal nodes that are further from the root	*O(b^m)*, where *m* is the maximum depth of the search and *b* is the branching factor	*O(m),* where *m* is the maximum depth of the search
IDDFS	Complete when the branching factor *b* is finite	Optimal when path cost is non-decreasing function of the depth of the node	*O(b^m)*, where *m* is the maximum depth of the search and *b* is the branching factor	*O(m),* where *m* is the maximum depth of the search
Dijkstra	Not complete as all paths cost the same and there is	Not optimal as all paths cost the same	*O(v^2)*, where v is the number of nodes	*O(v)*, where v is the number of nodes

	no heuristic			
A* Search	Complete when the branching factor *b* is finite	Optimal	*O(b^m)*, where *m* is the maximum depth of the search and *b* is the branching factor	*O(m),* where *m* is the maximum depth of the search
BFS	Complete always guarantees to find a solution if one exists	Optimal	*O(b^d)*, where *d* is the solution depth of the search and *b* is the branching factor	*O(b^d),* where *d* is the solution depth of the search and *b* is the branching factor
Bidirectional	Complete when BFS is used	Optimal	*O(b^(d/2))*, where *d* is the solution depth of the search and *b* is the branching factor	*O(b^(d/2)),* where *d* is the solution depth of the search and *b* is the branching factor

4. Experimental Setup

We utilized Python 3.8.1, along with NumPy (for NumPy array purposes) for developing this project. We structured our program to be Command-Line Driven, where we allow the user to select one of 20 puzzle instances using the following command:
'python3 PuzzleProblem.py [Puzzle Instance Number (1-20)] [Search Algorithm Name (dfs, iddfs, dijkstra, astar, bfs, bidirectional)]'
Although we used multiple environments to benchmark our experiments, a commonality we had between them was that we were all using MacOs. Below is a table summarizing our experimental setup:

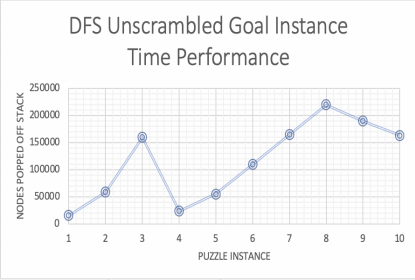
Environment	**Hardware**	**Processor**	**Memory**
Environment 1	MacBook Pro (13-inch, 2020, Four Thunderbolt 3 Ports)	2.3GHz quad-core Intel Core i7	16 GB
Environment 2	MacBook Pro (13-inch, 2019, Two Thunderbolt 3 ports)	1.7 GHz Quad-Core Intel Core i7	16 GB
Environment 3	MacBook Pro (13-inch, 2017, Two Thunderbolt 3 ports	2.3 GHz Dual-Core Intel Core i5	8 GB

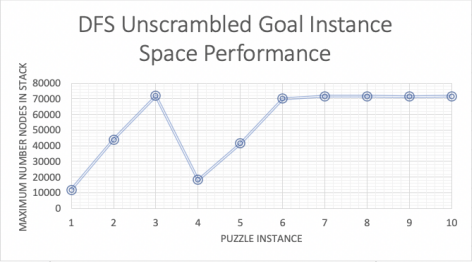
5. Experimental Results

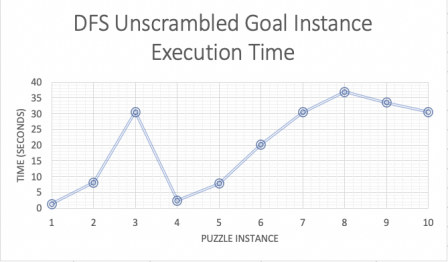
To graphically display our results, we made plots representing each run of our 20 puzzle instances. Below you will find plots for the respective algorithms containing the fundamental results to help us answer the questions mentioned in our report, which are partitioned into an unscrambled and scrambled goal instances section. For the written summary of the results, please see the following section.

