MEF UNIVERSITY

Computer Engineering

COMP 303 Analysis of Algorithm

Shortest Path in a Graph

Muhammed Rahmetullah Kartal - 041701008

Zeynep Ayyüce Çay – 041601021

Burak Özgür – 041601025

Date: 23.12.2019

Explanation and Time Complexity Analysis

The Dijkstra Algorithm is a single to many sources shortest path algorithm which can also find single to single source situations. The algorithm just works with the non-negative weights and can be implemented by heaps in this project we determined to use binary heap. Main idea behind the algorithm is walking onto graph without revisiting the visited vertices and at each path decision select the vertex which satisfies minimum total cost situation.

```
                                                                    Cost   Time
graph = defaultdict(list)                                           C1     1
for src,dst,weight in edges:                                        C2     E
    graph[src].append((weight,dst))                                 C3     E
heap, seen, weights = [(0, source,())], set(), {source:             C4     1
0}
while heap:                                                         C5     V
    (totalWeight,currSrc,path) = heapq.heappop(heap)                C6     VlgV
    if currSrc in seen: continue                                    C7     V
    seen.add(currSrc)                                               C8     V
    path += (currSrc,)                                              C9     V
    if currSrc == destination: return totalWeight, path             C10    1
    for weight, currDst in graph.get(currSrc, ()):                  C11    E
        if currDst in seen: continue                                C12    E
        prev = weights.get(currDst,None)                            C13    E
        next = totalWeight + weight ####                            C14    E
        if prev == None or next < prev:                             C15    E
            weights[currDst] = next                                 C16    E
            heapq.heappush(heap, (next, currDst, path               C17    ElgV
return99999999,()                                                   C18    1
```

Code 1: Analysis of Dijkstra Algorithm

(C1+C4+C10+C18) + (C2+C3+C11+C12+C13+C14+C15+C16) *E + (C5+C7+C8+C9) * V + (C6*VlgV) + (C17*ElgV) = O(ElgV)

As like in the analysis of Code 1, running time of the algorithm is O(ElgV) which E is number of Edges in the graph and V is number of vertices in the graph. Number of edges changes in different situations like, if graph is dense that number will be close to V^3, if it is sparse it will be close to V.

In the explanation of code there are 2 main pockets, a source and a destination vertex, one of the pockets is S(seen) and another is Q (not seen). At start, store and reposition all vertices into a min-priority queue, then repeatedly select the less weighted vertex which is adjacent to source from Q, add it to S, remove from Q and select as new source with a relaxed weight. After that continue the same process and relax all edges until the destination is reached.

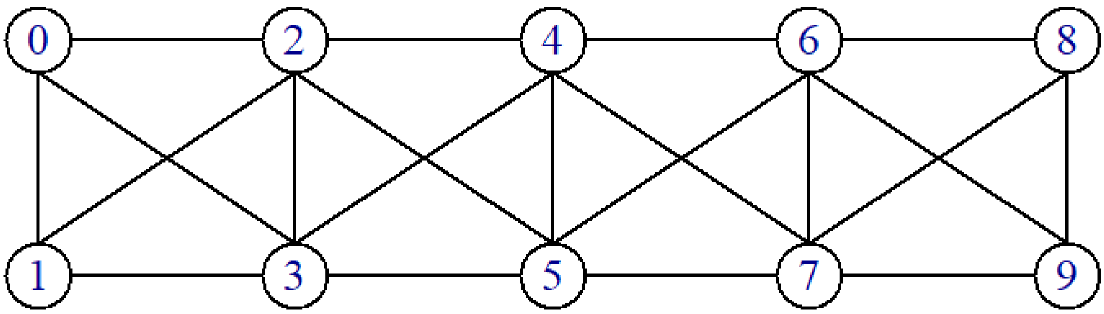


Figure 1: Example Graph

An example graph with vertices and edges is in Figure 1.

