



Project 5: Sheldon Air

Repository at:

<https://github.com/fabriziorocco/Project-Group-Luiss.git>

Michele Lizzit | Fabrizio Rocco | Andrea Vitto

The Problem

SheldonAir is an emerging airline in the airline business. Dr. Cooper, the CEO of SheldonAir, would like to know the minimum number of legs (flight segments) required to go from one city to another city.

This problem can be represented by an undirected graph where:

- vertices correspond to cities where SheldonAir is operating
- an edge between two vertices corresponds to a direct SheldonAir flight between the two corresponding cities.

In other terms, **given two vertices u and v , you are required to compute the length of a shortest path between u and v** (i.e., the minimum number of edges in a path from u to v).

Idea

Goal: Find the shortest path between vertices of a graph.

Assumption

We have to initialize all the nodes with a placeholder value, let's suppose ∞ , and we have to attribute to each edge a value.

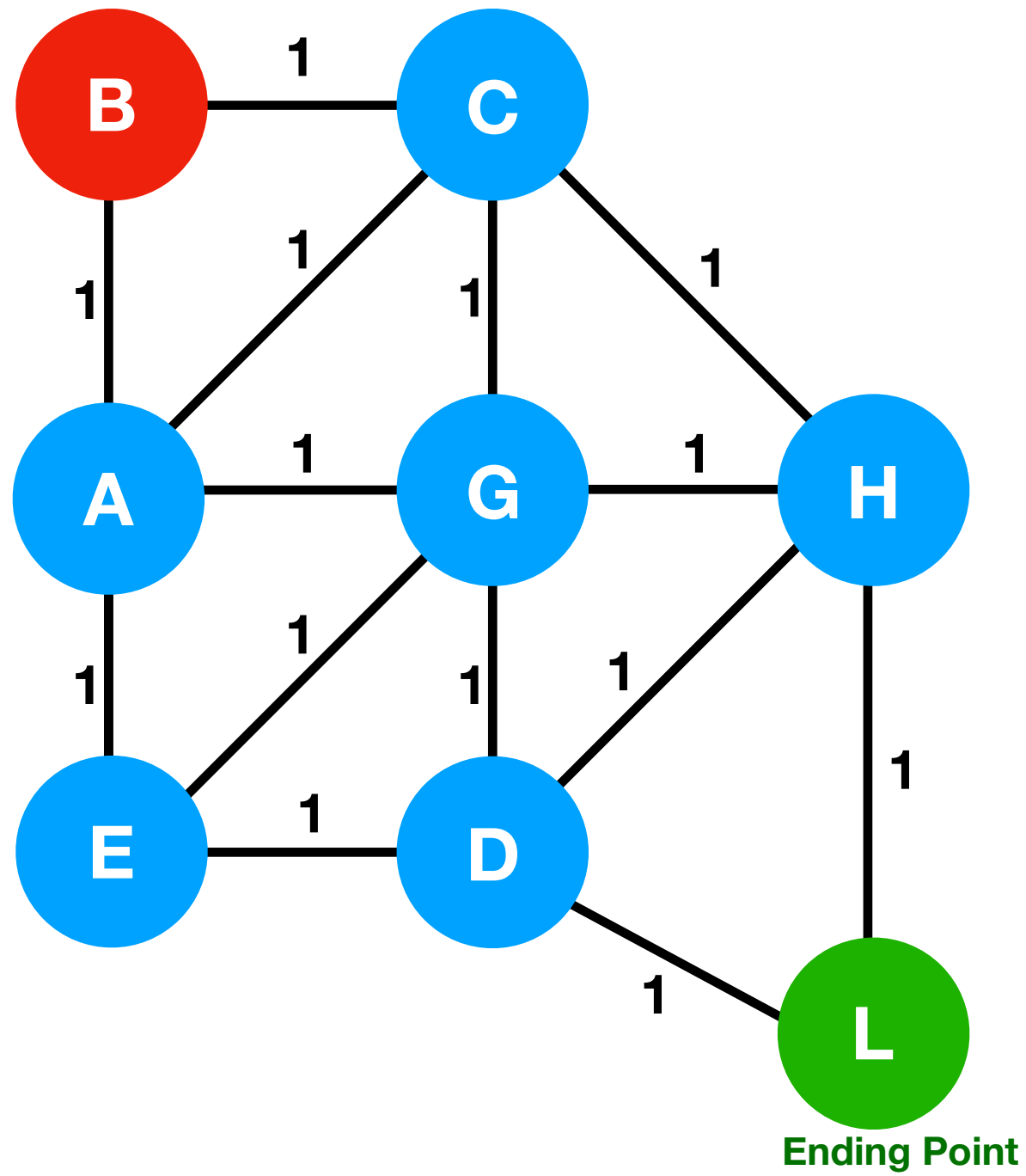
There will be a set S containing the nodes already visited, $S = \{ \}$ and a set T , containing the nodes to analyze yet. $T = \{ \}$.