Unscrambled Puzzle Results:

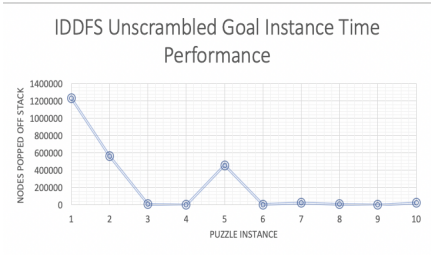
Depth First Search

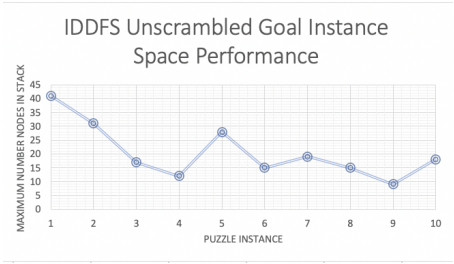


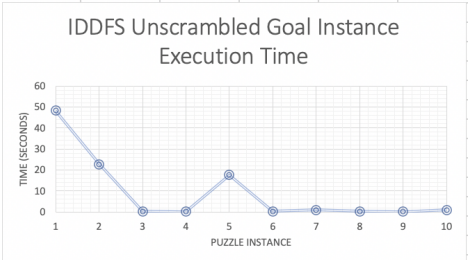




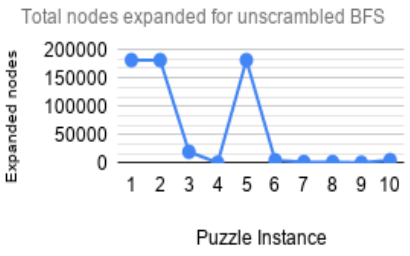
Iterative-Deepening Depth First Search

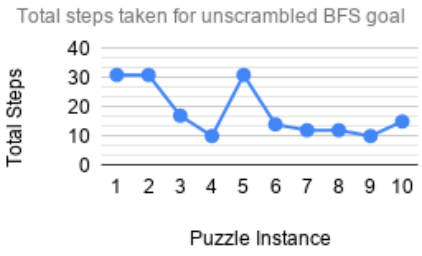


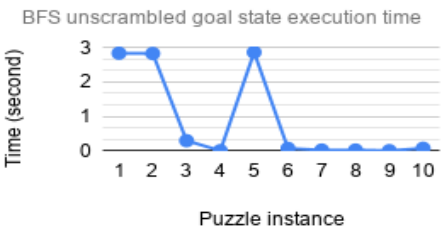




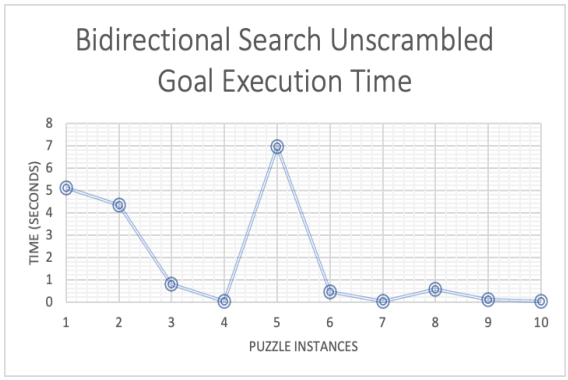
Breadth First Search



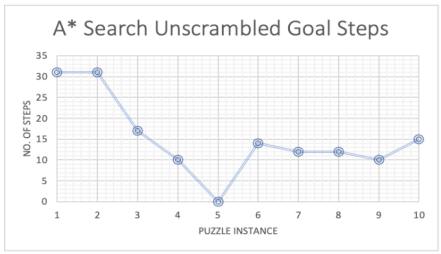