Implementation of Algorithm and Calculation of Edge Weights

ADDING EDGES

```
edges = list()
edges.append((0, 1, 1))
edges.append((0, 2, 2))
edges.append((0, 3, 3))
edges.append((1, 0, 1))
edges.append((1, 2, 3))
edges.append((1, 3, 4))
```

Code 2: Creating edges part 1

```
if (N % 2 == 0):
    for i in range(2, N, 2):
        if (i == N - 2):
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i + 1, 2*i+1))
        else:
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i + 1, 2*i+1))
            edges.append((i, i + 2, 2*i+2))
            edges.append((i, i + 3, 2*i+3))
    for i in range(3, N, 2):
        if (i == N - 1):
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i - 3, 2*i-3))
        else:
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i + 1, 2*i+1))
            edges.append((i, i + 2, 2*i+2))
            edges.append((i, i - 3, 2*i-3))
```

Code 3: Creating edges part 2

```
else:
    for i in range(2, N, 2):
        if (i == N - 1):
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
        elif (i == N - 3):
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i + 1, 2*i+1))
            edges.append((i, i + 2, 2*i+2))
        else:
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i + 1, 2*i+1))
            edges.append((i, i + 2, 2*i+2))
            edges.append((i, i + 3, 2*i+3))
    for i in range(3, N, 2):
        if (i == N - 2):
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i - 3, 2*i-3))
            edges.append((i, i + 1, 2*i+1))
        else:
            edges.append((i, i - 1, 2*i-1))
            edges.append((i, i - 2, 2*i-2))
            edges.append((i, i + 1, 2*i+1))
            edges.append((i, i + 2, 2*i+2))
            edges.append((i, i - 3, 2*i-3))
return edges
```

Code 4: Creating edges part 3

As like in Code 2, Code 3 and Code 4, first index is vertex1, second index is vertex2 and third index is weight.

STEPS OF DIJKSTRA

1. Extract a vertex u from Q
2. Insert U to S
3. Relax all edges leaving u
4. Update Q

Implementing Steps

1-S=<> Q=<0,1,2,3,4,5,6,7,8,9> extract 0 and set source at 1. All points except 1 become infinite.

2- S=<1> Q=<2,3,4,5,6,7,8,9> it can go 2 and 3. Since the weight of [1,2] is less than [1-3], 2 will be selected as new vertex.

3-- S=<1,2> Q=<5,6,7,8,9,10> We found the shortest way 1 to 2. It will go to 4 because path with 4 will be lighter.

4- S=<1,2,4> Q=<7,8,9,10> We found the shortest way 1 to 2 to 4. It will go to 6 because path will be lighter with 6.

5-S<1,2,4,6> Q=<7,8,9,10> We found shortest way 1 to 2 to 4 to 6. We are in the destination right now, but we will try others because it may produce a lighter way.

6-S<1,2,4,6> Q=<> The lighter path didn't change at relaxing stage so, the shortest path is 1 to 2 to 4 to 6.

Heap is a list of tuples. Seen is a set which has unique seen elements. Weights is a dictionary that holds weight of each source.

```
heap, seen, weights = [(0, source,())], set(), {source: 0}
```

