Homework IV - Analysis of Algorithms

Edwing Alexis Casillas Valencia
November 16, 2025

Abstract

In this homework, we implement and analyze the Skip-list structure, the Bellman-Ford algorithm and the Dijkstra's algorithm.

PROBLEM 1

Skip List

(a) Implement in C the skip list data structure,
(b) And see if the calculated complexities are correct for insertion, deletion and search operations by using a large enough random dataset of insertions, deletion and searches. Please, try to obtain an average to each instruction and compared with the calculated complexities.

Answer

To test the implementation, we'll try to insert the following set of values 12, 23, 26, 31, 34, 44, 56, 64, 78, if the skip list is implemented correctly, we'll try to search te values 23, 34, 45, 78, delete the value 34 and insert 2 new values.

```
Inserting values: 12 23 26 31 34 44 56 64 78

Skip List (height: 4, size: 9):
Level 4: -inf -> 26 -> +inf
Level 3: -inf -> 23 -> 26 -> 78 -> +inf
Level 2: -inf -> 23 -> 26 -> 31 -> 64 -> 78 -> +inf
Level 1: -inf -> 23 -> 26 -> 31 -> 64 -> 78 -> +inf
Level 0: -inf -> 12 -> 23 -> 26 -> 31 -> 34 -> 44 -> 56 -> 64 -> 78 -> +inf

Search tests:
Key 23: FOUND
Key 34: FOUND
Key 45: NOT FOUND
Key 78: FOUND

Deleting key 34...
Skip List (height: 4, size: 8):
Level 4: -inf -> 26 -> +inf
Level 3: -inf -> 23 -> 26 -> 78 -> +inf
Level 2: -inf -> 23 -> 26 -> 31 -> 64 -> 78 -> +inf
Level 1: -inf -> 23 -> 26 -> 31 -> 64 -> 78 -> +inf
Level 0: -inf -> 12 -> 23 -> 26 -> 31 -> 44 -> 56 -> 64 -> 78 -> +inf

Inserting 15 and 45...
Skip List (height: 4, size: 10):
Level 4: -inf -> 26 -> +inf
Level 3: -inf -> 23 -> 26 -> 78 -> +inf
Level 2: -inf -> 23 -> 26 -> 31 -> 64 -> 78 -> +inf
Level 1: -inf -> 15 -> 23 -> 26 -> 31 -> 64 -> 78 -> +inf
Level 0: -inf -> 12 -> 15 -> 23 -> 26 -> 31 -> 44 -> 45 -> 56 -> 64 -> 78 -> +inf
```

Fig. 1: Skip List implementation

As we can see on the above image, we have positive results on the implementation. Now, to verify the theoretical complexities, I used large datasets to measure the behavior for every operation. The first image shows the average time behavior (O(log n)), and the second one the total time behavior (O(n log n)).