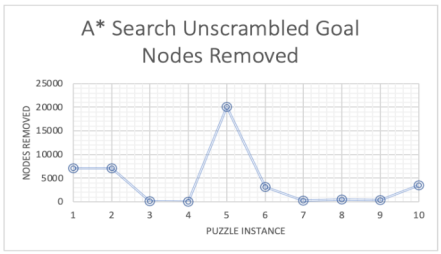


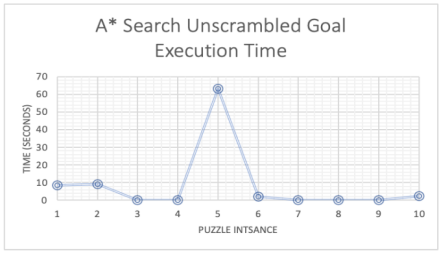
Bidirectional Search



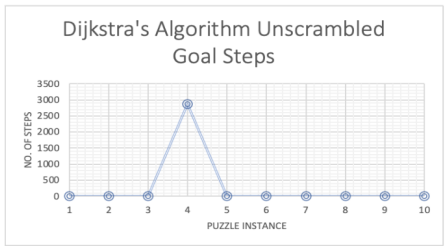
A* Search

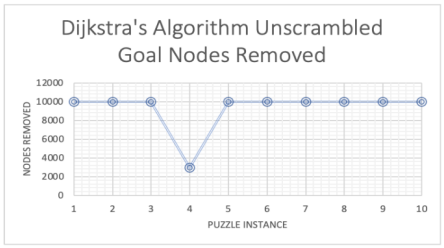


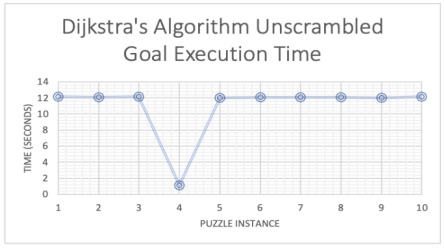




Dijkstra's Algorithm

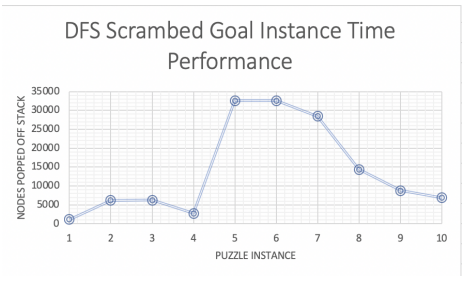


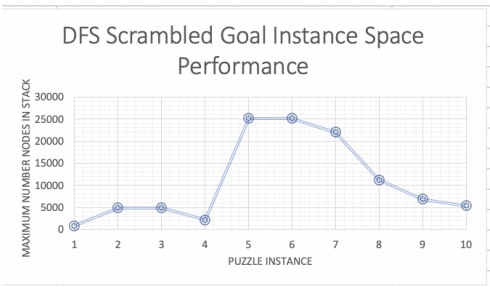


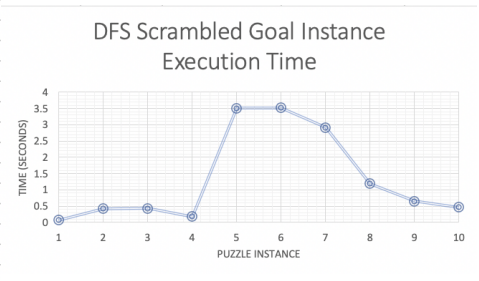


Scrambled Puzzle Results:

