Algorithms: Shortest Path Problem

Project Report

[1]Agha Syed Nasir Mahmood Azeemi, [1]Syed Hammad Ali
[1]Syed Ammar Alam, [1]Muhammad Salman Abid
[1]Habib University, Computer Science Block 15 Gulistan e Jauhar, Karachi, Pakistan, 75290

December 13, 2020

1 Introduction to Problem

The earliest examples of the **Shortest Path Problem** can be dated back to the Roman Empire, when it was at its peak. Under ambitious leaders like Julius Caesar, Rome set out to map the 'world' and lay their hands on it using a vast network of roads/routes across Europe.

Around 250,000 miles of roads were laid, some which have survived to this date. The remains suggest that routing of these ancient engineering feats was done so optimally that an element of the shortest path problem being solved is found. However, it is not clear how exactly did the Romans solved the problem.

Today many solutions to the problem exist which have helped in many different domains; from the ancient heritage of building road/railway networks to many different route optimization challenges. A modern use of finding the Shortest Path can be seen on the ubiquitous Google Maps and in state-of-the-art route planning algorithms for Self-driving cars.

From a CS perspective, the problem can be modelled as a graph and algorithms can be devised to traverse the graphs such that we may find a solution to the Shortest Path Problem.

2 Algorithms

2.1 Best-First Search Algorithm

2.1.1 Theory

The Best First Search algorithm is based on the Greedy approach combined with the usage of heuristics. A heuristic is a function that helps us estimate the distance to our goal using a domain-specific formula. E.g., in the case of maps and route planning, the Euclidean distance between a location and the goal location can serve as a useful heuristic. However, the approach does not work for negative distances between two points and furthermore does NOT guarantee the shortest path. When it works though, it is the fastest algorithm because, unlike Breadth First Search, it explores only those nodes that are closer to the goal. This "common sense" is provided by the use of heuristics.

2.1.2 Pseudo-code

```
Best-first search {

closed list = [ ]
open list = [start node]

    do {
        if open list is empty then{
            return no solution
        }
        n = heuristic best node
        if n == final node then {
            return path from start to goal node
        }
        foreach direct available node do{
            if node not in open and not in closed list do {
                add node to open list
                set n as his parent node
        }
        delete n from open list
        add n to closed list
    } while (open list is not empty)
}
```

Figure 1: pseudo-code for BFS

The algorithm starts off by calculating the heuristics to the goal node, for every other node. It then simply keeps track of the nodes it has visited in a closed list and the nodes to the border so to speak, in an open list. The search will expand on wards from the node in the open list that has the smallest heuristic value i.e., appears closest to our goal and hence we visit that node and add the child nodes to our open list. Thus, the border keeps expanding towards the goal node. In an ideal scenario, the complexity of this algorithm is $O(|V| * \log(|V|))$ where $|V|$ is the number of vertices. The complexity is such because the algorithm visits all nodes in the graph and for every node, it picks the smallest value from a sorted list which is often a min/max heap.

2.2 Dijkstra Algorithm

2.2.1 Theory

Dijkstra's algorithm is another algorithm based on the greedy approach as it follows the problem-solving heuristics of making the most locally optimal choice at any given stage. The original algorithm, published in 1956, would find the shortest path between any 2 given nodes in a graph. Today the algorithm has been modified and many variants exist where the common approach is to fix a single node as the source node and find the shortest path to all other nodes. This approach helps produce a Shortest Path Tree (SPT). The algorithm works on both weighted and unweighted graphs but fails if the graph has negative cycles. The algorithm also produces unreliable results if the graph has negatively weighted edges.

