Project Report

COMP 6651 Algorithm Design Techniques

Tejasvi Tejasvi (40292854)

Daivik Daivik (40279335)

Pruthviraj Horadi (40266887)

Table of Contents

Problem Description & its Purpose

This project's main goal was to complexities the intricacies of graph theory and algorithm creation. In particular, this project concentrated on the NP-hard issue of determining the longest simple route in any given undirected graph.

Our team developed a random undirected, unweighted geometric graph generator by implementing a random Euclidean graph generator as per the professor's heuristic which creates a graph in 2 formats. The behavior of several heuristic Longest Simple Path (LSP) algorithms was examined using these graphs.

Additionally, we chose and examined graphs from an internet database as provided in the project sheet. We determined the biggest connected component (LCC) for every graph. As a result, we gained a greater comprehension of the graphs' characteristics and structure.

The LCC of each network was then subjected to four heuristic LSP techniques by our team. These methods included a heuristic of our own invention, a modified version of Dijkstra's algorithm, a modified version of the A* algorithm to find LSP, and a depth-first search (DFS) heuristic. By using these methods, we were able to investigate several approaches to solving the LSP issue.

We used the professor's heuristics as a guide, but we discovered that the code needed some changes in order to operate. These adjustments were thoughtfully planned and put into practice to guarantee the performance of our algorithms.

Our simulations' outcomes were noted and examined. The number of nodes in the graph, the number of nodes in the LCC, the highest degree of any node in the LCC, and the length of the LSP discovered by the heuristic in the LCC were among the main metrics that we examined.

There was more to this project than merely solving problems. It was about understanding the characteristics of NP-hard issues and investigating approaches to solving them. We learned a great deal about the difficulties involved in algorithm design and computing complexity by putting these algorithms into practice and conducting experiments with them and finding the importance of approximation algorithms.

Implementation Details

"main runner" function

First of all, we will mention the "main" function for every algorithm file we have presented in the project.

We have used two types of main function in this project for our algorithms –

1. "main" function for algorithms using only the EDGES data
   In this main function we primarily just read the EDGES file of the format in which on each edge of line only 2 numbers are present which represent a undirected edge between 2 vertices. Below is the pseudocode of main function of DijkstraMax algorithm -

```
1. function mainRunner(filename):
2.     Open a file named ("filename")
3.     Initialize maxVertex to 0          // maxVertex here keeps the count of total vertices in graph
4.
5.     For each line in the file:
6.         Split the line into two parts, v and w
7.         Update maxVertex to be the maximum of maxVertex, v and w
8.
9.     Close the file
10.
11.    Initialize a Graph g with size maxVertex + 1
12.
13.    Open the file again
14.
15.    For each line in the file:
16.        Split the line into two parts, v and w
17.        Add an edge in the graph g from v to w
18.        Add an edge in the graph g from w to v
19.
20.    Close the file
21.
22.    For each vertex i from 0 to maxVertex:
23.        Run the dijkstraMax function on the graph g with source vertex i
24.
25.    Print "Longest simple path length: " followed by the value of Graph.maxDistance
26.
27.    Print a newline
28. End function
```

   The main function of DFS Heuristic is similar to this. In this pseudocode we open the graph file and first read the vertices to get the idea of total vertices in the graph, then using that information we initialise a graph of that vertex size.

   After that we again read the file but this time we add edges and connections to the graph and store that in adjacency list for the traversal of the algorithm.

2. For the 2nd kind of main function which is quite similar to the above one but only differs in the way that it reads the positional graph and put the "x" and "y" coordinates of the vertices in the "Node" object (between line 17 to 19 of code above) created using the Node Class below

```
1. class Node {
2.     int id;
3.     double x, y;
4.     int depth;
5.     Node parent;
6.     boolean visited;
7.     int color; // 0: WHITE, 1: GRAY, 2: BLACK
8.     int degree;
9.
10.     Node(int id, double x, double y) {
11.         this.id = id;
12.         this.x = x;
13.         this.y = y;
14.         this.depth = 0;
15.         this.parent = null;
16.         this.visited = false;
17.         this.color = 0; // Initially WHITE
18.         this.degree = 0;
19.     }
20. }
```

It is a typical Node class which stores details for every Node although the implementation is same across all of the algorithms but only data differs.

