INFSCI 2591- ALGORITHM DESIGN
Assignment-3

JDK used: Netbeans 8.2
Report Editor: Microsoft Word 2011 (14.7.7)
Operating System: Mac OS Sierra (10.12.6)

Floyd's Algorithm

Pseudo code:
Problem: Compute the shortest paths from each vertex in a connected, weighted and undirected graph to each of the other vertices. The weights are non-negative numbers.
Inputs: A weighted graph and the number of vertices in the graph (n). The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where W[i][j] is the weight on the edge from i^{th} vertex to j^{th} vertex.
Output: A two-dimensional array D, which has both its rows and columns indexed from 1 to n, where D[i][j] is the length of the shortest path from i^{th} vertex to j^{th} vertex.

```
Void Floyd (integer n, constant number W[][] ,number D[][])
{
index I, j, k ;
D=W; for ( k=1; k <= n ; k++ )
For ( I =1 ; I < = n ; i++ ) for ( j = 1 ; j < = n ; j++ )
D [ I ][ j ]=minimum (D[ I ][ j ], D[ I ][ k ] + D[ k ][ j ]);
}
```

Floyd's algorithm is implemented using three data structures, one-dimensional array, two-dimensional array and a linked-list.

Floyd's Algorithm using One-dimensional Array:

Number of nodes	Time to construct adjacency matrix	Time to compute shortest path	Total time taken:
100	0.005 seconds	0.031 seconds	0.036 seconds
1000	0.046 seconds	5.204 seconds	5.25 seconds
2000	0.189 seconds	11.558 seconds	11.747 seconds
3000	0.417 seconds	38.652 seconds	39.069 seconds
4000	0.63 seconds	98.37 seconds	99.0 seconds
5000	1.557 seconds	213.233 seconds	214.79 seconds
6000	2.115 seconds	377.083 seconds	379.198 seconds

Table-1

Table-1 shows the time taken to create adjacency matrix for *random numbers* using one-d array.

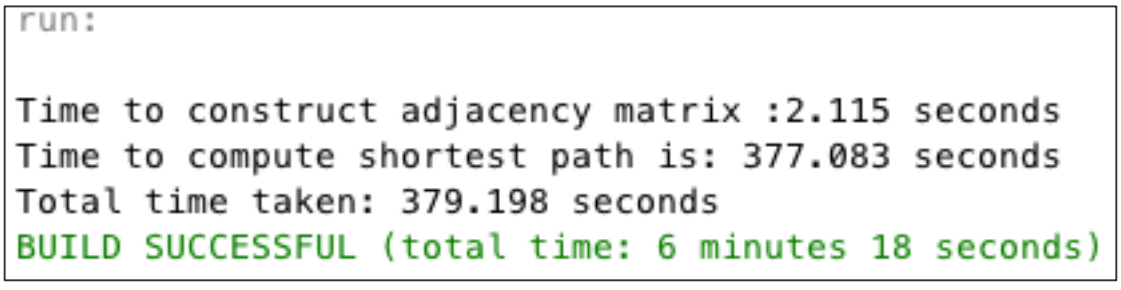


Figure- 1

When given the number of nodes=6000, we get the output shown in figure 1.

Floyd's Algorithm using two-dimensional array:

Number of nodes	Time to create Adjacency Matrix	Time to generate input and compute shortest path	Time to compute shortest if input is provided
100	0.095seconds	0.126seconds	6.613seconds
1000	7.96seconds	9.25seconds	12.36seconds
2000	26.675seconds	35.532seconds	31.038seconds
3000	56.645seconds	83.118seconds	117.845seconds
4000	95.485seconds	156.034seconds	128.112seconds
5000	134.491 seconds	387.42 seconds	241.55 seconds

Table-2

Table-2 shows the time taken to create adjacency matrix for *random numbers* using two-d array.