2.2.2 Pseudo-code

```
1:  function Dijkstra(Graph, source):
2:      for each vertex v in Graph:                 // Initialization
3:          dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
4:          previous[v] := undefined                // Previous node in optimal path from source
5:      dist[source] := 0                           // Distance from source to source
6:      Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
7:      while Q is not empty:                       // main loop
8:          u := node in Q with smallest dist[ ]
9:          remove u from Q
10:         for each neighbor v of u:               // where v has not yet been removed from Q.
11:             alt := dist[u] + dist_between(u, v)
12:             if alt < dist[v]                    // Relax (u,v)
13:                 dist[v] := alt
14:                 previous[v] := u
15:     return previous[ ]
```

Figure 2: pseudo-code for Dijkstra

The algorithm works by assigning each node a large value (e.g., Infinity). The source node is assigned 0. The node with the smallest value is selected (source node in the first iteration) and its neighbors are visited. A new value is calculated for each neighbor being visited by adding the value of the selected node to the edge value between the selected node and the neighbor being visited. The neighbor is assigned the new value if the new value is smaller than it's current assigned value. The process is repeated for all vertices. The algorithm can be implemented with different underlying data structures to store graphs and hence, can have different time complexities. E.g., the time complexity for running Dijkstra on a graph stored via Adjacency Lists is $O(|E| * Log(|V|))$ where $|E|$ is the number of edges and $|V|$ is the number of vertices in the graph. If the graph is stored as an Adjacency Matrix, as in our empirical analysis, the complexity is $O(|V|^2)$. It is as such because iteration over the vertices to find the vertex with the smallest value is $O(|V|)$. Then

for each selected vertex to find its neighbors is $O(|V|)$. To calculate the new value is just $O(1)$, Hence, the complexity is $O(|V|^2)$ where $|V|$ is the number of vertices.

2.3 Bellman-Ford Algorithm

2.3.1 Theory

Bellman Ford is an algorithm to find shortest paths from a source vertex to all the other vertices in a graph. It works in a similar way to the Dijkstra algorithm, however, it is a non greedy algorithm and hence has a complexity of $O(|V| * |E|)$, where $|V|$ and $|E|$ are the number of vertices and edges in the graph respectively. It runs a total of $|V| - 1$ iterations where it goes over all of the edges in the graph and tries to find the shortest distance from the source node to all the other nodes. Unlike the Dijkstra algorithm, it can work on a graph with negative weights on its edges and also has the ability to catch any negative cycles.

2.3.2 Pseudo-code

```
procedure shortest-paths(G, l, s)
Input:    Directed graph G = (V, E);
          edge lengths {l_e : e ∈ E} with no negative cycles;
          vertex s ∈ V
Output:   For all vertices u reachable from s, dist(u) is set
          to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil

dist(s) = 0
repeat |V| − 1 times:
   for all e ∈ E:
      update(e)


procedure update((u, v) ∈ E)
dist(v) = min{dist(v), dist(u) + l(u, v)}
```

Figure 3: pseudo-code for Bellman-Ford

The pseudo-code takes in a graph and a source node as input, and returns the the minimum distance to all the edges from the source node. Then it initializes a dictionary of size $|V|$ with all the vertices as keys. The values of the dictionary, i.e. distance from the source node are set to ∞ except for the source node, whose distance from itself is set to be 0. Then there are $|V| - 1$ iterations of a loop, in which the the algorithm goes to each vertex of the graph and checks whether it can relax any other node in the graph, that is, is there any edge from the current vertex u to any

adjacent vertex v, so that the already registered total distance of v from the source node, dist[v], can be reduced to a smaller distance of dist[u] + weight of edge —u to v—. If there is any, the value of dist[v] is updated. Lastly, we can add one more similar iteration to catch any negative cycle in the graph, if any of the distance values gets updated, it means that there is a negative cycle in the graph.

2.4 A* Algorithm

2.4.1 Theory

The A* algorithm is an algorithm which guarantees finding the shortest path, something that is not always true for other algorithms of this category. It is not a greedy algorithm as it is not short sighted in its decision making. It uses information about how much traversal it has incurred, and how much traversal it will further perform in the graph to reach the goal state. It uses that information to estimate the best possible node that it can choose. The evaluation function considers both incurred cost as well as the estimated cost to reach the goal node and while choosing which node to expand upon. The algorithm guarantees the shortest path by using heuristics just like Best-First Search. An interesting point is the use of admissible heuristics, because if the heuristics function is able to avoid overestimating the distance to the goal node, we will always find the shortest path. Perhaps not in the optimum amount of time, because if it underestimates the distance, the algorithm will run slower than if it was an exact estimation. If the heuristics also satisfies the heuristics constraint, it will cost lesser than the Dijkstra algorithm and simultaneously provide a guarantee of finding the shortest path.