At the beginning of the execution the two sets are so defined:

- $S = \{a\}$ # Only the starting vertex a
- $T = \{ \text{All the other nodes} \}$

We give to each edge the **weight 1**.

Starting Point



Intuition: Dijkstra Algorithm

- This algorithm is also known as “**Dijkstra Algorithm**”
“Dijkstra's algorithm (or Dijkstra's Shortest Path First algorithm, SPF algorithm) is an algorithm for finding the shortest paths between nodes in a graph. It was conceived by computer scientist **Edsger W. Dijkstra** in 1956 and published three years later.”

Source: Wikipedia

Key Points for “constant weight” Dijkstra Algorithm

- Every edge has weight 1 and we’re just interested in the shortest path of **segments**.
- Once we’ve moved to the node we’re going to visit, we will check each of its neighboring nodes.
- For each neighboring node, we’ll calculate the distance/cost for the neighboring nodes by summing the cost of the edges that lead to the node we’re checking *from the starting vertex*.
- Finally, if the distance/cost to a node is *less than* a known distance, we’ll update the shortest distance that we have on file for that vertex.

Running Time of Dijkstra Algorithm

Complexity can expressed by $|V|$ and $|E|$, which are respectively the number of vertices and the number of edges.

	m Decrease Key	n DeleteMin	n Insert	DIJKSTRA
ARRAYS	$\Theta(1)$	$O(V)$	$\Theta(V)$	$O(V ^2 + E)$
BINARY HEAP	$O(\log_2 V)$	$O(\log_2 V)$	$\Theta(V)$	$O((V + E) \log_2 V)$
FIBONACCI HEAP	$\Theta(1)$	$O(\log_2 V)$	$\Theta(V)$	$O(V \log_2 V + E)$
QUEUE	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$O(V + E)$

Since each edge has weight 1 (constant), our running time is $O(n)$. (linear)
To do this we use a queue.

Queue works correctly only if the edges have constant weight.

Implementation Choice

- We decided to implement this algorithm using **C++** in order to perform a benchmark as accurate as possible.
- For the test environment we used a **Ryzen 2700X**.
- Task pinned to a physical core and hyper threading disabled.
- Our implementation is single threaded.