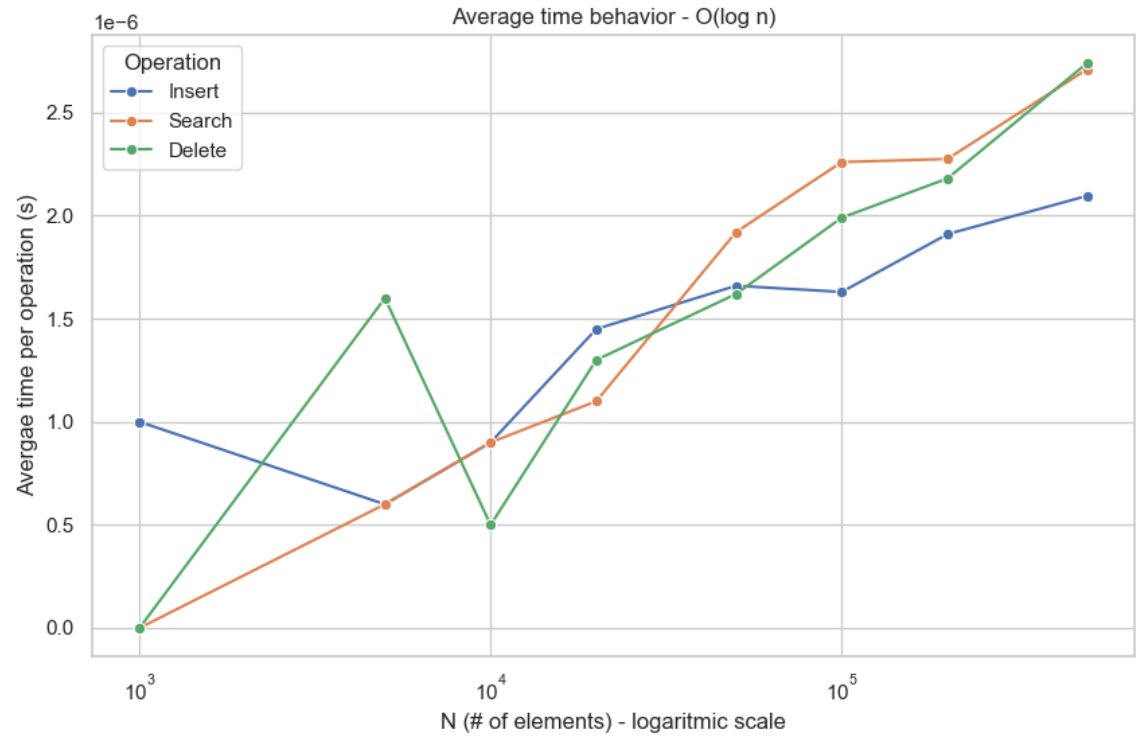

Fig. 2: Skip List average time (O(log n))

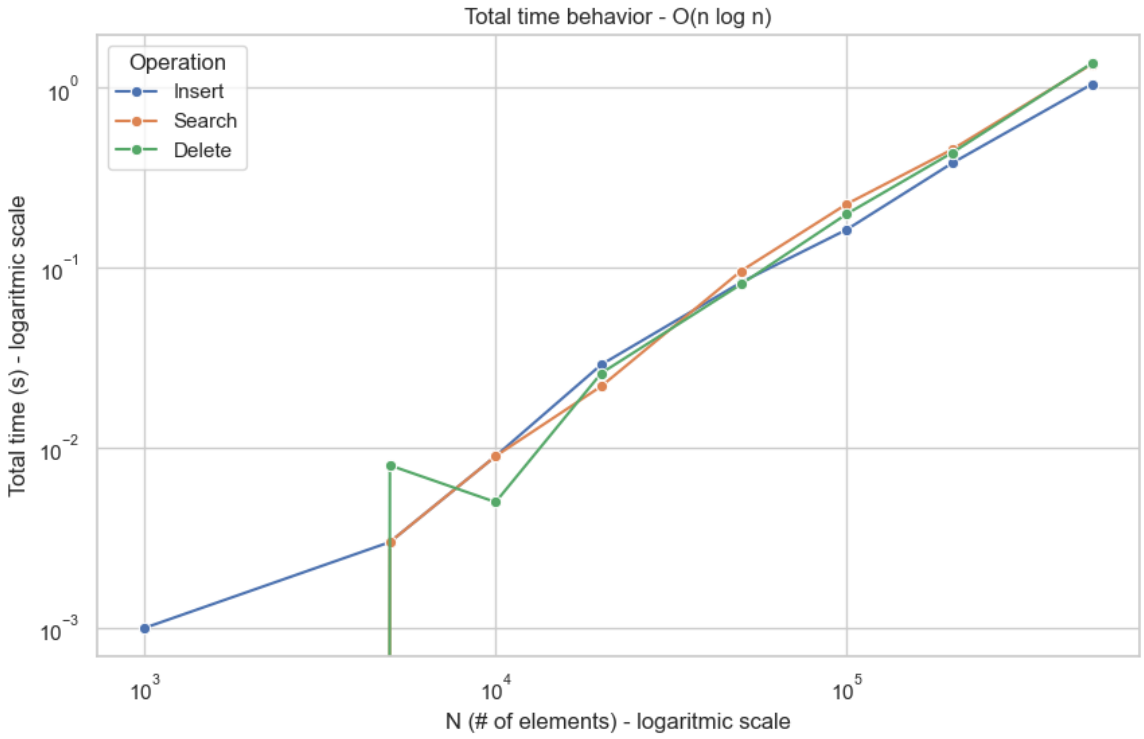

Fig. 3: Skip List total time (O(n log n))

As we can see, the average time grows slow, comparing it against a O(n) and is not flat as a O(1); on the total time, we observe that the behavior is similar to a straight line with slope 1, we can interpret it as something between O(n) and O(n^2),

so we can say that the theoretical complexities are correct.