2.4.2 Pseudo-code

```
1  Put node_start in the OPEN list with f(node_start) = h(node_start) (initialization)
2  while the OPEN list is not empty {
3    Take from the open list the node node_current with the lowest
4         f(node_current) = g(node_current) + h(node_current)
5    if node_current is node_goal we have found the solution; break
6    Generate each state node_successor that come after node_current
7    for each node_successor of node_current {
8      Set successor_current_cost = g(node_current) + w(node_current, node_successor)
9      if node_successor is in the OPEN list {
10       if g(node_successor) ≤ successor_current_cost continue (to line 20)
11     } else if node_successor is in the CLOSED list {
12       if g(node_successor) ≤ successor_current_cost continue (to line 20)
13       Move node_successor from the CLOSED list to the OPEN list
14     } else {
15       Add node_successor to the OPEN list
16       Set h(node_successor) to be the heuristic distance to node_goal
17    }
18     Set g(node_successor) = successor_current_cost
19     Set the parent of node_successor to node_current
20   }
21   Add node_current to the CLOSED list
22 }
23 if(node_current != node_goal) exit with error (the OPEN list is empty)
```

Figure 4: pseudo-code for A* Search

The pseudo-code given for A* illustrates a similar operation to the Best First search, but also incorporates the traversal cost in evaluating the potential path to take. In comparison with best first search with A*, the Best-First search uses only a heuristics function $F(n) = h(n)$ where the lowest heuristics value of a neighbor will be used to choose the next node. It is a greedy approach, and it does not take into consider the cost of expanding the current path itself. The A* search algorithm chooses the next path based on the formula $F(n) = h(n) + g(n)$, where $h(n)$ is the heuristics function and $g(n)$ is the cost already incurred on the path from the initial state. Therefore, it does not simply choose the best heuristics value, but instead uses a combination of both the $g(n)$ and $h(n)$ for evaluating the best path. The complexity of the algorithm is $O(E)$ where the worst case is the algorithm traversing over all edges of the graph.

3 Empirical Analysis & Results

The empirical analysis was performed by measuring the amount of time required for each algorithm to run on identical graphs with different number of nodes. The **Networkx** library in python was used to produce multiple un-directed unweighted graphs with different number of nodes (e.g., 10, 50, 100, 500, 1000, 5000, 10000 etc.). The data set for each algorithm consisted of time required to run for different number of nodes. The data set was plotted and curve fitted to produce graphs for comparison.