Code 5: Heap, seen, weight implementation

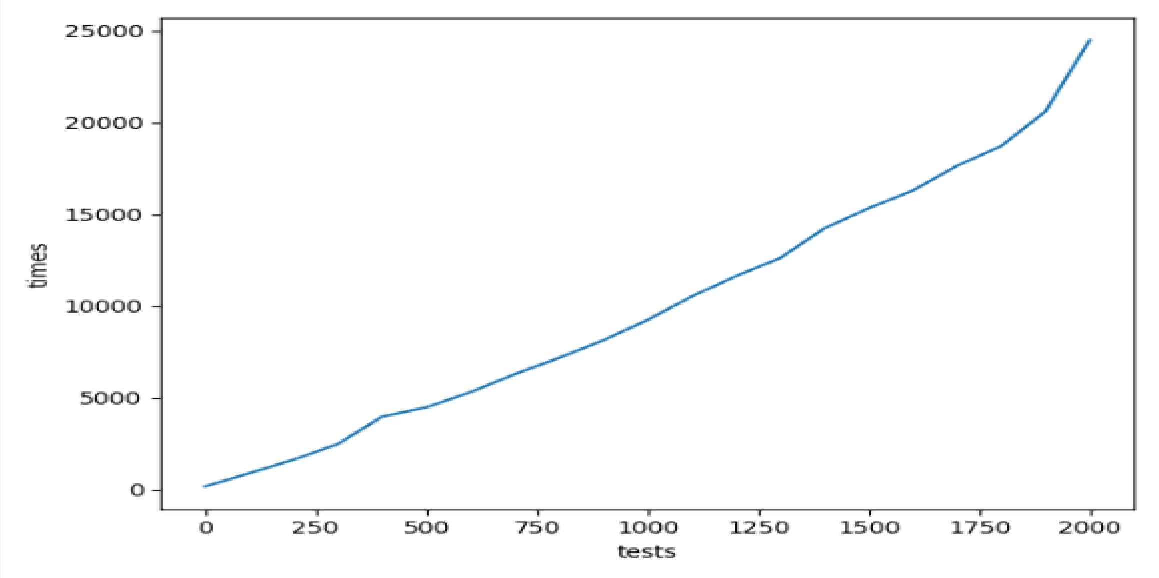
```
while heap:

    (totalWeight,currSrc,path) = heapq.heappop(heap)
    if currSrc in seen: continue
    seen.add(currSrc)
    path += (currSrc,) #add new source to path

    if currSrc == destination: return totalWeight, path
```

Code 6: Code to find optimal path

```
for weight, currDst in graph.get(currSrc, ()):
    if currDst in seen: continue
    prev = weights.get(currDst,None)
    next = totalWeight + weight
    if prev == None or next < prev:
        weights[currDst] = next
        heapq.heappush(heap, (next, currDst, path))
```

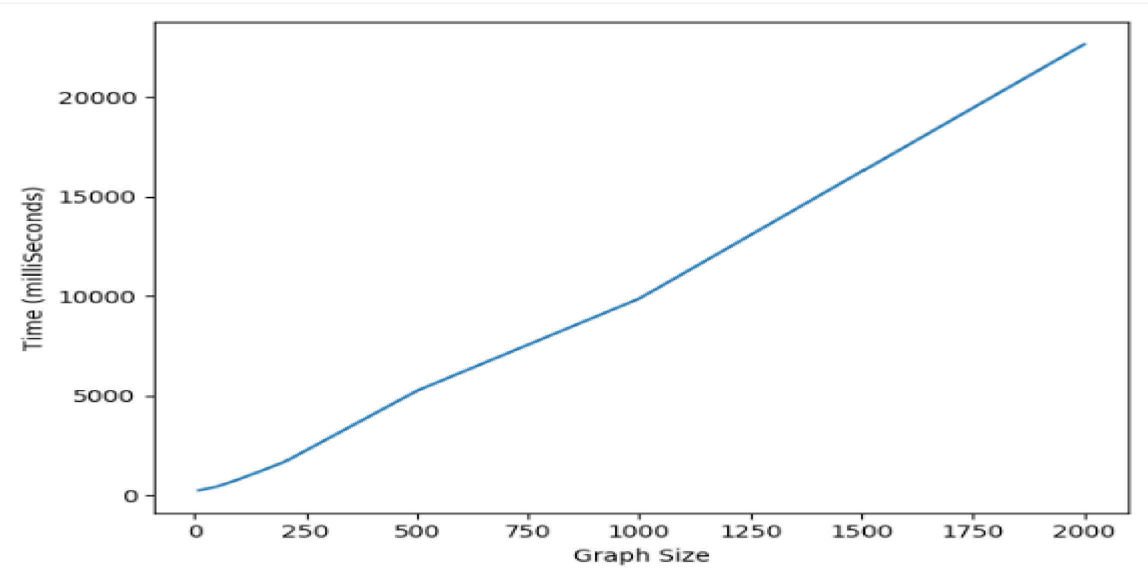
Code 7: Code to fill the weights of whole graph