DFS Heuristic

For the DFS there are quite a change in our heuristic in contrast to the professor's. See the pseudocode below with explanation

```
1. Define function DFSHeuristic:
2.     Initialize maxLength to 0
3.
4.     If the length of nodes is greater than 0:
5.         For i from 0 to MAX_ATTEMPTS:
6.             Select a random startNode from nodes
7.             Call DFSVisit with startNode and depth 0
8.
9.             Find the node with maximum depth, maxDepthNode
10.             Reset all nodes
11.             Call DFSVisit with maxDepthNode and depth 0
12.
13.             Update maxLength to be the maximum of maxLength and the depth of the node with maximum
depth
14.     Else:
15.         Print "Error: Node list is empty."
16.
17.     Return maxLength
18. End function
```

1st of all rather than looping over for only the number of time the quantity of LCC in line 5 we have used a global variable of around 10000. This ensures more trails of the DFS because in every iteration a Node is selected as random and it doesn't guarantee that each node is picked as a starting one, also

DFS algorithm tend to take different paths every time on same graph due to its nature of being depth first so there's always some uncertainty of the path it crosses. Also in the line 10 after the first DFS call finishes we resets the all the variables such as parent, distance, etc of nodes to a fresh set of variables. It ensures clean run of DFS.

Other than this is implementation of the DFS is same as provided in the Project manual.

Dijkstra Heuristic

There are quite many implementation details which we were needed to add into it so as to run it perfectly.

```
1. Define function dijkstraMax with input s:
2.     Initialize dist, prev, and inQueue arrays of size V
3.     Initialize a priority queue pq with a custom comparator that compares nodes based on their dist values
4.
5.     For each node i from 0 to V:
6.         If i equals s, set dist[i] to 0, else set it to Integer.MIN_VALUE
7.         Set prev[i] to -1 and inQueue[i] to true
8.         Add a new node with id i and distance dist[i] to pq
9.
10.     While pq is not empty:
11.         Poll a node u from pq and set inQueue[u.id] to false
12.         If u.dist is not equal to dist[u.id], continue to the next iteration
13.
14.         For each adjacent node v of u:
15.             If dist[v] is less than dist[u.id] + 1 and v is in the queue:
16.                 Update dist[v] and prev[v], set inQueue[v] to true, and add a new node with id v and distance
dist[v] to pq
17.
18.     Initialize maxDist to 0 and endNode to -1
19.     For each node i from 0 to V:
20.         If dist[i] is greater than maxDist, update maxDist and endNode
21.
22.     If maxDist is greater than maxDistance, update maxDistance and maxEnd
23. End function
```

This function is designed to find the longest path in a graph, which is a more challenging problem. Here are some of its distinguishing features:

- **Priority Queue with Custom Comparator**: This function uses a priority queue to select the node with the maximum distance at each step. The custom comparator we created ensures that the node with the maximum distance is always at the front of the queue.
- **Distance and Previous Arrays**: The dist array keeps track of the maximum distance from the source node to each node, while the prev array keeps track of the previous node (parent node of a node) on the path to each node. These arrays are crucial for reconstructing the longest path at the end of the function.
- **In-Queue Array**: The inQueue array is used to check whether a node is in the queue. This is necessary because in Java's PriorityQueue, checking whether an element is in the queue takes linear time, but with this array, it can be done in constant time and also to prevent cycles.
- **Updating Distances**: When a node is found to have a longer path, its distance is updated, and it is added back to the queue. This is a key part of the algorithm, as it allows the function to continually update the distances as it discovers longer paths.

- **Finding the Longest Path**: After all nodes have been processed, the function iterates over the dist array to find the node with the maximum distance, which is the end of the longest path. It also updates maxDistance and maxEnd to keep track of the longest path found so far.

A* Heuristic

The A* algorithm accommodates the same logics as we have seen in the Dijkstra heuristic to make the algorithm work, below is the pseudocode along with some niche details of the changes to make this algo work.

```
1. Define function aStarMax with inputs s, d, and nodes:
2.     Initialize dist, prev, and inQueue arrays of size V
3.     Initialize a priority queue pq with a custom comparator that compares nodes based on their dist and h values
4.
5.     For each node i from 0 to V:
6.         If nodes[i] is null, continue to the next iteration
7.         Calculate the heuristic value h for nodes[i] using the Euclidean distance formula
8.
9.     For each node i from 0 to V:
10.        If nodes[i] is null, continue to the next iteration
11.        If i equals s, set dist[i] to 0, else set it to Double.NEGATIVE_INFINITY
12.        Set prev[i] to -1 and inQueue[i] to true
13.        Add a new node with id i, coordinates nodes[i].x and nodes[i].y, distance dist[i], and heuristic value nodes[i].h to pq
14.
15.    While pq is not empty:
16.        Poll a node u from pq and set inQueue[u.id] to false
17.        If u.dist is not equal to dist[u.id], continue to the next iteration
18.
19.        For each adjacent node v of u:
20.            If dist[v.id] is less than dist[u.id] + 1, v is in the queue, and prev[u.id] is not v.id:
21.                Update dist[v.id] and prev[v.id], set inQueue[v.id] to true, and add a new node with id v, coordinates nodes[v.id].x and nodes[v.id].y, distance dist[v.id], and heuristic value dist[v.id] + nodes[v.id].h to pq
22.
23.    Initialize maxDist to 0 and endNode to -1
24.    For each node i from 0 to V:
25.        If dist[i] is greater than maxDist, update maxDist and endNode
26.
27.    Print "Longest path length: " followed by maxDist
28.    Print "Path: "
29.    For each node v from endNode to -1, following the path in prev:
30.        Print v followed by a space
31.    Print a newline
32. End function
```