3.1 Best-First Search Algorithm

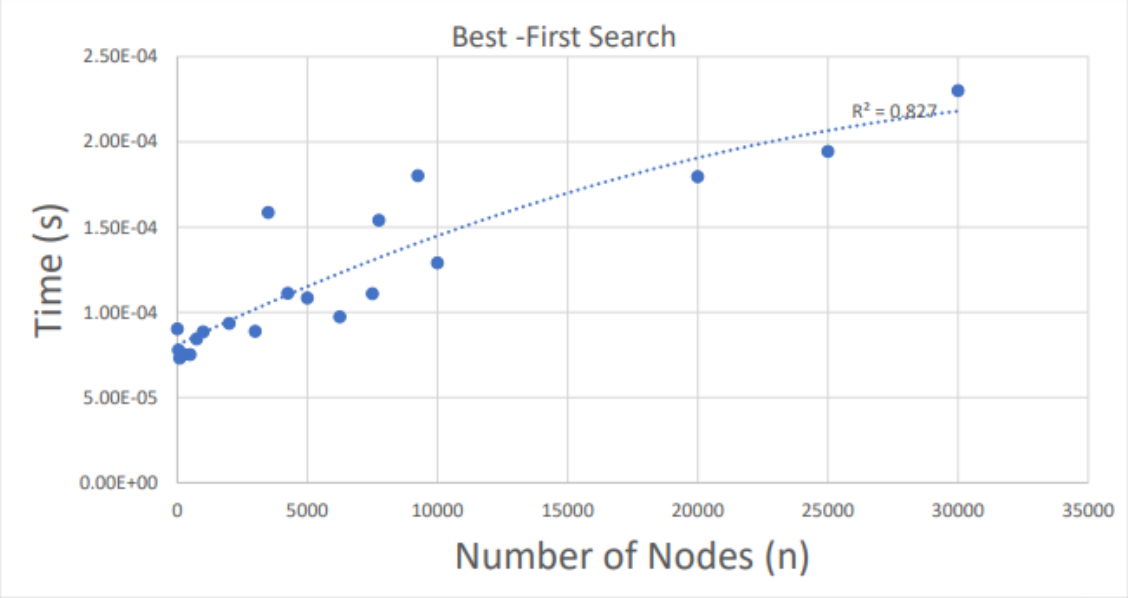

Figure 5: Time VS No. of Nodes plotted for BFS

The graph is close to a logarithmic graph because when BFS has good heuristics it tends closer to the best case complexity which is the depth of the graph ($\Omega(d)$). This makes BFS very fast in certain scenarios. The results can been seen scattered in the start but tend to get closer to the best-fit curve.

3.2 Dijkstra's Algorithm

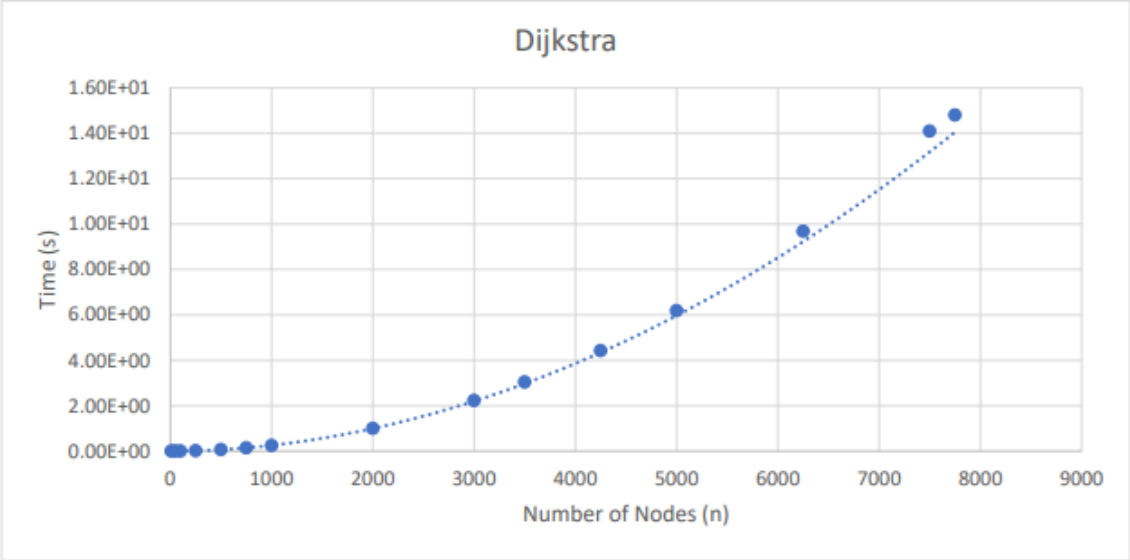

Figure 6: Time VS No. of Nodes plotted for Dijkstra

Dijkstra's Algorithm produces different results when different data structures are used. In our analysis we ran Dijkstra's algorithm on graphs that were represented by adjacency matrices. The result is a quadratic curve, which verifies the theoretical complexity of the algorithm running on adjacency matrices, which is $O(|V|^2)$.