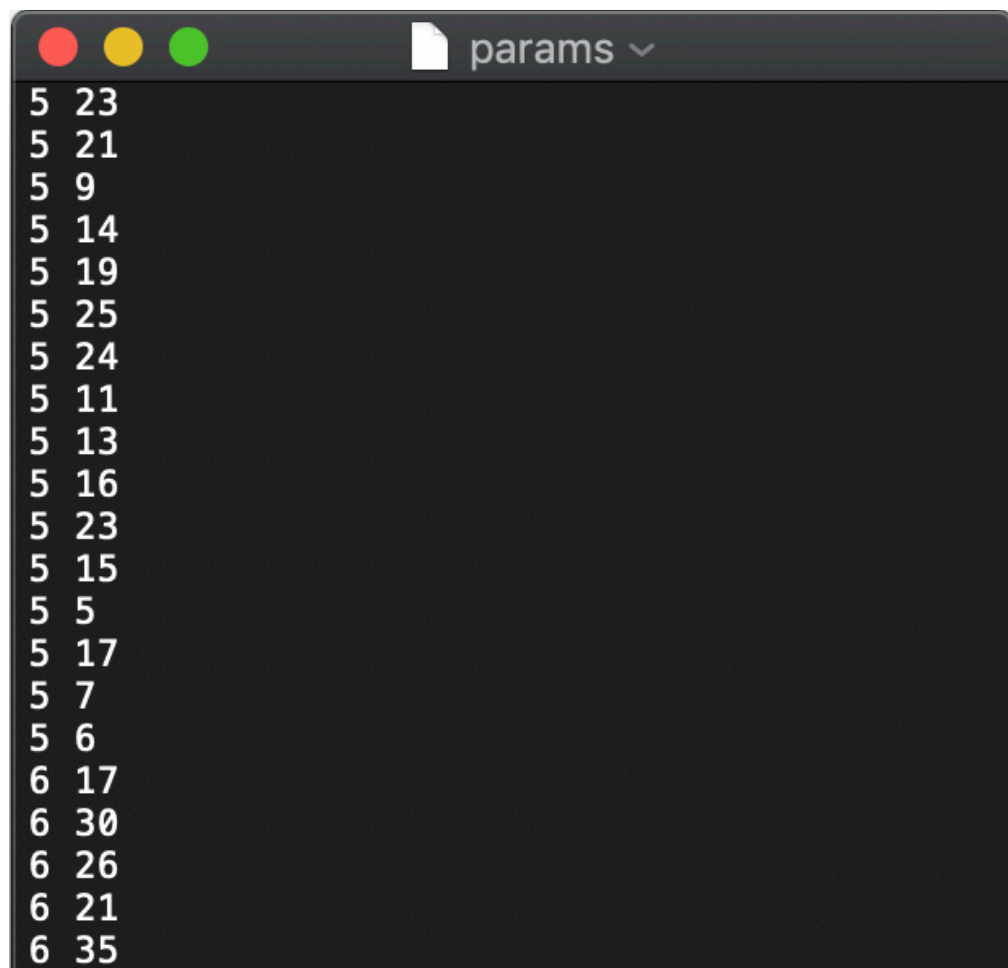
Algorithm

- The full project is uploaded in the repository on Git-Hub.
- The code is written with a CamelCase style
- Extra comment are on the full project

```
1 #pragma GCC optimize("Ofast")
2 #include <bits/stdc++.h>
3
4 using namespace std;
5 vector<long long int> enGraph[123456];
6 long long int curNodo;
7
8 int main() {
9     FILE* fin = fopen("input.txt", "r");
10    FILE* fout = fopen("output.txt", "w");
11    long long int N, V;
12    fscanf(fin, "%lli %lli", &N, &V);
13    long long int startNodo = 0;
14    long long int endNodo = N-1;
15    long long int a, b;
16    for (int i = 0; i < V; i++) {
17        fscanf(fin, "%lli %lli", &a, &b);
18        enGraph[a].push_back(b);
19        enGraph[b].push_back(a);
20    }
21    vector<bool> visited;
22    visited.resize(N);
23    vector<long long int> cost;
24    cost.resize(N);
25    memset(&cost[0], -1, N*sizeof(long long int));
26    cost[startNodo] = 0;
27
28    queue<long long int> q;
29    q.push(startNodo);
30    while ((!q.empty()) && (cost[endNodo] == -1)) {
31        curNodo = q.back();
32        q.pop();
33        for (auto e : enGraph[curNodo]) {
34            if (visited[e])
35                continue;
36            cost[e] = cost[curNodo] + 1;
37            q.push(e);
38            visited[e] = 1;
39        }
40    }
41    fprintf(fout, "%lli\n", cost[endNodo]);
42 }
```