PROBLEM 2

Implement in C the Bellman-Ford Algorithm:

(a) Test against a suitable large random graph and analyze its running time.
(b) Generate a graph with a negative weight cycle and see what happens with the algorithm.

Answer

First, to test if the implementation is correct, we'll try to find the shortest path and the minimum weights on the following graph.

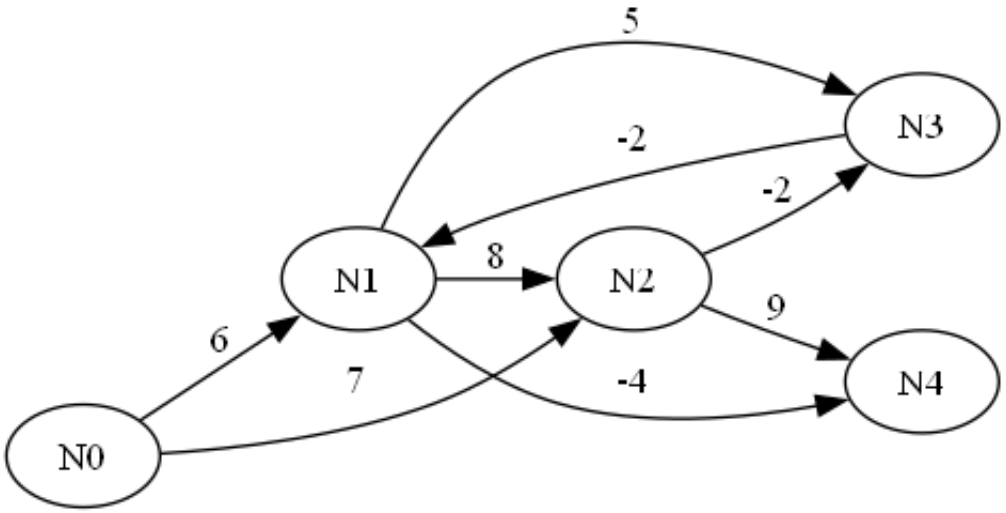

Fig. 4: Bellman-Ford

For this cycle we got the following

```
Initial graph
 (0) --(6)--> (1)
 (0) --(7)--> (2)
 (1) --(8)--> (2)
 (1) --(5)--> (3)
 (1) --(-4)--> (4)
 (2) --(-3)--> (3)
 (2) --(9)--> (4)
 (3) --(-2)--> (1)

there are not cycles with negative weights
distances and paths from vertex 0:
vertex 0: [0]   path: 0
vertex 1: [2]   path: 0 -> 2 -> 3 -> 1
vertex 2: [7]   path: 0 -> 2
vertex 3: [4]   path: 0 -> 2 -> 3
vertex 4: [-2]  path: 0 -> 2 -> 3 -> 1 -> 4
```

