

# CS201 LAB-6 REPORT

**Name : Shobit Nair**

**Entry Number : 2019CSB1121**

## AIM

To implement the Johnsons' algorithm for all pair shortest paths ( which involves use of bellman ford and dijkstra's algorithm).Moreover , dijkstra's algorithm runtime is further analysed by using various kinds of heap data structures.

## ALGORITHM DESCRIPTION

*Johnson's algorithm* consists of the following steps:

- First, a new **node**  $q$  is added to the graph, connected by zero-weight edges to each of the other nodes.
- Second, the Bellman-Ford algorithm is used, starting from the new vertex  $q$ , to find for each vertex  $v$  the minimum weight  $h(v)$  of a path from  $q$  to  $v$ .
- If this step detects a negative cycle, the algorithm is terminated.
- Next the edges of the original graph are reweighted using the values computed by the Bellman–Ford algorithm: an edge from  $u$  to  $v$ , having length  $w(u,v)$  is given the new length  $w(u,v) + h(u) - h(v)$ .
- Finally,  $q$  is removed, and Dijkstra's Algorithm is used to find the shortest paths from each node  $s$  to every other vertex in the reweighted graph. The distance in the original graph is then computed for each distance  $D(u, v)$ , by adding  $h(v) - h(u)$  to the distance returned by Dijkstra's algorithm.

The runtime of this algorithm will be  $O(\text{runtime of bellman ford} + V \cdot (\text{runtime of Dijkstra}))$  , where  $v$  will be the number of vertices in the graph.

The runtime of this algorithm is tested by using Dijkstra's algorithm with 4 types of heaps.

- Array-based implementation
- Binary Heap
- Binomial Heap
- Fibonacci Heap

## Array

The extract-min operation takes  $O(v)$  time , while the decrease key works in constant time.

Hence , the **running time** is  $O(E \cdot (Dk) + V(Em)) \Rightarrow O(E + V^2)$

- The decrease key is done by changing the destined value in the array in  $O(1)$  time.
- Extract min takes  $O(v)$  time for the linear search of the minimum distance node in the 1-dimensional array.

## Binary , Binomial and Fibonacci Heaps

OPERATION	BINARY HEAP	BINOMIAL HEAP	FIBONACCI HEAP
insert	$O(\log N)$	$O(\log N)$	$O(1)$
find-min	$O(1)$	$O(\log N)$	$O(1)$
delete	$O(\log N)$	$O(\log N)$	$O(\log N)$
decrease-key	$O(\log N)$	$O(\log N)$	$O(1)$
union	$O(N)$	$O(\log N)$	$O(1)$

**Time complexity** =  $O(E * (\text{decrease-key}) + V * (\text{extract\_min}))$

Time complexity of Dijkstra Algorithm will take using heaps will solely depend on the number of decrease\_key , extract\_min and insert operations performed and the time complexity of individual operations are given in the table above.

### IMPLEMENTATION

- The implementation for these heaps involve insertion of all nodes except the source node with distance = infinity and the source node is pushed with distance = 0. Insertion for binomial and binary heap takes  $O(\log(v))$  time while fibonacci inserts nodes lazily in an  $O(1)$  amortized time.
- Now in every iteration the minimum node from the heap is extracted. This process takes  $O(\log(v))$  time for all types of heaps.
- Using this Min\_node , we relax the necessary neighbors using the decrease\_key operation on the already inserted nodes. The decrease key takes  $O(\log(v))$  time for binomial and binary heaps. However , fibonacci heap does it in  $O(1)$  amortised time due to which it significantly improves Dijkstra's algorithm theoretically.
- To maintain the heap property after above operations Binary heaps involve heapify operation in  $O(\log n)$  , Binary involve union and merge operation in  $O(\log n)$  and the fibonacci heap does consolidation lazily due to which the amortized cost comes out to be  $O(1)$ .
- The time complexity for Binomial and Binary heaps come out to be  $O(v \log v + E \cdot \log v)$  and the fibonacci heap runs in  $O(v \log v + E)$  which is the best theoretical time complexity among the following heap data structures.

