CSCE - 629

# Analysis of Algorithms

Course Project

NIRAJ GOEL
226009509

## Implementation details

The implementation was done using Python. Following are the major parts of the implementation:

1. **Random graph generation :**
   The assignment required us to generate two types of random undirected graphs, a sparse and a dense graph. In my implementation of the random graph generator, the number of edges from each vertex is calculated within a standard deviation of 2 and given mean i.e. 8 for sparse graph and 20% (around 1000) for dense graph. All the vertices are connected once in order to ensure the connectivity and then additional edges are added randomly to reach the degree average mentioned above. The degree of each vertex is calculated using the *stats.truncnorm* function of *scipy* python module. Built-in function *random.randint* is used for choosing the vertices to connect randomly. Finally weights are assigned to the graph edges randomly using the same random number generator. All the above is abstracted in a class *GraphGenerator* which also has other utility functions to get information about the graph.

   - *Class Graphgenerator:*

     - *graph* : member, this represents the graph. In my implementation it is nested dictionary with the following structure :
       - graph = *{ u1 : { v1 : w1, v2 : w2 ..}, u2 : { … } … }*

       $u_i$, $v_i$ and $w_i$ represents the source vertices, destination vertices and weight of the edge respectively. The weight can be accessed by *grap[$u_i$][$v_i$]*.

     - *Vertex* : This member contains the list of names of all the vertices. In this implementation the names are nothing but the numbers 1 to n.

     - *connectAllEdges* : This member function ensures that the graph is fully connected. As suggested in the assignment copy by the professor, It creates an initial connectivity between all the vertices.

     - *connectVertices*: This member function adds more edges to the graph as per requirement. It takes the average degree of the vertices as an argument. This function also assigns random weights to all the vertices.

**2. Dijkstra’s maximum bandwidth path algorithm**

Dijkstra’s algorithm for maximum bandwidth path as taught in class was implemented. The Algorithm starts with a given source vertex and keeps iterating till the destination vertex has been reached. In each iteration it looks at all the adjacent vertices of the current vertex and picks the vertex for which the bandwidth is maximum. The algorithm ensures that at the end we have the maximum bandwidth from source to destination. In this version of the implementation, the fringe with maximum weight is found by looping through all the fringes and looking into their weights, this operation asymptotically is *O(n)*. The overall theoretical running time of this algorithm is *$O(n^2)$.*

The implementation of this algorithm is abstracted in the class *Dijkstra* and the utility function to find the path returns the maximum bandwidth and the respective path from source to destination.

Following are the list of classes and functions with brief summary :

- *Class Dijkstra*
    - *dijkstra* : This is the member function where the actual algorithm is implemented. The function returns the maximum bandwidth and the maximum bandwidth path.

    - *maxBwFringe*: This member function parses the graph and finds the fringe with maximum bandwidth. It is used by the previous function as a helper in every iteration of the algorithm.

**3. Dijkstra’s maximum bandwidth path algorithm using heap**

There is no fundamental difference between this algorithm and the previous algorithm. However, this implementation uses a MAX HEAP data structure to maintain the fringes and select the fringe with maximum bandwidth in every iteration. Max heap is a data structure where the asymptotic complexity of accesses are *O(logn)*. Since finding the fringe with the maximum bandwidth is a significant aspect of the Dijkstra’s algorithm, the gain obtained by using this better data structure improves the overall performance of the algorithm. The overall asymptotic time achieved is *O(m + mlogn)*. The implementation uses the same class as the above except that the *maxBwFringe* subroutine now gets the fringe with maximum bandwidth from a heap.

Following are the classes and functions details for the max Heap implementation

- *Class Node:* Represents each fringe
    - *Name* and *value* are the two members (variables) of this class.

- *Class Heap:*
    - *heapList* : Member of the class. A list (array) which is used to store all the fringes.
    - *buildHeap*: This member function initially builds up the heap
    - *heapifyUp*: This member function heapifies the heap from bottom to top inorder to keep the heap property intact.
    - *heapifyDown*: This member function heapifies the heap from top to bottom inorder to keep the heap property intact
    - *maxChild*: Returns the index of child with larger value
    - *Insert*: Inserts a fringe into the heap
    - *Delete*: deletes a fringe from the heap
    - *Update*: updates the value of a fringe in the heap. This essentially uses delete and insert member functions.
    - *Maximum*: returns the fringe with maximum bandwidth

## **4. Kruskals maximum bandwidth path algorithm:**

Kruskal's algorithm for finding spanning tree is another algorithm which can be used to find the maximum bandwidth path. The algorithm first creates a maximum spanning tree of the given graph and then parses the obtained spanning tree in-order to obtain the maximum bandwidth path. The theoretical asymptotic time complexity of this algorithm is *O(mlogn).*
Following are the Classes and important methods from the implementation:

- *Class UF* : This class has two methods union( a, b) and find(a). The class implements the union find algorithm which is used to detect whether the selected edge is forming cycle or not during the Kruskals' algorithm. This version of the implementation is Union Find using path compression and rank. The important member of this class is "parent" which is essentially a dictionary keeping track of all vertices and their parent.