Fig. 5: Test Bellman-Ford

We can observe that the Bellman-Ford algorithm is working good for this part, it's giving us the costs and paths to move from vertex 0 to any other vertex on the graph. Now, to test the running time against a large graph, I'll introduce a set of large graphs to observe the behavior against them. We can observe the results on the following images.

```
Making graph with 100 vertices and 1000 edges...
result: with negative cycles
execution time: 0.0010 sec
theoric complex O(V*E): O(100*1000) = O(100000)

Making graph with 500 vertices and 5000 edges...
result: with negative cycles
execution time: 0.0530 sec
theoric complex O(V*E): O(500*5000) = O(2500000)

Making graph with 1000 vertices and 10000 edges...
result: with negative cycles
execution time: 0.1980 sec
theoric complex O(V*E): O(1000*10000) = O(10000000)

Making graph with 2000 vertices and 20000 edges...
result: with negative cycles
execution time: 0.6630 sec
theoric complex O(V*E): O(2000*20000) = O(40000000)

Making graph with 3000 vertices and 30000 edges...
result: with negative cycles
execution time: 1.5930 sec
theoric complex O(V*E): O(3000*30000) = O(90000000)

Making graph with 4000 vertices and 40000 edges...
result: with negative cycles
execution time: 2.6280 sec
theoric complex O(V*E): O(4000*40000) = O(160000000)

Making graph with 5000 vertices and 50000 edges...
result: with negative cycles
execution time: 4.4260 sec
theoric complex O(V*E): O(5000*50000) = O(250000000)
```

Fig. 6: Bellman-Ford verification complex

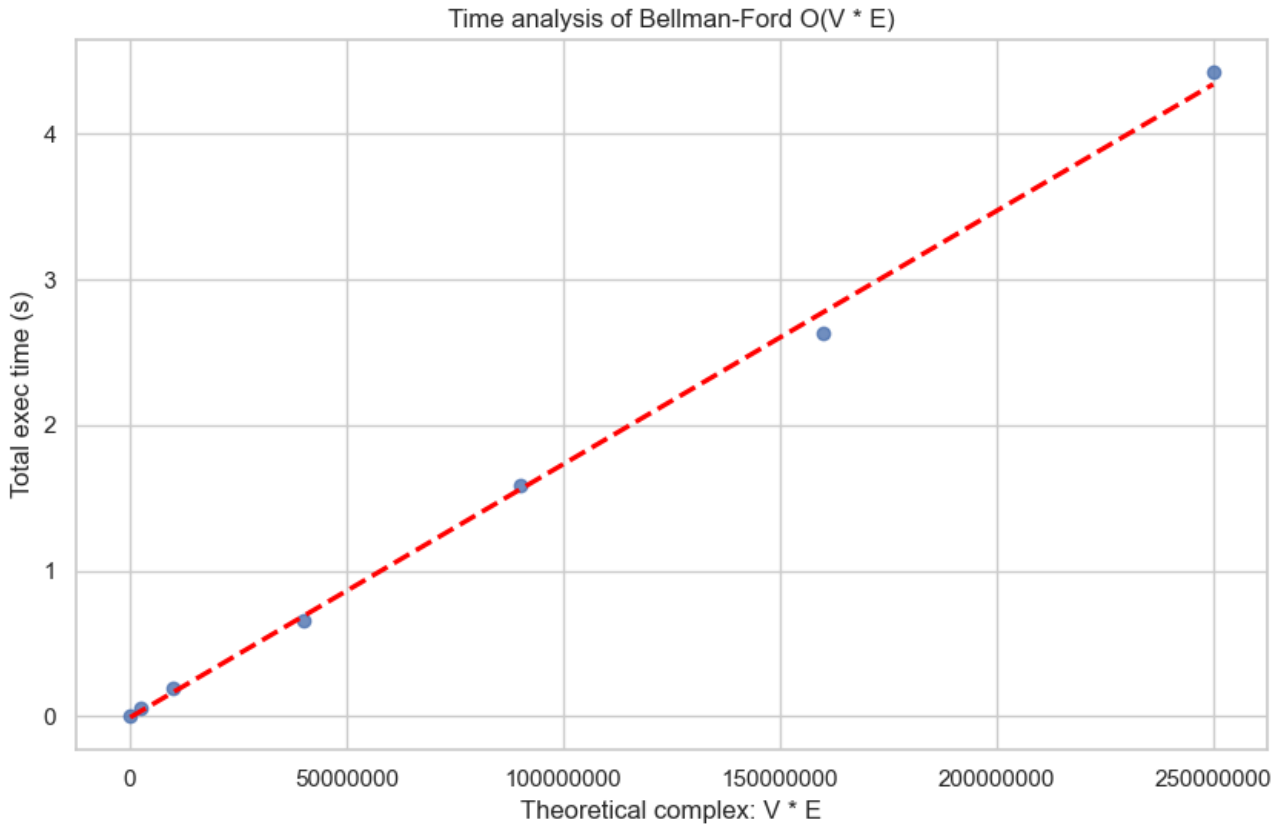

Fig. 7: Bellman-Ford behavior

As we can observe, we obtained a linear response from the Bellman-Ford algorithm, verifying the theoretical complexity of O(V*E).