3.3 Bellman-Ford's Algorithm

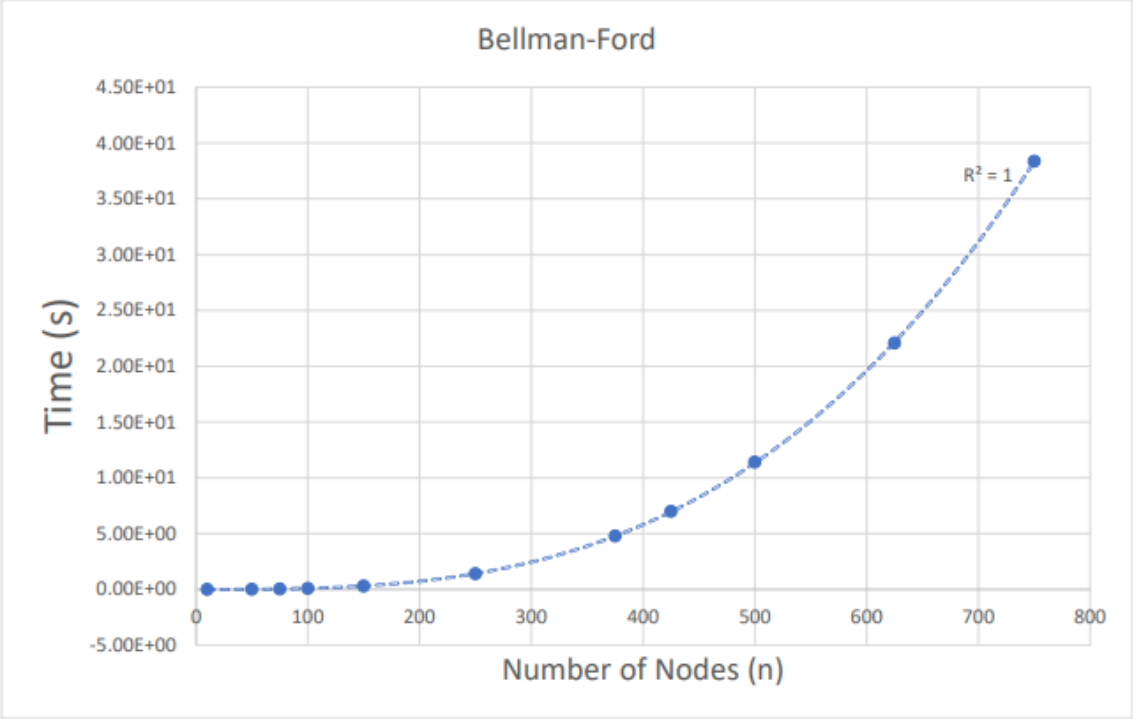

Figure 7: Time VS No. of Nodes plotted for Bellman-Ford

The result for Bellman-Ford's Algorithm is a quadratic curve. The algorithm runs for $|V| - 1$ iteration as it goes over each node. In each iteration the the algorithm visits the edges of the respective node. Hence, the theoretical complexity become$O(|V| + |E|)$. We can approximate the $|E|$ to $|V|$ which results in $O(|V|^2)$. The theoretical complexity matches our results.