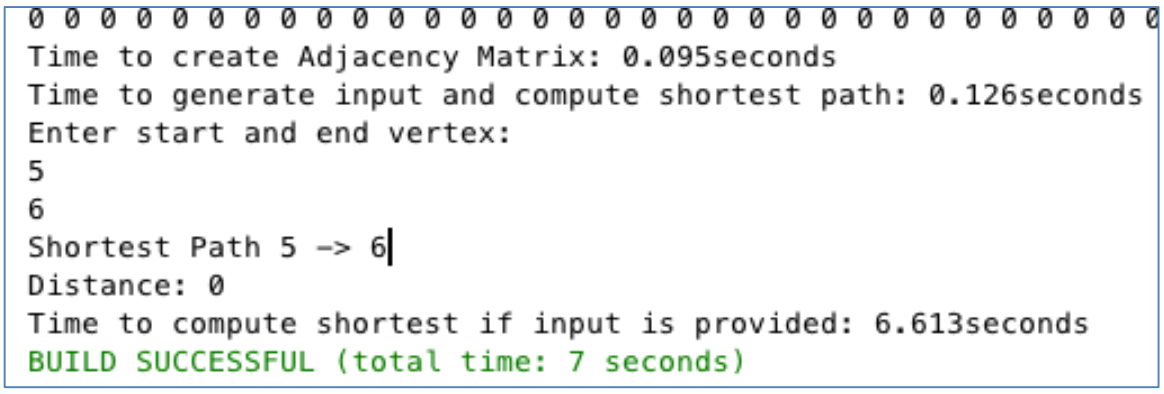


Figure-2

When given the number of nodes=100, we get the output shown in Figure 2.

Floyd's algorithm using linked-list:

Number of nodes	Time to create adjacency matrix	Time to generate input and compute shortest path	Total time
100	0.201 seconds	0 seconds	0 seconds
200	0.216 seconds	6 seconds	6 seconds
300	1.07 seconds	36 seconds	37 seconds
400	1.365 seconds	51 seconds	52 seconds
500	1.645 seconds	123 seconds	124 seconds

Table-3

Table-3 shows the time taken to create adjacency matrix for *random numbers* using a linked-list.

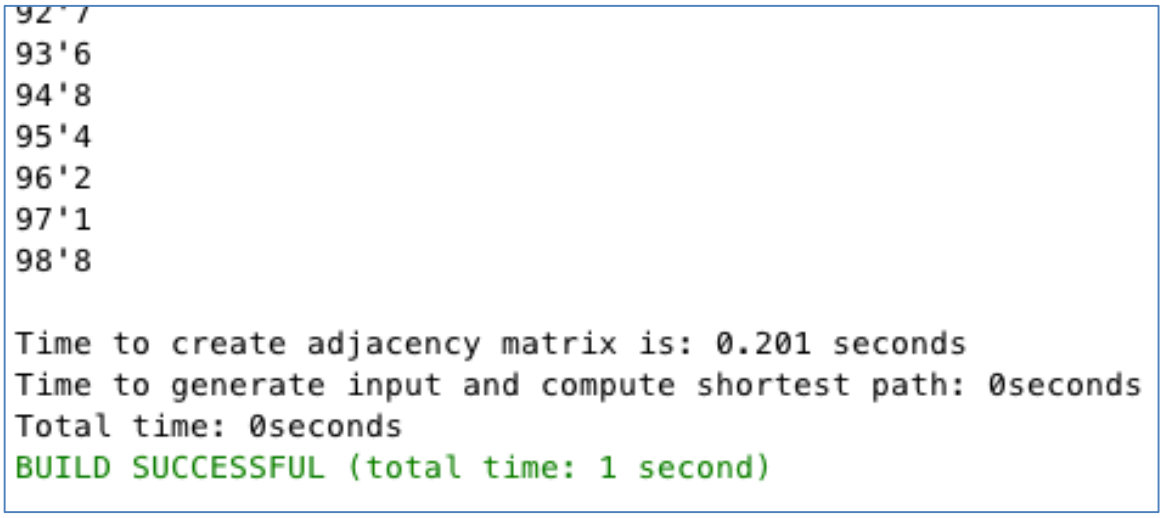


Figure-3

When given the number of nodes=100, we get the output shown in Figure 3.

the system crashed on inputting number of nodes=1000

Dijkstra's Algorithm :

Pseudo code :

Problem: Determine the shortest path from v1 to all other vertices in weighted graph.