Now we want to now what happens when we have a negative cycle into the graph, in theory, the algorithm is going to stop the process and return us a warning message, so we'll not know any cost or path from an origin vertex to another. To do this, I'm going to use the same graph that I used to test the code, I'll just change the weights from the cycle 1-2-3-1 to make it negative.

```
Initial graph
 (0) --(6)--> (1)
 (0) --(7)--> (2)
 (1) --(-8)--> (2)
 (1) --(5)--> (3)
 (1) --(-4)--> (4)
 (2) --(-3)--> (3)
 (2) --(9)--> (4)
 (3) --(-2)--> (1)

there is a cycle with negative weight
```

Fig. 8: Bellman-Ford negative cycle

Why the code stop the process? It stops because when the algorithm detects a negative cycle, there's no sense to find the shortest path, because we can move into an infinite cycle with an infinite cost, that's why the algorithm stop processing the calculations.

PROBLEM 3

Implement in C the Dijkstra's Algorithm:

(a) Test against a suitable large random graph and analyze its running time.

Answer

To test the implementation of Dijkstra, we'll use the following graph to find the shortest path from vertex 0 to the rest of the vertex.

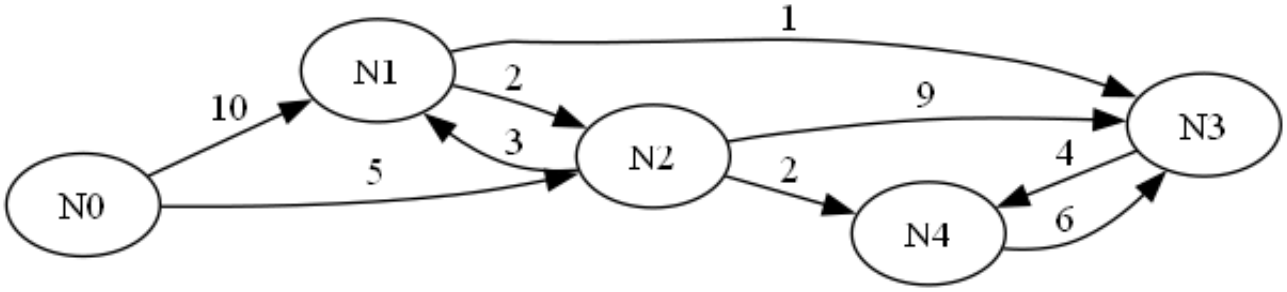

Fig. 9: Dijkstra example

The result we got is the following.

```
initial graph
Vertices: 5, Edges: 9
  (0) -> (2, w:5) | (1, w:10) |
  (1) -> (3, w:1) | (2, w:2) |
  (2) -> (4, w:2) | (3, w:9) | (1, w:3) |
  (3) -> (4, w:4) |
  (4) -> (3, w:6) |
------------------------------

distances from vertex 0:
vertex 0: 0 (predecessor: -1)  |        path: 0
vertex 1: 8 (predecessor: 2)  |        path: 0 -> 2 -> 1
vertex 2: 5 (predecessor: 0)  |        path: 0 -> 2
vertex 3: 9 (predecessor: 1)  |        path: 0 -> 2 -> 1 -> 3
vertex 4: 7 (predecessor: 2)  |        path: 0 -> 2 -> 4
```

Fig. 10: Dijkstra example solution

We can see that the algorithm to find the shortest path is working pretty good, now, it's time to test it against a large set of graphs to see that if it's behavior is the same as the theoretical O(E log V), because I'm implementing the binary min-heap to improve the complexity, otherwise, it shold be O(V^2).

```
Generating graph with V=1000, E=5000...
  Average time: 0.0013 seg.
Generating graph with V=5000, E=25000...
  Average time: 0.0059 seg.
Generating graph with V=10000, E=50000...
  Average time: 0.0117 seg.
Generating graph with V=20000, E=100000..
  Average time: 0.0291 seg.
Generating graph with V=40000, E=200000..
  Average time: 0.0822 seg.
Generating graph with V=50000, E=250000..
  Average time: 0.1060 seg.
```