Visualizing and Interpreting the Process Time Results

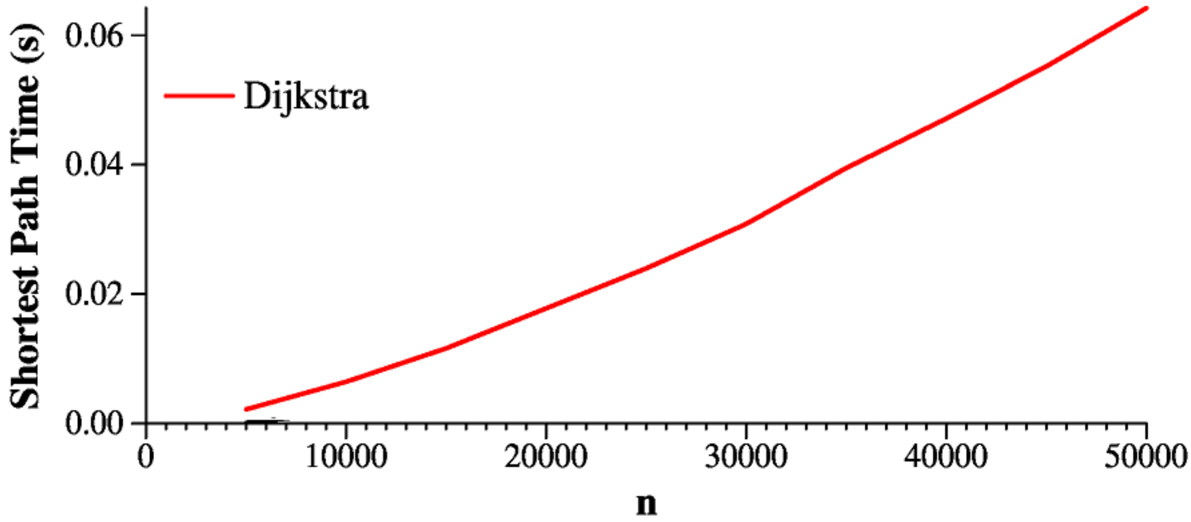


Graph 1: Test sizes 0 to 2000 increasing 100 – Time: milliseconds

The graph 1 is a test graph which have test values 0-2000 increasing by 100, these values are to better visualize process time O(ElgV). In our program the running time of Dijkstra Algorithm will be O(V^3lgV) because it is implemented by using binary heap and it is a dense graph which means have too much edges between vertices. If algorithm is implemented in sparse graph the running time will be O(VlgV).



Graph 2: Test sizes [10,50,100,200,500,1000,2000] - Time : milliseconds



Graph 3: Dijkstra Algorithm Theorical Graph

As you can see the test graphs (Graph 1 and Graph 2) are almost same with the Graph 3, that means Dijkstra algorithm implemented fine.

```
Size 10 - 224 milliseconds
Size 50 - 529 milliseconds
Size 100 - 903 milliseconds
Size 200 - 1872 milliseconds
Size 500 - 5685 milliseconds
Size 1000 - 10462 milliseconds
Size 2000 - 22484 milliseconds
```