Inputs: integer n>=2 and a connected weighted undirected graph with n vertices.

The graph is represented by a two dimensional array W, which has both its row and column indexed from 1 to n, where W[i][j] is the weight on the edge from ith vertex to jth vertex.

Output: Set of edges F containing edges in shortest path

```
Void Dijkstra*int n, const number W[][], set_of_edges &F)
{
index i, vnear;
edge e;
index touch [2..n];
number length [2..n];
F=Ø;
for(i=2;i<=n;i++){
touch[i]=1;
length[i]=W[1][i];
}

                              //Add all n-1 vertices to Y
repeat(n-1 times)
{
min=∞;                        //Check each vertes of having shortest path
for(i=2;i<=length;i++)
```

```
if(0<=length[i]<min)
{
min=length[i];
vnear=i;
}
e=edge from vertex indexed by touch[vnear] to vertex indexed by
vnear;
        add e to F;         //For each vertex not in Y , update its
                            shortest path.
for(i=2; i<=n;i++)
                            //Add vertex indexed by vnear to Y
if(length[vnear]+W[vnear][i]<length[i]){
length[i]=length[vnear]+W[vnear][i];
touch[i]=vnear;
}
length[vnear=-1;]
}
}
```

Dijkstra's algorithm is implemented using three data structures, one-dimensional array, two-dimensional array and a linked-list.

One-dimensional Array:

Number of nodes	Time required for creating adjacency matrix	Time to generate computing shortest path	Overall time taken
100	0.14 seconds	0.142 seconds	0.282 seconds
200	0.182 seconds	0.187 seconds	0.369 seconds
300	0.646 seconds	0.658 seconds	1.304 seconds
400	0.832 seconds	0.845 seconds	1.677 seconds
500	1.715 seconds	1.73 seconds	3.445 seconds

Table-4

Table-4 shows the time taken to create adjacency matrix for *random numbers* using a one-d array.

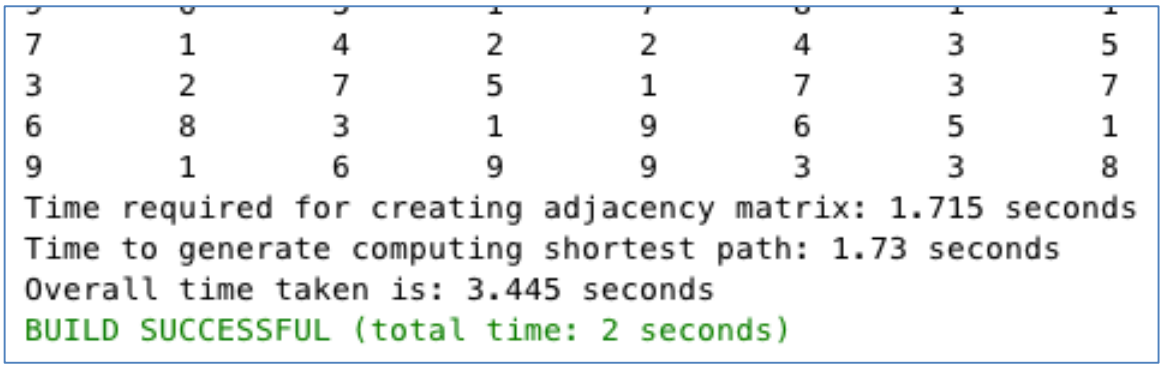


Figure-4

When given the number of nodes=500, we get the output shown in Figure 4.