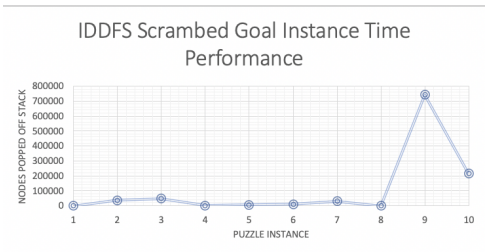
Depth First Search

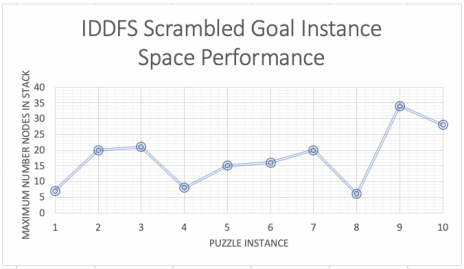


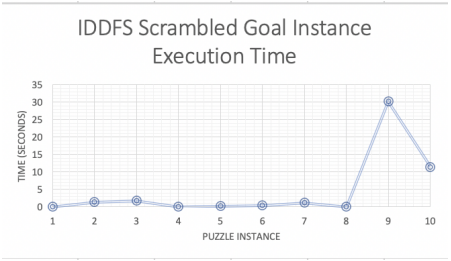




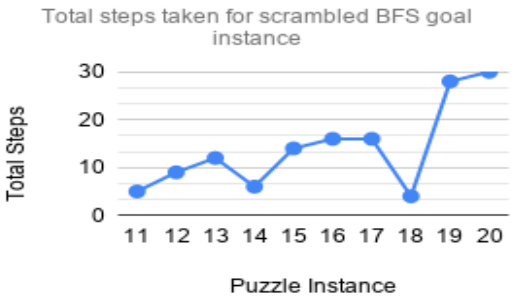
Iterative-Deepening Depth First Search

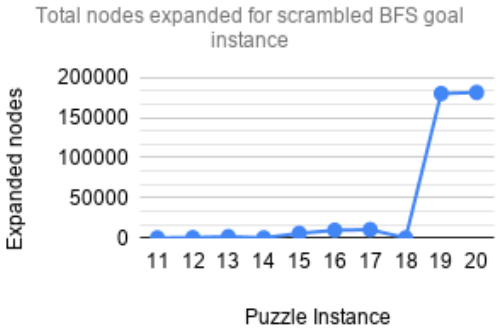


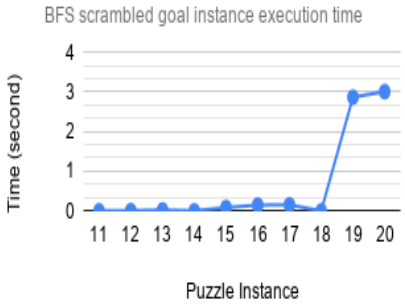




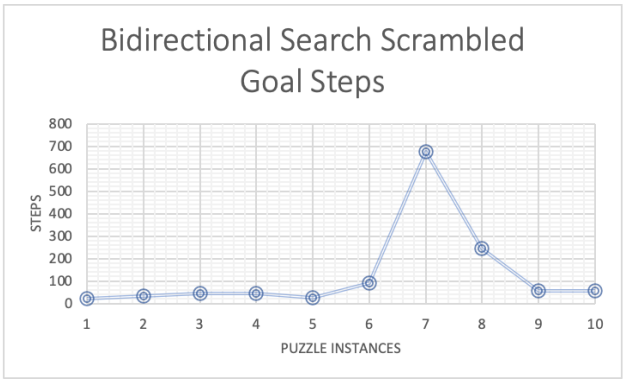
Breadth First Search

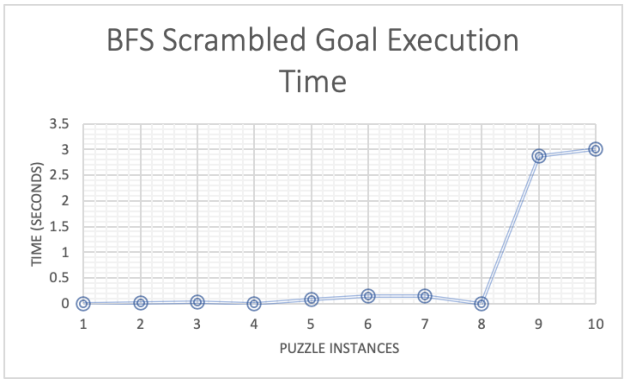




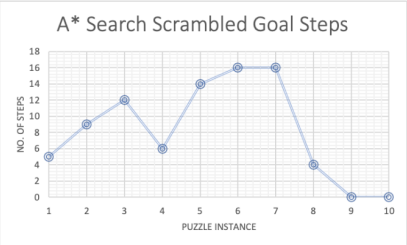


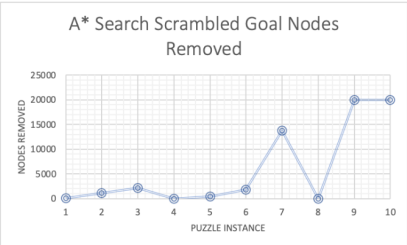
Bidirectional Search



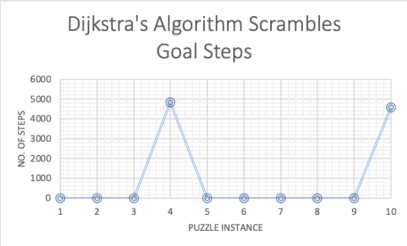


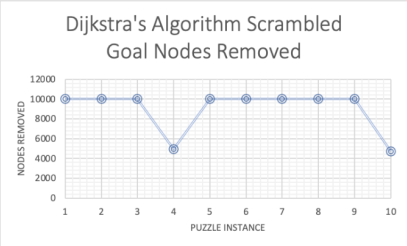
A* Search

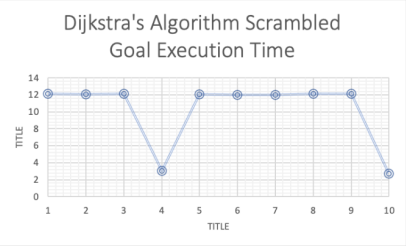




Dijkstra's Algorithm

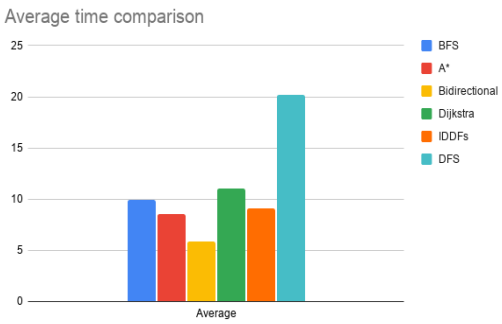


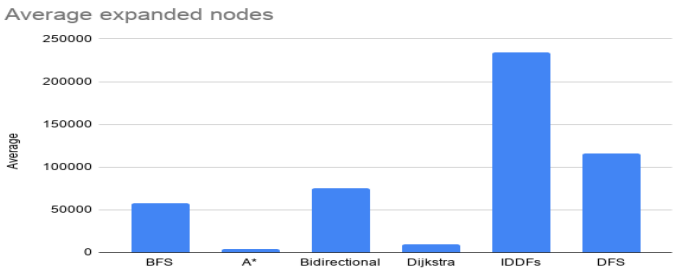




6. Conclusions

The algorithms without heuristics had the worst results. Most puzzle instances that used Dijkstra's algorithm were unsolvable, and the ones that were solvable provided suboptimal results. Another drawback was the fact that all actions cost the same. Therefore, algorithms that typically used priority queues provided suboptimal results. The algorithms with good heuristic provided the best results, for example: A*. The puzzle instances with unscrambles vs scrambled goals were not very different in terms of total steps taken or nodes removed. Dijkstra's algorithm was the worst by far. For BFS, DFS and IDDFS, even