Figure 2: Test Sizes and Running Times

How Graphical User Interface Works

Figure 3: Start screen of GUI

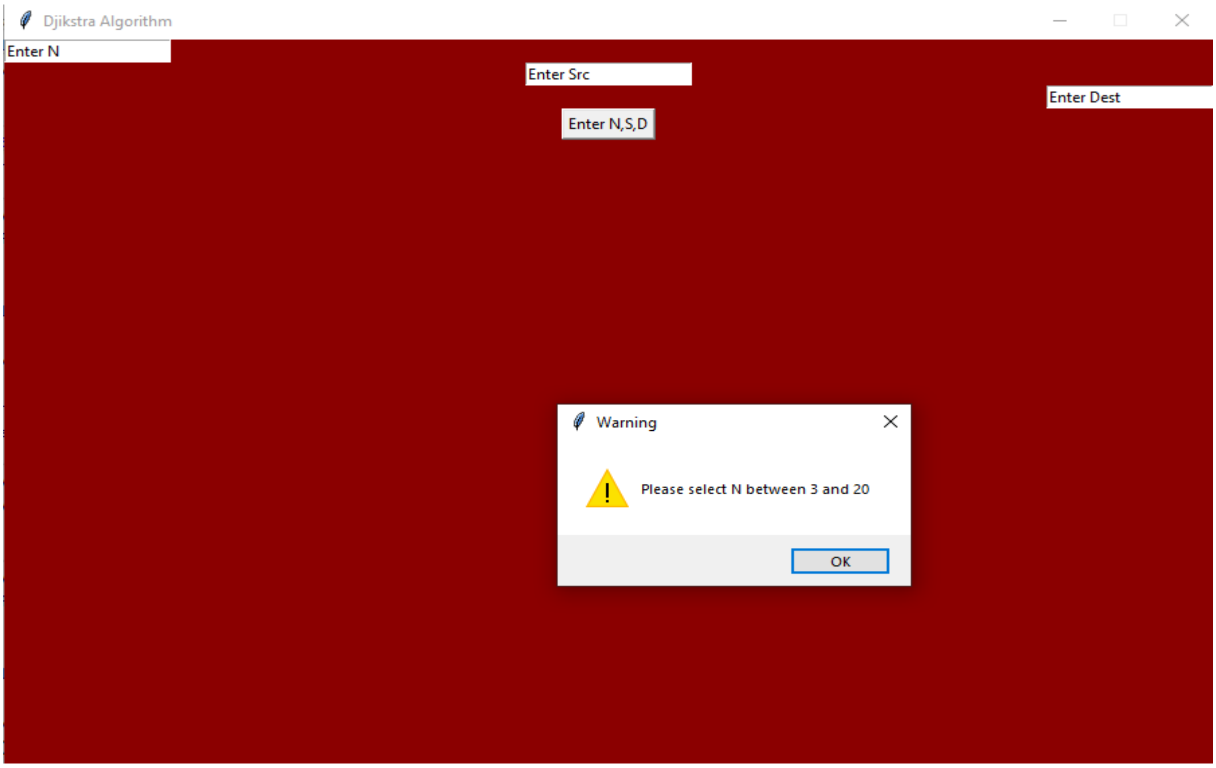