3.4 A* Search Algorithm

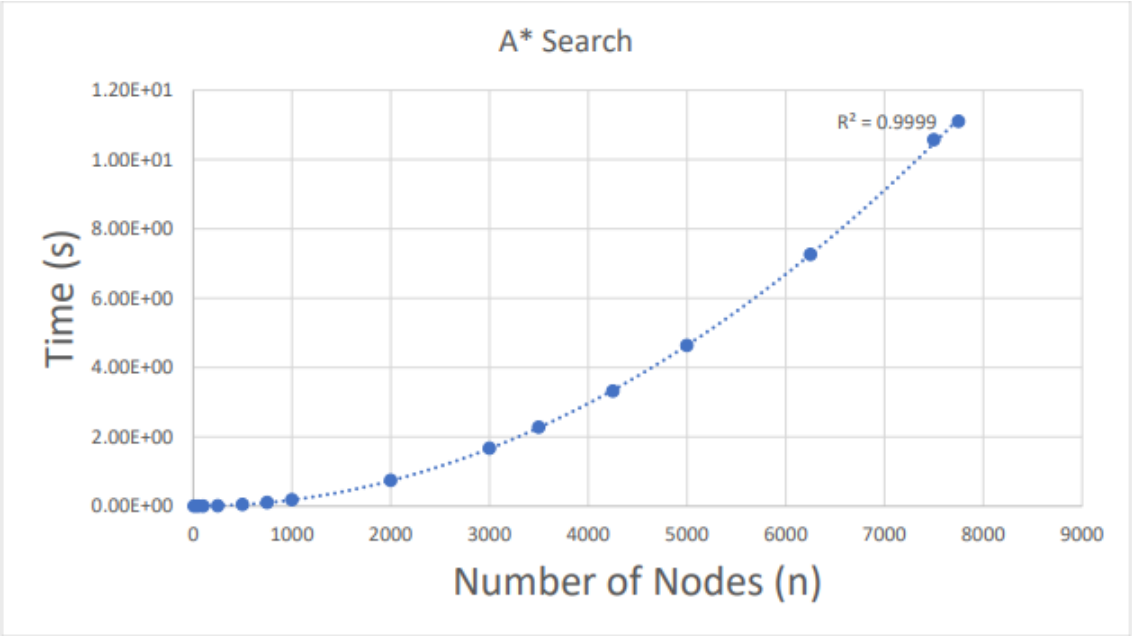

Figure 8: Time VS No. of Nodes plotted for A* Search

The result for A* Search Algorithm is a quadratic curve. A* search is highly depends on heuristics. If the heuristics for A* are bad then the algorithm acts similar to Dijkstra's algorithm. Hence, we the results for both are similar. Given good heuristics the A* search algorithm can work in constant time complexity.

3.5 Conclusion

BFS was the fastest because it is a greedy search technique in an ideal scenario the greedy method will usually be the fastest method, followed by the A* which combines the best of BFS and Dijkstra to do a non-exhaustive and smart search for the goal node while Dijkstra has to look for a lot more nodes around the goal node before it considers the shortest distance to goal as a solution, as it is an extension of the BFS. Bellman Ford does an exhaustive search and is hence the slowest out of the four.

4 Project Video Link

5 Q/A session

Q:How did you guys perform the empirical analysis?

Each Algorithm was run and timed on a set of graphs with varying amounts of vertices on the same machine and language (Python). The graphs were produced via the **Networkx** library function **fast_gnp_random_graph()**. The time taken by each algorithm was plotted as a function of number of nodes in the graph and curve-fitted.

Q:Did you guys find anything interesting in your results?

We were surprised by the speed of Best First search algorithm. It was the only algorithm that we were able to run for a graph with 30,000 nodes. Bellman-Ford was only run for a maximum of 750 nodes. The results for Dijkstra and A* search were almost identical which suggested bad heuristics in A* search. This is probably due to the fact that the example network had no walls and the Best-First Search in that case is the most optimal algorithm to find the shortest path. In this case, the time complexity of Best-First is its best case complexity which is $\Omega(d)$ where d is the depth of the goal node.

6 GitHub Repository Link

7 Resources Used

1. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms Ittai Abraham, Amos Fiat, Andrew V. Goldberg, Renato F. Werneck
2. `https://drivethruhistoryadventures.com/roads-roman-empire/`
3. Zaharija, Goran & Mladenovic, Sasa & Dunić, Stefan. (2017). Cognitive Agents and Learning Problems. International Journal of Intelligent Systems and Applications. 9. 1-7. 10.5815/ijisa.2017.03.01.
4. `https://www.annytab.com/a-star-search-algorithm-in-python/`
5. `https://www.geeksforgeeks.org/python-program-for-dijkstras-shortest-path-algorithm-greedy-algo-7/`
6. `https://www.geeksforgeeks.org/best-first-search-informed-search/`
7. `https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/`