User Guide

1. To run the code, all we need is just to run the file **run.sh**.



A terminal window with a title bar containing three colored circles (red, yellow, green) and a file icon labeled 'params'. The terminal displays a list of numbers, each preceded by a line number. The numbers are: 23, 21, 9, 14, 19, 25, 24, 11, 13, 16, 23, 15, 5, 17, 7, 6, 17, 30, 26, 21, 35.

```
5 23
5 21
5 9
5 14
5 19
5 25
5 24
5 11
5 13
5 16
5 23
5 15
5 5
5 17
5 7
5 6
6 17
6 30
6 26
6 21
6 35
```

```
1  #!/bin/bash
2
3  echo "Compiling..."
4  g++ ./gen.cpp -Ofast -o ./gen.o
5  g++ ./sol.cpp -Ofast -o ./sol.o
6
7  echo "Generating parameters..."
8  python3 genParams.py
9
10 echo "Testing..."
11 python3 test.py < params > res
12 echo "Done!"
13 echo 'Run "cat res" to view test results'
14
```

```
bash /path/to/script.sh
```

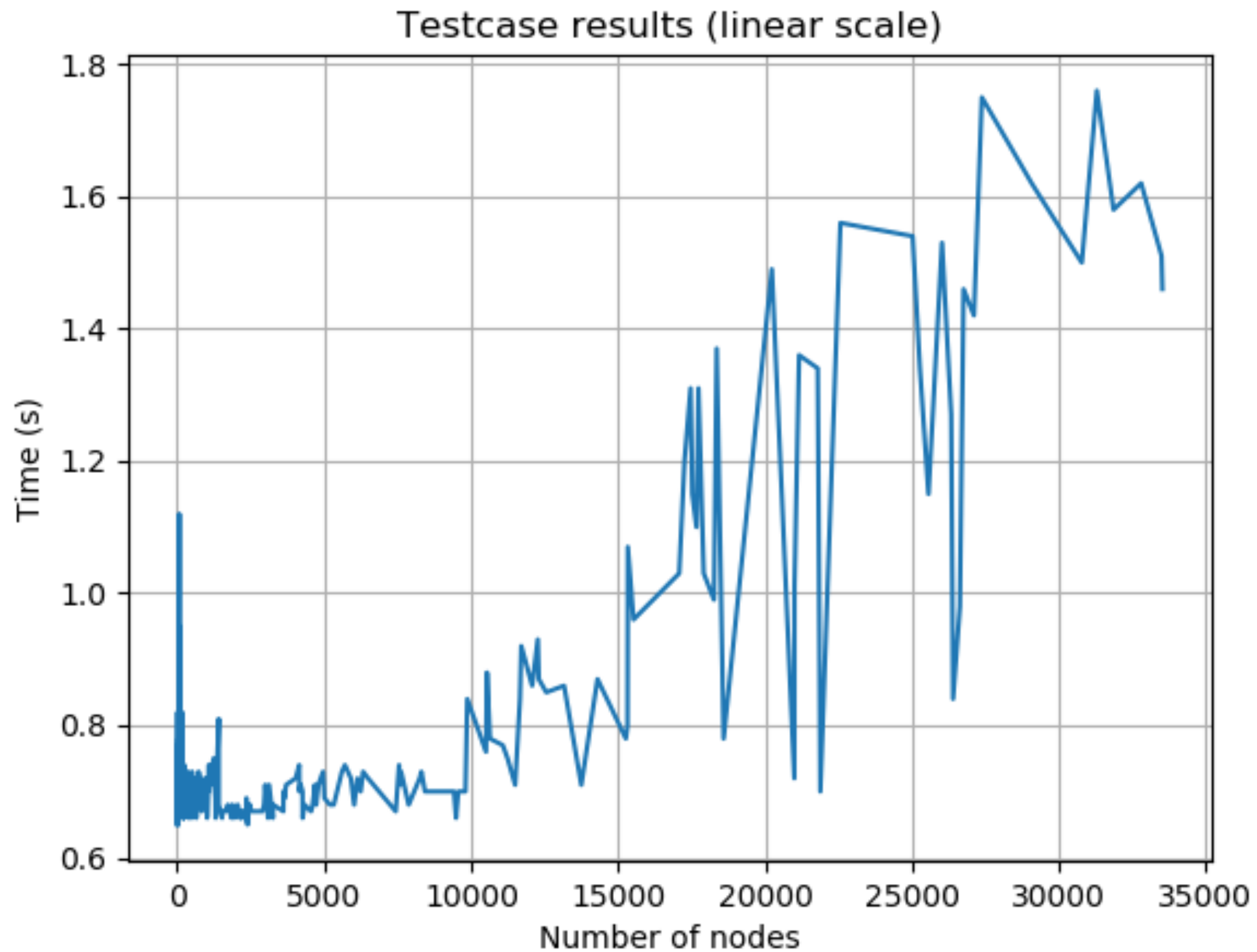
2. This script is able to **compile the generator** of parameters, it **tests it** and at the end it returns the results on the file **res**. Will be created also some **plots** based on the test results.