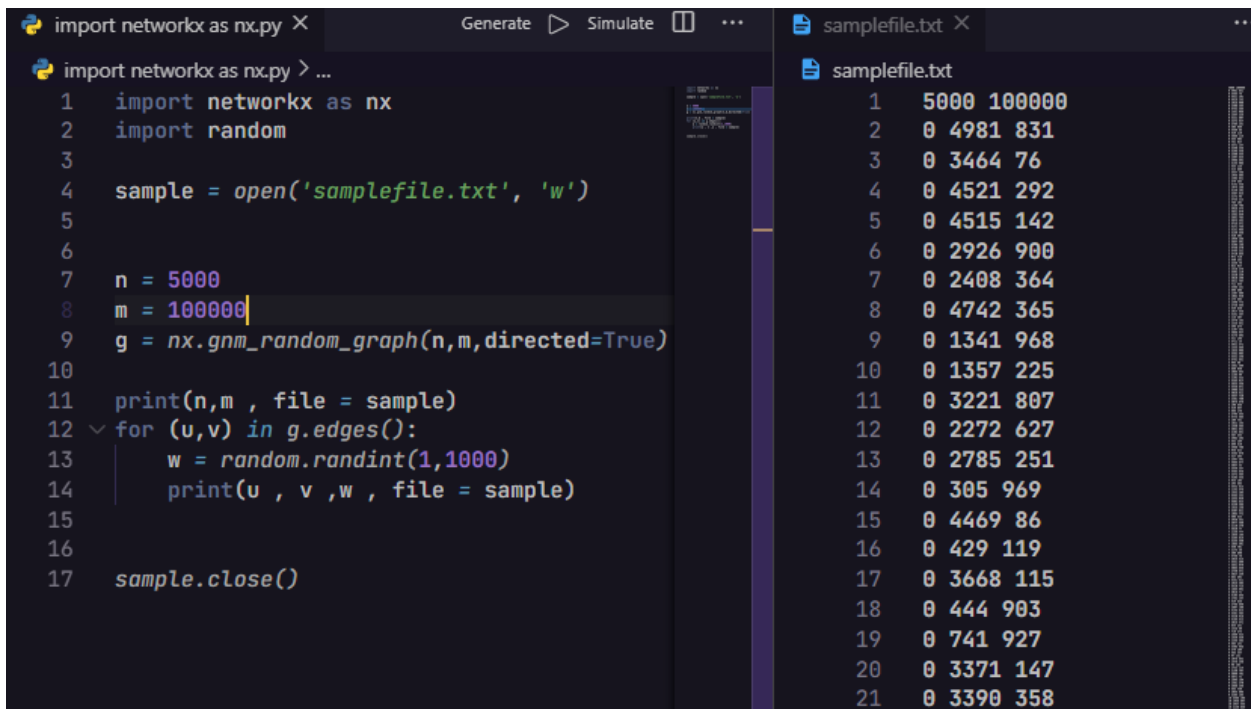
## Analysis of Dijkstra's Algorithm in various types of Heaps

The time taken during bellman ford is independent of the heap used during the process. So the total runtime will be Runtime(bellman ford) + Runtime(Dijkstra).

For large scaled graphs it was not feasible to run dijkstra's algorithm for every source vertex , since the program will take a lot of time to execute. Hence , I will run the dijkstra for around 5% nodes and take the average to find the runtime for dijkstra's algorithm for one source vertex.

For small-scaled graphs , dijkstra is run for all vertices and average of them is taken for V nodes since it was feasible to run dijkstra V times within time constraint.

Random graphs are obtained by using the Networkx library in python to generate random directed graphs for given N , M. The data is printed in (u , v , w) format to take input easily into an adjacency list containing pairs.



The screenshot shows a Python script in a Jupyter Notebook environment. The script uses the NetworkX library to generate a random directed graph with 5000 nodes and 100000 edges. The edges are written to a file named 'samplefile.txt' in an adjacency list format (u, v, w). The file is opened in 'w' mode and closed after writing.

```
import networkx as nx.py X Generate Simulate ... samplefile.txt X ...
import networkx as nx > ...
1 import networkx as nx
2 import random
3
4 sample = open('samplefile.txt', 'w')
5
6
7 n = 5000
8 m = 100000
9 g = nx.gnm_random_graph(n,m,directed=True)
10
11 print(n,m , file = sample)
12 for (u,v) in g.edges():
13     w = random.randint(1,1000)
14     print(u , v ,w , file = sample)
15
16
17 sample.close()
```

The file 'samplefile.txt' contains the following data:

u	v	w
1	5000	100000
2	0	4981 831
3	0	3464 76
4	0	4521 292
5	0	4515 142
6	0	2926 900
7	0	2408 364
8	0	4742 365
9	0	1341 968
10	0	1357 225
11	0	3221 807
12	0	2272 627
13	0	2785 251
14	0	305 969
15	0	4469 86
16	0	429 119
17	0	3668 115
18	0	444 903
19	0	741 927
20	0	3371 147
21	0	3390 358

The generated graph is taken in adjacency list format and runtime of dijkstra's algorithm is analysed for various heaps.

The graphs have been classified into small-scaled and large-scaled graph to find the trend and better understand which heap would be more reliable depending on our constraints.

### Average Runtime of Dijkstra's Algorithm in small-scaled graphs

Row headings contain the number of nodes and column heading denotes the number of edges used. The time written is in milliseconds.