Now this function is too designed to find the longest path in a graph, which is a more challenging problem. Here are some of its distinguishing features:

- **Priority Queue with Custom Comparator**: This function uses a priority queue to select the node with the maximum distance plus heuristic value at each step. The custom comparator ensures that the node with the maximum distance plus heuristic value is always at the front of the queue.

- **Heuristic Calculation**: The function calculates a heuristic value for each node based on the Euclidean distance to the destination node. This heuristic value is used to guide the search towards the destination.
- **Distance and Previous Arrays**: The dist array keeps track of the maximum distance from the source node to each node, while the prev array keeps track of the previous node on the path to each node. These arrays are crucial for reconstructing the longest path at the end of the function.
- **In-Queue Array**: Again we have used an inQueue array is used to check whether a node is in the queue because in Java's PriorityQueue, checking whether an element is in the queue takes linear time, but with this array, it can be done in constant time.
- **Updating Distances**: When a node is found to have a longer path, its distance is updated, and it is added back to the queue. This is a key part of the algorithm, as it allows the function to continually update the distances as it discovers longer paths.
- **Finding the Longest Path**: After all nodes have been processed, the function iterates over the dist array to find the node with the maximum distance, which is the end of the longest path.

Iterative DFS

We changed the DFS based heuristic to iterative one as our 4th heuristic as it provides some performance and software benefits from the recursion one as we will see in a bit, before that its pseudocode is below –

```
1. Define function DFSVisit with inputs startNode, time, and visited:
2.     Initialize a stack and push startNode into it
3.     Mark startNode as visited and color it as GRAY
4.
5.     While the stack is not empty:
6.         Peek the top node from the stack
7.         For each edge in edges:
8.             If the edge contains the node:
9.                 Get the other node of the edge, neighbor
10.                 If neighbor is not visited:
11.                     Set neighbor's parent to node
12.                     Mark neighbor as visited and color it as GRAY
13.                     Increment time and set neighbor's depth to time
14.                     Push neighbor into the stack
15.
16.         Color the node as BLACK and pop it from the stack
17.
18.     Return startNode
19. End function
20.
```

The DFSVisit function demonstrates that the iterative form of Depth-First Search (DFS) using a stack has many distinct advantages over the recursive version that might be more advantageous in certain situations:

Memory Efficiency: Because recursive DFS uses the system stack, it may result in a stack overflow for sufficiently large graphs. Nevertheless, the iterative version uses a user-defined stack, which is

often more memory-efficient and able to handle larger graphs.

route tracking: The iterative DFS in the DFSVisit function keeps track of each node's parent in order to reconstruct the route from the start node to any other node. Recursive DFS may also do this, although more data structures or parameters are often required.

Control Over Traversal: In the iterative version, you have more control over the traversal process. For example, it is easy to implement policies that forbid revisiting nodes or, in certain cases, terminate the traverse early.

Thread Safety: Recursive functions may pose problems in multi-threaded environments since they depend on shared system resources. Iterative functions are thus often safer.

Color Scheme: When a node is first visited, it is colored GRAY; however, after all of its neighbors have been visited, it becomes BLACK. The function makes use of this color palette. It's common practice to track the DFS's evolution using this color scheme.

Time and Depth: The time variable is raised each time a new node is accessed. This time is then used to set the node's depth, which shows when the node was visited throughout the DFS.

In conclusion, even though both techniques may handle the same issues, choosing between recursive and iterative DFS usually comes down to the specific requirements of the problem as well as the characteristics of the input data.

Implementation Correctness

The correctness of our implementation of the algorithms were verified using different techniques which is listed below.