Test Results

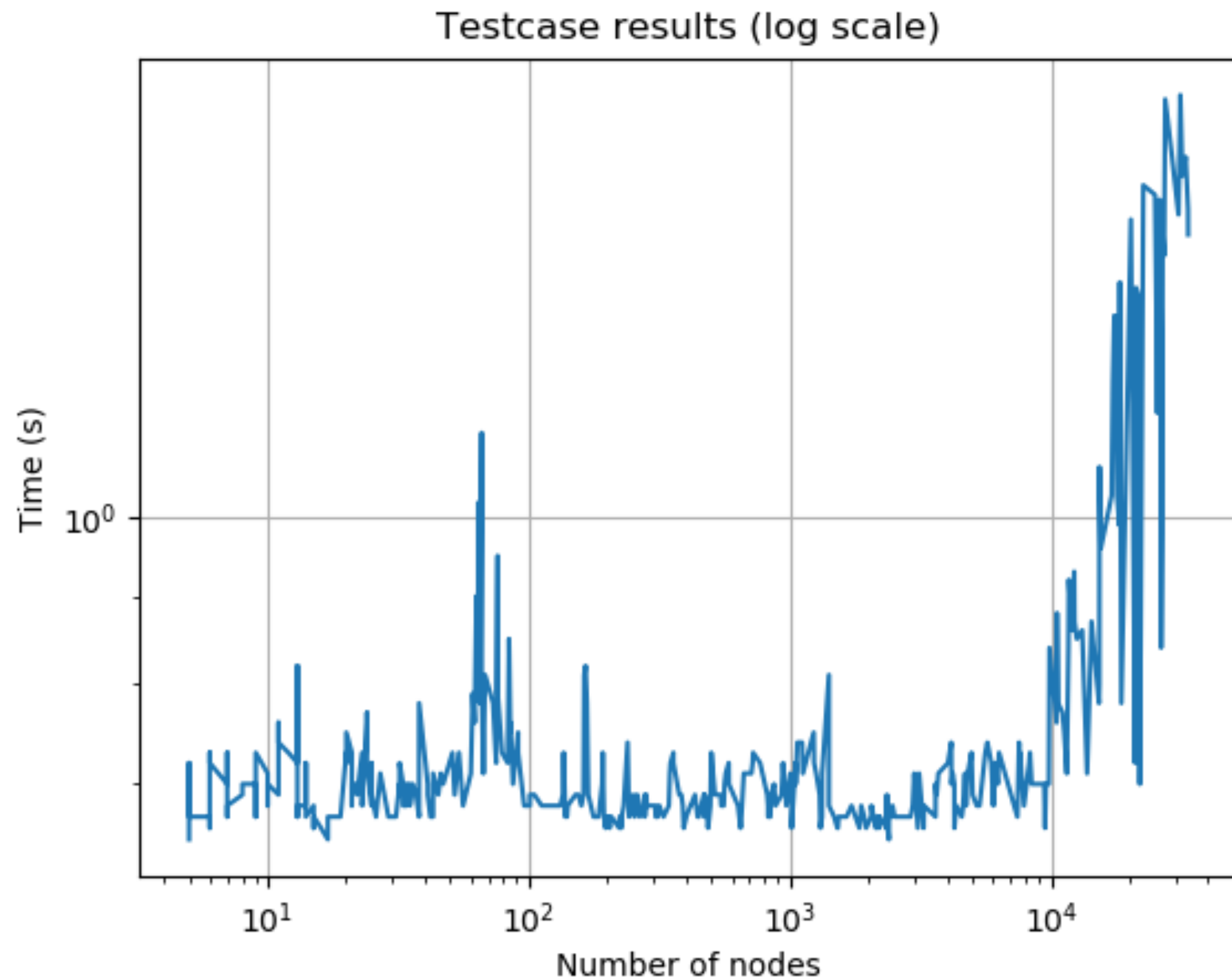
Nodes	Edges	Time (sec)	Nodes	Edges	Time (sec)	Nodes	Edges	Time (sec)	Nodes	Edges	Time (sec)
5	9	0.67	17	92	0.66	10540	94832526	0.88	21001	20246546	0.72
5	14	0.66	17	21	0.67	10643	47163130	0.78	21007	111205845	1.01
5	19	0.66	18	57	0.67	11078	42671090	0.77	21163	421451894	1.36
5	25	0.67	19	319	0.67	11242	42221146	0.75	21798	295346015	1.34
5	24	0.72	19	278	0.67	11499	7842265	0.71	21886	11983290	0.7
5	11	0.66	20	370	0.73	11664	70389426	0.84	22569	356391531	1.56
5	13	0.67	20	21	0.72	11708	10315874	0.92	25027	210658405	1.54
5	16	0.66	20	229	0.75	12079	39271909	0.86	25275	254782217	1.34
5	23	0.66	21	325	0.73	12270	11754570	0.93	25567	170688843	1.15
5	15	0.67	21	238	0.68	12305	15750155	0.87	26027	640914604	1.53
5	5	0.66	21	361	0.69	12559	6887880	.85	26340	480156665	1.27
5	17	0.66	21	234	0.7	13165	70279860	0.86	26403	55301082	0.84
5	7	0.66	22	450	0.7	13751	4575130	.71	26635	107826250	0.98
5	6	0.67	22	114	0.7	14302	79736328	0.87	26761	416957058	1.46
6	17	0.67	22	101	0.69	15267	39158667	0.78	27114	386686413	1.42