Figure 4: Warnings for Unwanted Situations

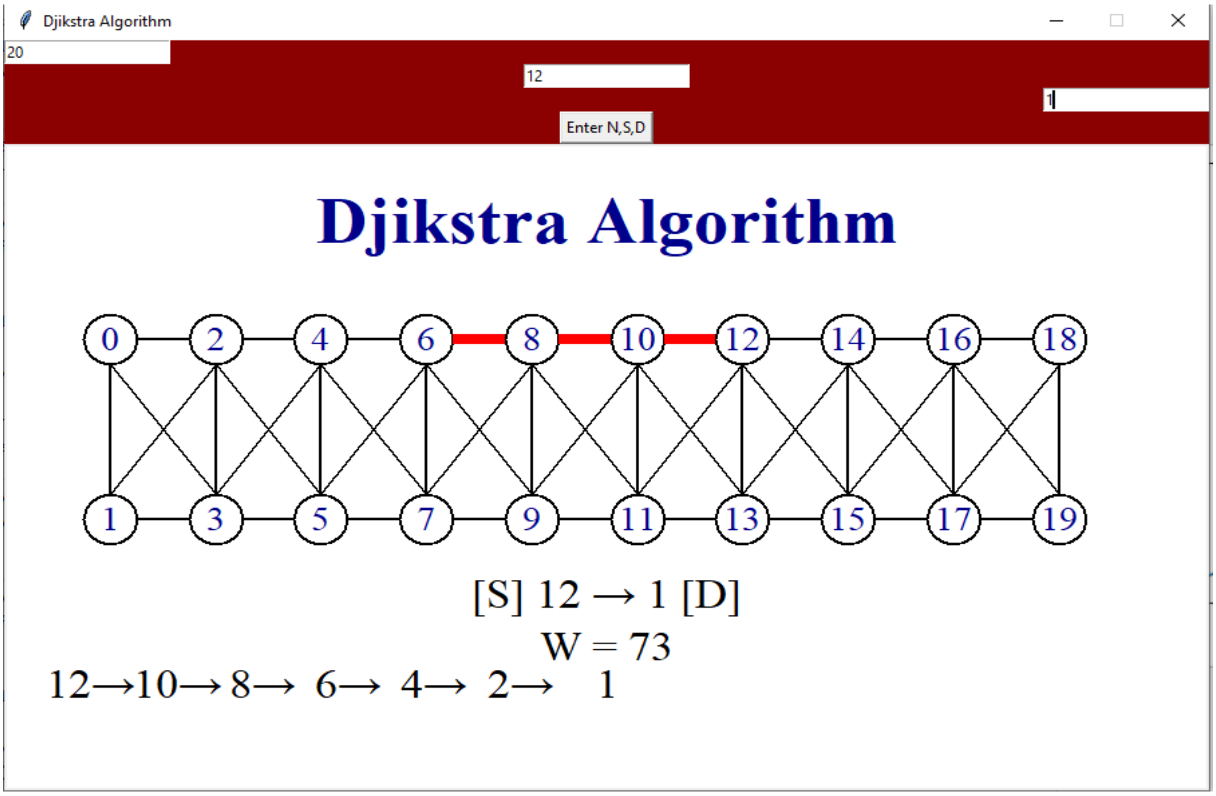


Figure 5: Running the Algorithm and Start of Iterations