Result Correctness

First we used some small graphs for testing the correct solution by hand drawing them on paper and every time we got the correct result. Relaxing edges, calculating the values of the “dist arrays” for the graphs. Taking the benchmark as the Results of the Dijkstra’s Heuristic we concluded that the result for other algorithms as for example in case if we have bigger graphs the Results of other algorithm gets deviated from the original results.

To also let the marker check manually we have also added a 15 vertex graph file and its result in the project.

Software Correctness

To check the software correctness we used techniques like giving the starting node to the algorithm which is not present in the graph and in case of A* also giving it some endNode not available will throw the error, so it is boundary tested, also if we removed the inQueue array check then the algorithm went to the infinite loop so breaking the code.

Also initially before implementing the Longest Path Heuristic, we first implemented the smallest path algorithms to make sure that the code setup works fine and then converted that algorithm to the Longest Path one and this made sure that the code setup works fine.

Results

The results for different graph of different sizes are given below in tables. There will be 2 different tables for the algorithms using normal EDGES format graph and graphs which need the positional information.

Algorithm	Graph File	n	r	$\lvert V_{LCC} \rvert$	$\Delta(LCC)$	$\overline{k(LCC)}$	L_{max}
DFS Heuristic							
	smallgraph	15	0.4	6	4	2	5
	graph300r28	300	0.28	279	12	5	169
	graph400r26	400	0.26	341	14	5	140
	graph500r24	500	0.24	365	13	5	170
	DSJC500-5	500	-	500	286	250	499
	Inf-euroroad	1200	-	1039	10	2	302
	Inf-power	5000	-	4941	19	2	1014
Dijkstra Heuristic							
	smallgraph	15	0.4	6	4	2	5
	graph300r28	300	0.28	279	12	5	181
	graph400r26	400	0.26	341	14	5	149
	graph500r24	500	0.24	365	13	5	174
	DSJC500-5	500	-	500	286	250	499
	Inf-euroroad	1200	-	1039	10	2	331
	Inf-power	5000	-	4941	19	2	1038

Table for non-positional Data driven graphs

The result for graph file “DSJC500-5” using DFS will take around 15 minutes because of having a large average node degree so the DFS have to check every possible path.

Algorithm	Graph File	n	r	$\lvert V_{LCC} \rvert$	$\Delta(LCC)$	$\overline{k(LCC)}$	L_{max}
A* Heuristic							
	smallGraphPositional	15	0.4	6	4	2	5
	graphPositional300r28	300	0.28	279	12	5	232
	graphPositional400r26	400	0.26	341	14	5	221
	graphPositional500r24	500	0.24	365	13	5	261
	DSJC500-5 Experimental	500	-	500	316	196	499
	Inf-euroroad Experimental	1200	-	1013	16	5	512
	Inf-power Experimental	5000	-	4992	19	7	4274 From vertex 0 to maxVertex
Our Iterative DFS Based Heuristic							
	smallgraph	15	0.4	6	4	2	5
	graph300r28	300	0.28	279	12	5	86
	graph400r26	400	0.26	341	14	5	113
	graph500r24	500	0.24	365	13	5	99
	DSJC500-5	500	-	500	286	250	499
	Inf-euroroad	1200	-	1039	10	2	234
	Inf-power	5000	-	4941	19	2	1173

Also A* was taking around 3 hours plus to get results on Inf-Power experimental graph, so we manually chose the start and end vertex to get that answer in the code file as in code the distance is checked between each and every vertex to get the max distance

Conclusions

This project report presents the efforts of our team to explore the graph longest and smallest path finding and calculating algorithms, specifically working on the NP-Hard problem of the Longest Path Algorithm.

The major finding of this project was that as the size of the Graph grows the answers become more uncertain and deviated from the real answers but these will give the correct results on the smaller graphs so these algorithms are merely kind of approximation algorithms as this problem is NP-Hard.

Our team used the professor's heuristics as a guide, but found that the code needed some modifications to function. These adjustments were carefully planned and implemented to ensure the performance of their algorithms. The outcomes of their simulations were recorded and analyzed.

Team Work Distribution

This project was definitely a team effort, but we did distribute certain tasks among us for example Tejasvi (40292854) wrote and tested the GraphGenerator, LargestConnectedComponent, Dijsktra. Daivik(40279335) implemented the DFSHeuristic and also provided changes to LCC, Pruthviraj(40266887) implemented A* as it was a little bit complicated. As for the 4th Algorithm was mainly devised and implemented by each team member's efforts and important inputs.

Project Report was a team effort as results and pseudocodes was populated by each member after calculating them on their respective machines.

References

- Textbook: Introduction to Algorithms, Fourth Edition, By Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/