6	30	0.67	23	441	0.73	15323	34486807	0.8	27383	543328247	1.75
6	26	0.66	23	261	0.7	15334	13461121	1.07	29077	205012326	1.62
6	21	0.67	23	292	0.69	15526	93393275	0.96	30784	753721269	1.5
6	35	0.7	23	354	0.71	17083	11710759	1.03	31297	661641082	1.76
6	22	0.73	23	235	0.68	17263	19493223	1.2	31865	522173315	1.58
6	22	0.72	24	318	0.76	17471	28167688	1.31	32805	853377074	1.62
7	41	0.7	24	186	0.77	17527	18526678	1.15	33499	511916368	1.51
7	14	0.73	24	277	0.75	17671	16735519	1.1	33527	800836659	1.46
7	21	0.69	24	387	0.72						
7	37	0.67	25	423	0.68						
			25	36	0.72						
			25	427	0.69						

The entire set of results is contained in the res file

Plots



Plots



Critical Analysis

- From the plots we can see that our experimental data is compatible with our theoretical complexity analysis.
- We can understand that the time complexity of Dijkstra constant weight algorithm, in our case, is **linear** and implemented with a queue is a **Breadth First Search**.
- In general, some implementations of Dijkstra Algorithm are $O(n)$ when the graph has constant cost for each single edge.