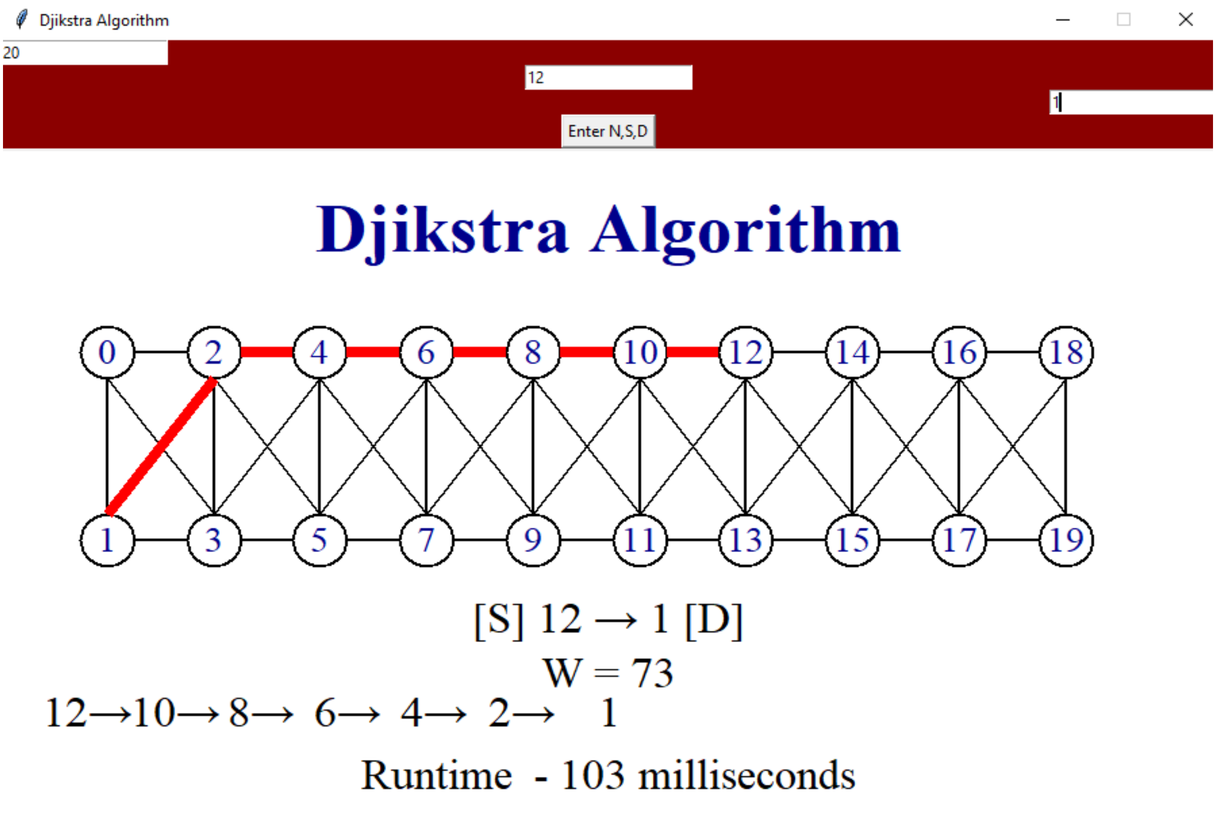


Figure 6: End of Iterations and Last Scene

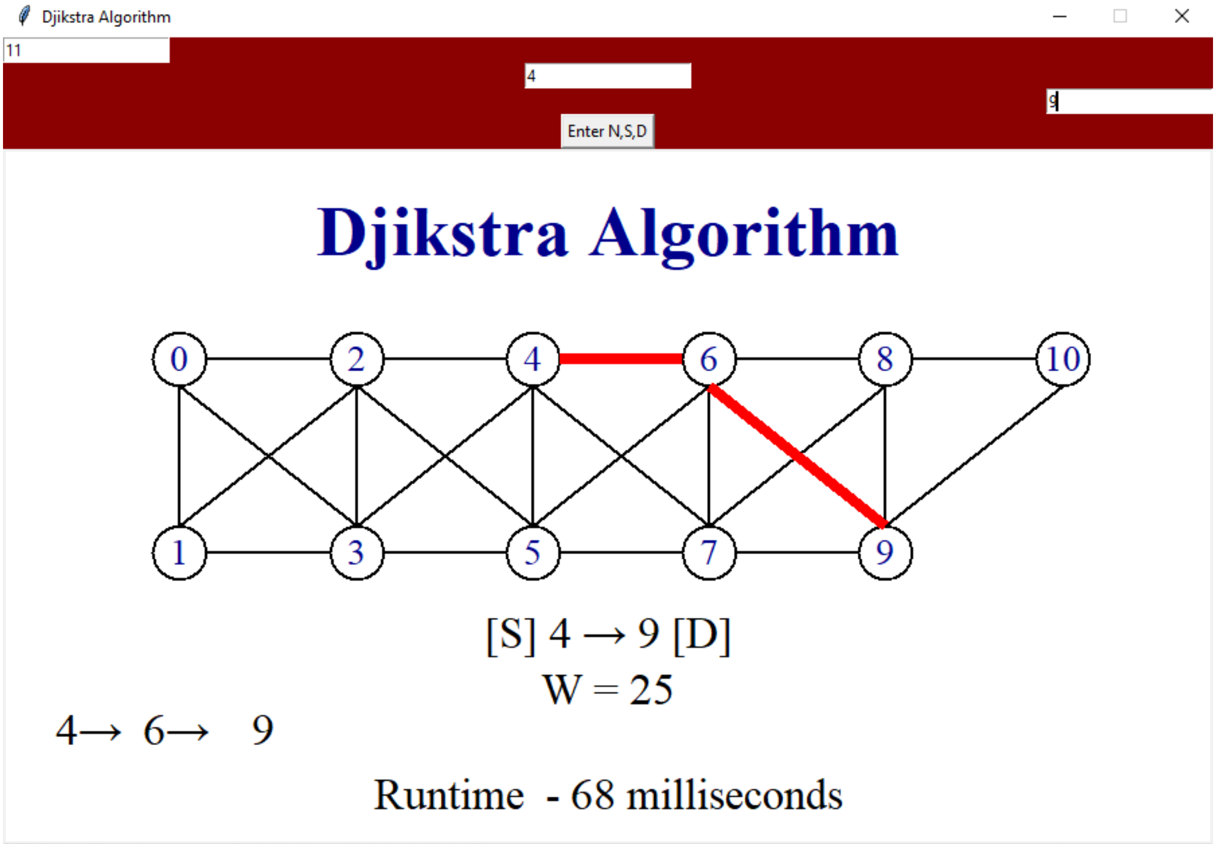


Figure 7: Another try with different values