Two-Dimensional Array:

Number of nodes	Time required for creating adjacency matrix	Time to generate computing shortest path	Overall time taken
100	0.096seconds	0.101seconds	0.197 seconds
1000	1.565 seconds	1.578 seconds	3.143 seconds
2000	6.666 seconds	6.692 seconds	13.35 seconds
3000	15.634 second	15.66 sec	31.296 sec
4000	26.61 sec	26.67 sec	53.268 sec

Table-5

Table-5 shows the time taken to create adjacency matrix for *random numbers* using a two-d array.

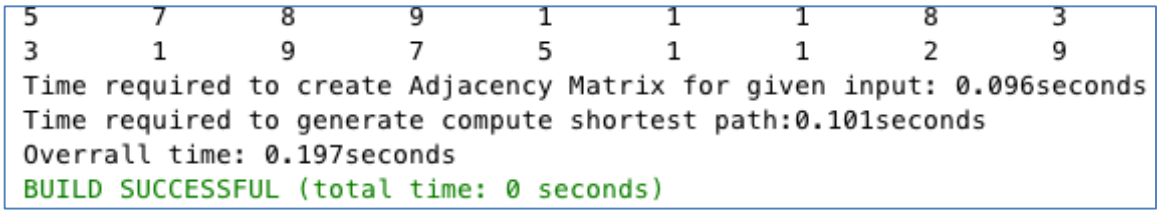


Figure-5

When given the number of nodes=100, we get the output shown in Figure 5

Linked list implementation of Dijkstra's algorithm:

Number of nodes	Time required for creating adjacency matrix	Time to generate computing shortest path	Overall time taken
100	0.004 seconds	0.116 seconds	0.12 seconds
1000	0.037 seconds	6.574 seconds	6.611 seconds
2000	0.927 seconds	28.7 seconds	29.62 seconds
4000	4.012 seconds	121.65 seconds	125.66 seconds
5000	4.609 seconds	216.069 seconds	220.678 seconds
6000	15.113 seconds	329.94 seconds	345.053

Table-6

Table-6 shows the time taken to create adjacency matrix for *random numbers* using a two-d array.

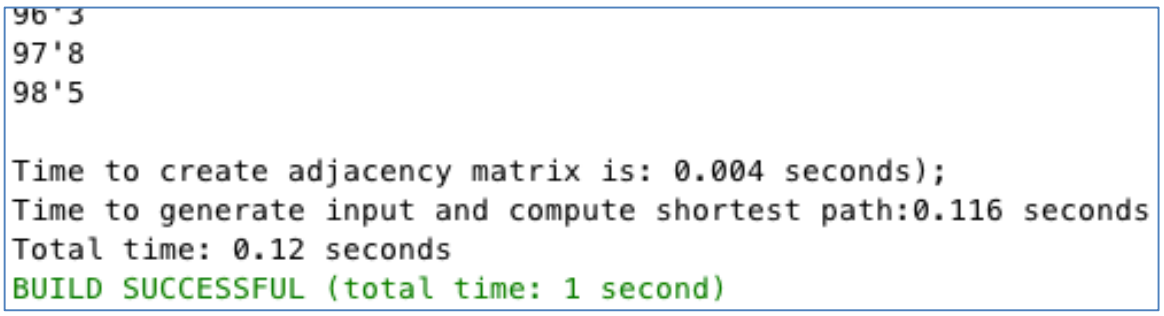


Figure-6

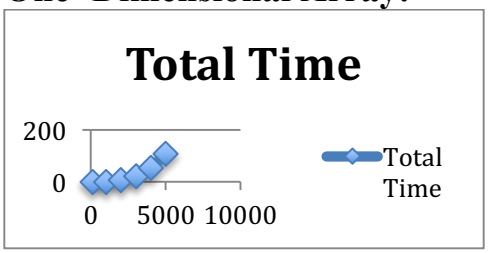
When given the number of nodes=100, we get the output shown in Figure 6