The times given are AVG time taken for each Dijkstra for given Graph(N,M)

	10000	7500	5000	2500	2000	Heap - Type
150	0.2	0.2	0.2	0.2	0.2	Array
	1.3	1	0.9	0.9	0.8	Binary
	1.8	1.5	1.4	1.2	1	Binomial
	0.3	0.3	0.3	0.3	0.2	Fibonacci
250	0.4	0.4	0.4	0.4	0.4	Array
	1.7	1.5	1.4	1.2	1.2	Binary
	1.5	1.3	1.3	1.3	1.3	Binomial
	0.5	0.4	0.3	0.3	0.3	Fibonacci
500	1.5	1.4	1.4	1.4	1.4	Array
	2.6	2.5	2.3	2.1	2.1	Binary
	3	2.8	2.7	2.4	2.2	Binomial
	1.2	1.4	1.2	0.8	0.97	Fibonacci

- For small scaled graphs array method seems to perform excellently , even for greater edges. This is expected since the array takes  $O(e + v^2)$  time and  $v^2$  and  $e$  is very comparable. Eg :- 150C2 is nearly 10000.
- Whereas binary and binomial takes more time compared to fibonacci and array due to  $E \cdot \log v$  time. Eg :-  $10000 \cdot \log(150) \cdot k$  will be roughly the upper bound for binary and binomial which is greater than fibonacci and array upper bound for small scale graphs.
- For a small scaled graph , fibonacci has asymptotic time similar to array based implementation.
- Array implementation predominantly depends on the number of nodes as per the above data and there is very negligible change in runtime for change in number of edges keeping the number of nodes constant.
- For very small graphs , array works faster compared to all other methods. This may probably be due to very low constant factor.
- Since the graph is dense , the number of decrease key operations will be quite more due to which array and fibonacci gets a slight upper hand compared to binary and binomial heap.
- **Performance : Array == Fibonacci > Binary > Binomial**

### Average Runtime of Dijkstra's algorithm in a large-scaled graph.

Row headings contain the number of nodes and column heading denotes the number of edges used. The time written is in milliseconds.

The times given are AVG time taken for each Dijkstra for given Graph(N,M)

Vertices \ Edges	200000	100000	75000	50000	25000	Heap - Type
1000	7.4	6.5	6.3	6.3	6.2	Array
	10.6	10.3	9.9	9.4	7.9	Binary
	12.3	11.8	9.7	8.9	8.1	Binomial
	8.2	7.3	7.8	7.3	7	Fibonacci
2000	21.8	23.2	0.4	22.4	21.8	Array
	25.8	25.1	24.3	24.1	23.3	Binary
	27.1	26.7	26.4	26.4	25.1	Binomial
	18.4	18.8	18.1	18.3	18.1	Fibonacci
3500	74.2	72.8	70.2	70.1	69.2	Array
	48.3	45.8	44.1	43.2	41.3	Binary
	61.1	58.5	55.1	53.7	48.3	Binomial
	41.8	38.3	38.8	38.1	37.2	Fibonacci
5000	187.9	165.1	145.6	146.4	144.8	Array
	77.4	69.8	67.5	62.3	59.8	Binary
	87.1	81.3	73.4	71.1	69.8	Binomial
	61.8	60.6	58.3	58.1	56.8	Fibonacci

- For large scaled graphs , we can see that the average runtime with arrays keeps getting worse when we increase the number of nodes. Moreover , the time taken by it is also significantly high since  $V^2$  dominates E for larger scaled graphs.
- $V^2$  dominates  $E \log v$  and  $V \log v$  factors due to which array takes more time for nodes greater than 1000. Moreover the change in runtime significantly rises for array implementation as the number of nodes increases.
- Binary performs better than binomial heap in majority of the dense graphs due to the lower constant factors compared to binomial heap due to its array based implementation compared to the much more complex pointer structure binomial heap has with similar time bounds.
- The time taken for binary and fibonacci is pretty comparable for large sparse datasets. However , for large dense datasets , fibonacci heap tends to out-perform binary heap by a slight factor.
- **Performance is Fibonacci  $\geq$  Binary  $>$  Binomial  $>$  Array**

## **Conclusion**

Clearly fibonacci heaps do run better than normal binary heaps for dense graphs requiring many decrease key operations. However, the time advantage obtained does not go on par with the implementation difficulty compared to a binary heap. The reward obtained while using fibonacci heap is not very significant compared to binary.

For a number of nodes  $< 500$ , array or binary methods can be effectively used, since the constraints are low and these heaps are easy to implement.

For large-sparse graphs, binary is better since the runtime for binary was very close to fibonacci.

For large-dense graphs, fibonacci is better due to its  $O(1)$  amortized decrease key operation obtaining a slight upper-hand while relaxing nodes.