Fig. 11: Dijkstra verification complex

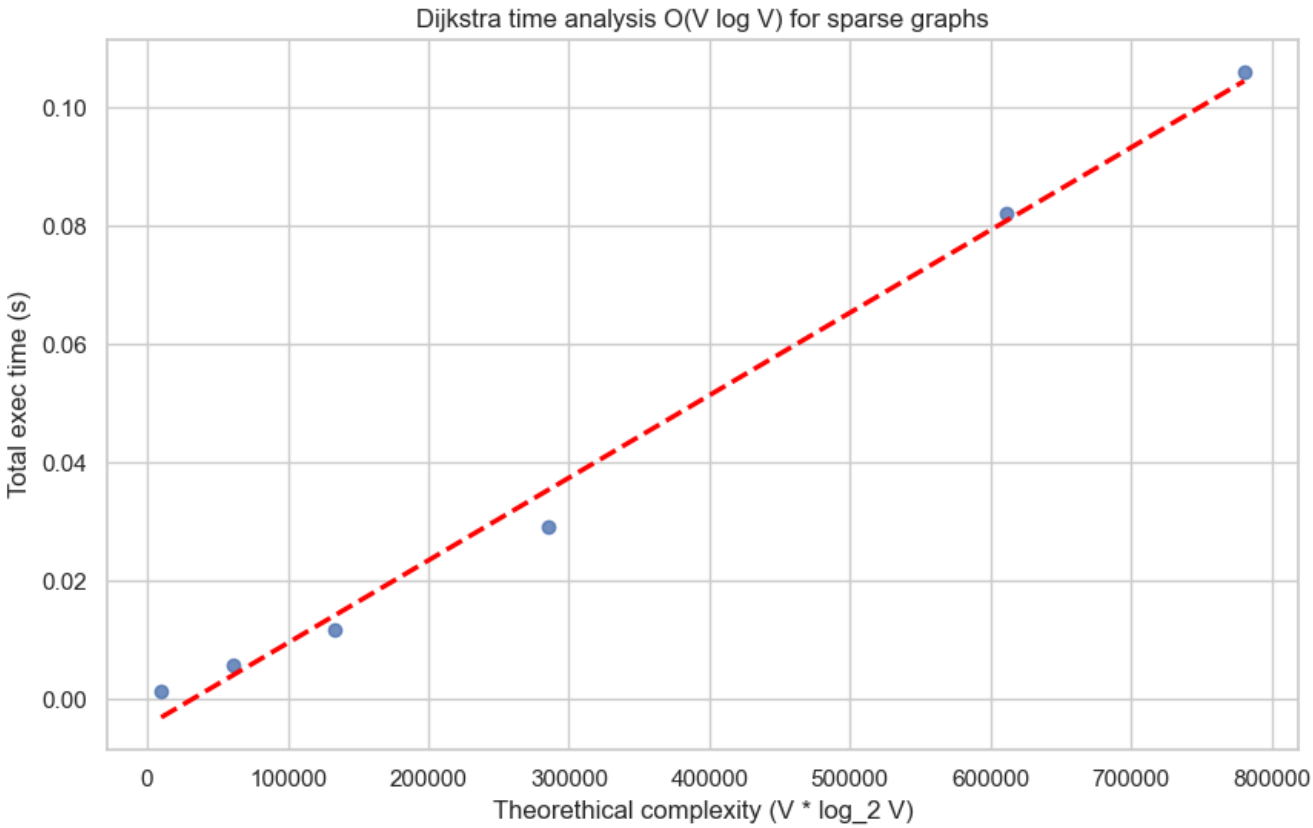

Fig. 12: Dijkstra behavior

Looking at the results, we can observe that the graph is almost linear, similar as the theoretical complexity of Dijkstra implementing the binary min-heap.