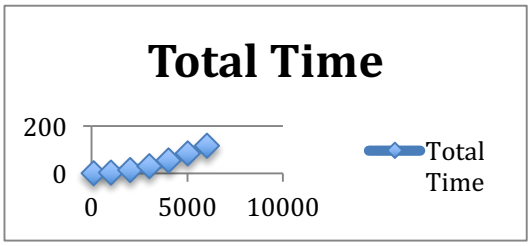
Time Complexity Analysis:

Comparison of time taken by the 2 algorithms, using data structures:

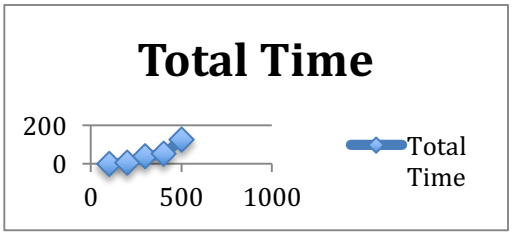
FLOYD'S DIJKSTRA'S

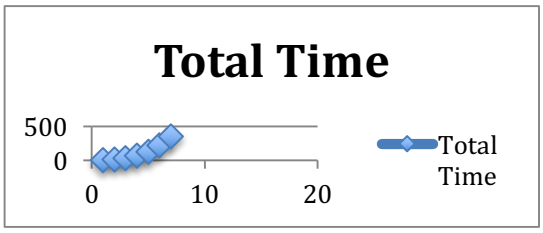
One- Dimensional Array:



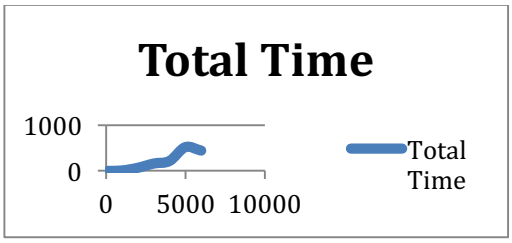


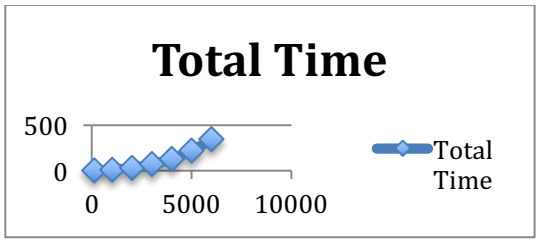
Linked List:





Two- Dimensional Array:





We calculated the time taken by different data structures in the two algorithms to compute shortest distances. They are plotted in the graphs above. It was noted that with the increase in number of nodes, the graph took

more significantly more time to compute the optimum distances. By comparing the three graphs above, it can be seen that one-dimensional arrays take the least amount of time as compared to the other data structure, i.e. a two-dimensional array and linked lists.

One more important inference, the optimization algorithm provided by Djikstra performs better than that of Floyd. All the above graphs when compared row-wise show that the time taken by Djikstra's algorithm is less than that of Floyd. Or we can say that greedy algorithms perform better than dynamic algorithms when time constraints are in consideration.

The time complexity is as follows:
Floyd's Algorithm: $O(n)=n^3$
Dijkstra's Algorithm: $O(n)=n^2$

n←number of vertices in the graph

Space Complexity Analysis:

The one-dimensional and two-dimensional arrays consume less space than linked lists. This is because linked-lists store an extra value for reference. We have converted a two-dimensional array into a one-dimensional array**.** Due to this approach, the one-dimensional array data structure consumes equivalent amount of space as two-dimensional.
The space consumption of different data structure is in the order,
Two Dimensional< One dimensional< Linked List.

Sources and References:

1. Foundations-of-algorithms-5th-richard-neapolitan_w
2. http://www.tutorialspoint.com/data_structures_algorithms/linked_lists_algorithm.htm
3. http://www.studytonight.com/data-structures/introduction-to-linked-list
4. https://github.com/cheryl06/Optimization-Algorithm
5. http://stackoverflow.com/questions/730620/how-does-a-hash-table-work
6. http://rosettacode.org/wiki/Dijkstra%27s_algorithm
7. http://www.vogella.com/tutorials/JavaAlgorithmsDijkstra/article.html