Assignment No: 01

CSE-0408 Summer 2021

Kamrunnahar Somapti
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
somaptiesuborno@gmail.com

***Abstract*—The objective of this report is to present an overview of 4 common local search algorithms on the N-Queens problem and to display comparisons between the techniques as well as the parameters within them when applicable. Section I, outlines the N-Queens problem and explores the structure of each algorithm. Section II discusses the implementation of each approach while Section III debates the efficiency in time complexity of each algorithm in regard to data structures, parameters and more**

I. Introduction

An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares). One of the squares is empty. The object is to move to squares around into different positions and having the numbers displayed in the ”goal state”. The image to the left can be thought of as an unsolved initial state of the ”3 x 3” 8 puzzle.

II. Literature Review

The 8 puzzle program was written as a 2-person project for Dr. Tim Colburn’s Software Development course (CS2511) by Brian Spranger and Josh Richard. The assignment was to write a program that is intelligent enough to solve the 8-puzzle game in any configuration, in the least number of moves. We were also to provide a Graphical User Interface (GUI) to the user to make the program easier to use. The Intelligent engine was written in C++ and the GUI was written in Motif/X11 converted from standard C to use encapsulated Object Orientated Design Constructs

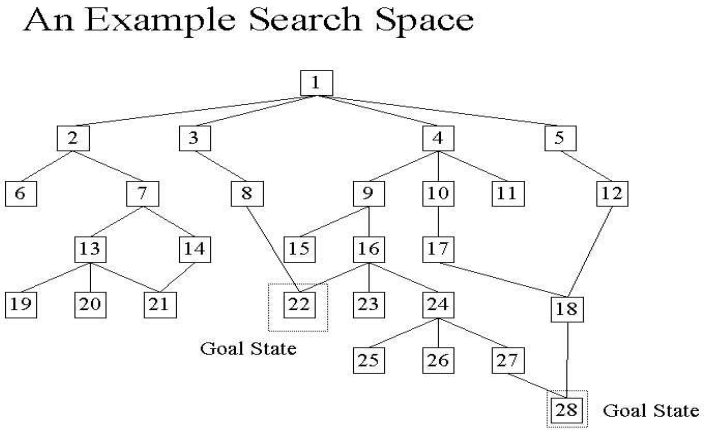
III. Problem Approach

The complexity of possible moves toward the final solution in a game like this is great. It can take an average computer great lengths of time to find the correct sequences for a particular configuration of the 8 puzzle game if the search method is ”blind”. The problem is how do you make the computer order the moves intelligently to find the shortest path to the winning game state? Solving a problem such as this can be done two ways:

- Guess through every possible state by doing a blind search of the state space.
- Implement a search operation that is guaranteed to find the shortest solution using ”informed methods” (How?)

IV. Heuristic Value Calculations:

In order for the game to be able to solve itself, we need to approach the solution with an idea of how to differentiate between a good move and a bad one. This is called a heuristic. Our heuristic was to order the moves based on the number of tiles out of place. We also took into consideration how far away the out of place tiles was from their correct position (called Manhattan distances). This entire process determines the heuristic value (H). We can then take H at any given time and successively find the path to a solution. But this will not provide us with the shortest solution. See figure:



V. A* Value and Priority Queues:

The answer to the question posed is to refine the approach used to determine the next move. If we think about fig. b again, we can see that winning state occurs in two (or more) places in the game tree (states 22, and 28). We can order these moves with the help of a special queue that orders based on priority. The priority is determined by taking the current heuristic value and adding it to the depth. This gives us our A* (a star) value

for a particular state. If the queue is ordered properly (lowest first), the next best move is always ”next in line” for expansion. This process will always yield the correct solution in the least number of moves, as long as the puzzle is initially a solvable state.

VI. A* Searching Algorithm:

The A* searching algorithm is a recursive algorithm that continuously calls itself until a winning state is found. When the game is designed, in order to randomly generate different games, start with the goal state and make many random moves on the state. This way, you can be sure to have a game state that can be solved This will keep you from having an algorithm that gets lost in its own thoughts and never evaluates to true (infinite recursion) The answer to the question posed is to refine the approach used to determine the next move. If we think about fig. b again, we can see that winning state occurs in two (or more) places in the game tree (states 22, and 28). We can order these moves with the help of a special queue that orders based on priority. The priority is determined by taking the current heuristic value and adding it to the depth. This gives us our A* (a star) value for a particular state. If the queue is ordered properly (lowest first), the next best move is always ”next in line” for expansion. This process will always yield the correct solution in the least number of moves, as long as the puzzle is initially a solvable state.

VII. Conclusion

An approach for solving the 8-puzzle problem has been proposed which minimizes the memory required while achieving optimum results. All the heuristics have been applied to a* algorithm to bring uniformity and the results were shown. Comparison of memory used makes sense only in a star algorithm. At last memory use of all the heuristics are compared. Ida* algorithm was implemented but only with one heuristic i.e. number of misplaced tiles. A solvability check was implemented at the beginning of every program to avoid unnecessary run time errors and crashing of the program.

Acknowledgment

I would like to thank my honourable **Khan Md. Hasib Sir** for his time, generosity and critical insights into this project.

References

[1] Othar Hansson Andrew E. Mayer Mordechai M. Yung “Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models” Information Sciences, 19920915. Tool,” http://cmsmatrix.org/ (accessed Aug. 16, 2010).