though the algorithms did not have any heuristics, they provided results. The results weren't very optimal as these algorithms tend to stop as soon as they reach the goal state. Below are the plots displaying the averages for our experiments.





References

Figure One: https://8-puzzle.readthedocs.io/en/latest/

Figure Two: https://sandipanweb.wordpress.com/2017/03/16/using-uninformed-informed-search-algorithms-to-solve-8-puzzle-n-puzzle/

Figure Three: https://stackoverflow.com/questions/5849667/a-search-algorithm

Figure Four: https://brilliant.org/wiki/a-star-search/

Figure Five: https://brilliant.org/wiki/dijkstras-short-path-finder/

Figure Six: https://brilliant.org/wiki/dijkstras-short-path-finder/

Figure Seven: https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Figure Eight: https://en.wikipedia.org/wiki/Depth-first_search

Figure Nine: Artificial Intelligence book by Peter Norvig and Stuart Russel

Figure Tent: https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search

Figure Eleven: : https://www.geeksforgeeks.org/bidirectional-search/

Figure Twelve: https://efficientcodeblog.wordpress.com/2017/12/13/bidirectional-search-two-end-bfs/

Figure Thirteen: https://hackr.io/blog/breadth-first-search-algorithm

Figure Fourteen: https://en.wikipedia.org/wiki/Breadth-first_search

Figure Fifteen: https://www.researchgate.net/figure/8-Puzzle-problem-instances_tbl1_280545587