[2] peter e. hart,Nils j. nillson, member IEEE, “A formal basis for the heuristic determination of Minimum cost paths”.

[3] Doran, 1. and Michie, D. “Experiments with the Graph-Traverser Algorithm” Proceedings of the Royal Society, 294 (A), 1966, pp. 235-259.

[4] Michael Katz and Carmel Domshlak Faculty of Industrial Engineering Management Technion, Israel “Optimal Additive Composition of Abstractionbased Admissible Heuristics”, Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008).

Assignment No: 02

CSE-0408 Summer 2021

Kamrunnahar Somapti
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
somaptiesuborno@gmail.com

***Abstract*—Breadth First Search (BFS) and other graph traversal techniques are widely used for measuring large unknown graphs, such as online social networks. It has been empirically observed that incomplete BFS is biased toward high degree nodes. In contrast to more studied sampling techniques, such as random walks, the bias of BFS has not been characterized to date**

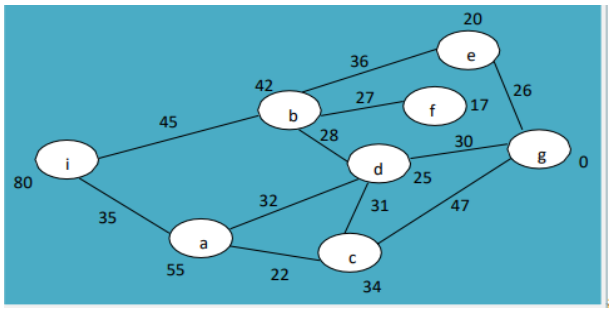
I. Introduction

Breadth-first search (BFS) is an algorithm used for tree traversal on graphs or tree data structures. BFS can be easily implemented using recursion and data structures like dictionaries and list.Breadth First Traversal (or Search) for a graph is similar to Breadth First Traversal of a tree (See method 2 of this post). The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

II. The Algorithm

- Pick any node, visit the adjacent unvisited vertex, mark it as visited, display it, and insert it in a queue.
- If there are no remaining adjacent vertices left, remove the first vertex from the queue.
- Repeat step 1 and step 2 until the queue is empty or the desired node is found.

III. Implementation:



IV. Explanation

```
In [8]: from queue import Queue
        adj_list ={
            "e":["b","g"],
            "b":["i","d","f"],
            "i":["b","a"],
            "d":["b","a","c","g"],
            "f":["b"],
            "a":["i","d","c"],
            "c":["g","d","a"],
            "g":["e","d","c"],
        }
```

```
In [9]: visited= {}
        level = {}
        parents = {}
        bfs_traversal_output = []
        queue = Queue()

        for node in adj_list.keys():
            visited[node] = False
            parents[node] = None
            level[node] = -1

        source_node = "b"
        visited[source_node] = True
        level[source_node] = 0
        queue.put(source_node)

        while not queue.empty():
            u = queue.get()
            bfs_traversal_output.append(u)

            for v in adj_list[u]:
                if not visited[v]:
                    visited[v] = True
                    parents[v] = u
                    level[v] = level[u]+1
                    queue.put(v)
        print(bfs_traversal_output)
```

- The illustrated graph is represented using an adjacency list. An easy way to do this in Python is to use a dictionary data structure, where each vertex has a stored list of its adjacent nodes.
- visited is a list that is used to keep track of visited nodes.
- queue is a list that is used to keep track of nodes currently in the queue.
- The arguments of the bfs function are the visited list, the graph in the form of a dictionary, and the starting node b.
- bfs follows the algorithm described above:

1.It checks and appends the starting node to the visited list and the queue.

2.Then, while the queue contains elements, it keeps

taking out nodes from the queue, appends the neighbors of that node to the queue if they are unvisited, and marks them as visited.

3.This continues until the queue is empty.

V. Time Complexity

Since all of the nodes and vertices are visited, the time complexity for BFS on a graph is O(V + E)O(V+E); where VV is the number of vertices and EE is the number of edges.

VI. Conclusion

When crawling large, undirected networks, BFS tends to discover high-degree nodes first. As a result, the nodes sampled by an incomplete BFS are biased with respect to their degrees, and also to many other properties that correlate with the degree (see [5] for examples). To the best of our knowledge, this is the first paper to quantify this bias. In particular, we calculated the node degree distribution expected to be observed by BFS as a function of the fraction f of covered nodes, in a random graph RG(pk) with a given degree distribution pk. Furthermore, we also showed that, for RG(pk), all commonly used graph traversal techniques that sample nodes without replacement lead to the same bias, and we showed how to correct for this bias.

Acknowledgment

I would like to thank my honourable **Khan Md. Hasib Sir** for his time, generosity and critical insights into this project.

References

[1] J. Leskovec, J. Kleinberg, and C. Faloutsos, “Graphs over time: densification laws, shrinking diameters and possible explanations,” in KDD, 2005 Tool,” http://cmsmatrix.org/ (accessed Aug. 16, 2010).

[2] M. Newman, “Assortative mixing in networks,” in Phys. Rev. Lett. 89, 2002.

[3] R. Motwani and P. Raghavan, Randomized Algorithms. Cambridge University Press, 1990

[4] M. Newman, “Assortative mixing in networks,” in Phys. Rev. Lett. 89, 2002.