- *Class Edge* : The edge class represents an edge of the graph. The members are source, destination and weight.

- *Class Kruskals* : This class abstracts the implementation of kruskals' algorithm. The member function findPath() is where the actual algorithm is placed. The class has multiple helper member functions as well.

- *Class HeapSort* : This class uses the Heap class mentioned above along with a member function to return the sorted list of edges.

**5. Time Calculation:**
The time is recorded using Python's *clock()* function from the time module. This function gives the processor time i.e. the time spent by the cpu in running this particular process ( algorithm in this context ). The clock is obtained at the start of the algorithm and the end of the algorithm, the time taken by the algorithm is given by *End - Start*. The time is recorded just for the actual algorithm part and any sort of initializations are not taken into consideration.

## Performance Analysis

The three different algorithms performs differently in different scenarios. The theoretical asymptotic time complexities of the algorithm is mentioned in section 1, Following is the practical observations. The data was obtained by running the algorithms on 5 randomly generated dense graphs and 5 randomly generated sparse graphs, for each graph 5 pairs of source and destination were chosen randomly.

**1. Sparse Graph:**

- No. of vertices = 5000, Average edge degree = 8
- No of times each algorithm runs for sparse graph: 25

Following is the average time taken by each algorithm in this case:

| Algorithm | Average time (secs) | ** Updated Average time(secs) |
|---|---|---|
| Dijkstra without heap | 1.6838 | 1.6838 |
| Dijkstra with heap | 1.08736 | 0.20903 |
| Kruskals algorithm | 0.28494 | 1.764732 |

**Note on update: After submission of the printed report, I optimized my heap implementation to use indexing i.e. instead of searching for the index of the value to be deleted or updated in the heap, it does a bookkeeping of index using a hash. This significantly reduced the time complexity of the Dijkstra with heap implementation. The updated column shows the new timing.
Additionally, I discovered a bug in average time calculation of Kruskal algorithm (time of only one instance was getting divided by total instances i.e. 25), I have updated the timings in the updated column with the correct values.

***Observation:***
Here we observe that the performance of Dijkstra with heap is better than Dijkstra without heap. This is as per the theoretical performances of both the algorithms. The number of edges and hence the fringes at any point of the time would be small in a sparse graph, hence the factor "m" in complexity *O(mlogn)* as mentioned in above section would be small in the Dijkstra with heap implementation. On the other hand the O($n^2$) complexity of the Dijkstra without heap will remain unaffected by the density of the graph.

Kruskals algorithm on the other hand performs significantly better in this case. The reason being that the time complexity of Kruskal is *O(mlogn)*, Since in a sparse graph the number of edges are less i.e. m is small the algorithm runs faster. In addition to that the union find implementation has been optimized using path compression and rank finding methods which also adds to the performance here.

## <u>2. Dense Graph:</u>

- No. of vertices = 5000, Average edge degree = 1000
- No of times each algorithm runs for sparse graph: 25

Following is the average time taken by each algorithm in this case:

| Algorithm | Average time (secs) | **Updated Average time (secs) |
|---|---|---|
| Dijkstra without heap | 7.4836 | 7.4836 |
| Dijkstra with heap | 12.7036 | 5.8484 |
| Kruskals algorithm | 17.0094 | 321.7036 |

*** Refer "note on update" in above section.*

***Observation:***
Here we observe that the performance of Dijkstra's algorithm without heap is better than the Dijkstra without heap. Since, the graph is extremely dense, the number of fringes at a particular time would be large and the factor m in the time complexity of *$O(m + mlogn)$* would be large i.e. tending to $n^2$.

The performance of Kruskals algorithm on dense graph was observed to be poor than that of Dijkstra's. Since the number of edges is extremely large, the *$m$* in its bound of *$O(mlogn)$* tends to go near to *$n^2$*. Additionally, parsing a dense maximum spanning tree later *($O(V + E)$)* to find the maximum bandwidth path also adds to the time taken as *$E$* becomes very large. Another reason for Kruskals performance being low in comparison to Dijkstra with heap (theoretical complexities being of same order) can be attributed to the size of heap getting created. In a dense graph the MST will consist of all the edges which is very large i.e. the heap size in case of kruskals while for Dijkstra the heap size would be maximum number of fringes at any point of time, which would be relatively lesser than that in Kruskals.

## Further Improvements

- Use of Fibonacci heap in the Dijkstra's algorithm : The 2nd implementation of Dijkstra's algorithm currently uses a regular maximum heap. As discussed in the class and suggested in the textbook as well, Fibonacci heaps can be used to improve the performance even more. The complexity of *$O(VlogV + E)$* is achievable with using Fibonacci heap.
- In Kruskal's algorithm, after maximum spanning tree is generated, I have done a BFS in-order to find the maximum bandwidth and the maximum bandwidth path. This step can be avoided by book-keeping of the bandwidth paths during the generation of MST itself. Thus, the additional *$O(V + E)$* complexity of BFS can be saved.
- Trying out the Part 6 of the Assignment i.e. the linear time algorithm for finding maximum bandwidth path.

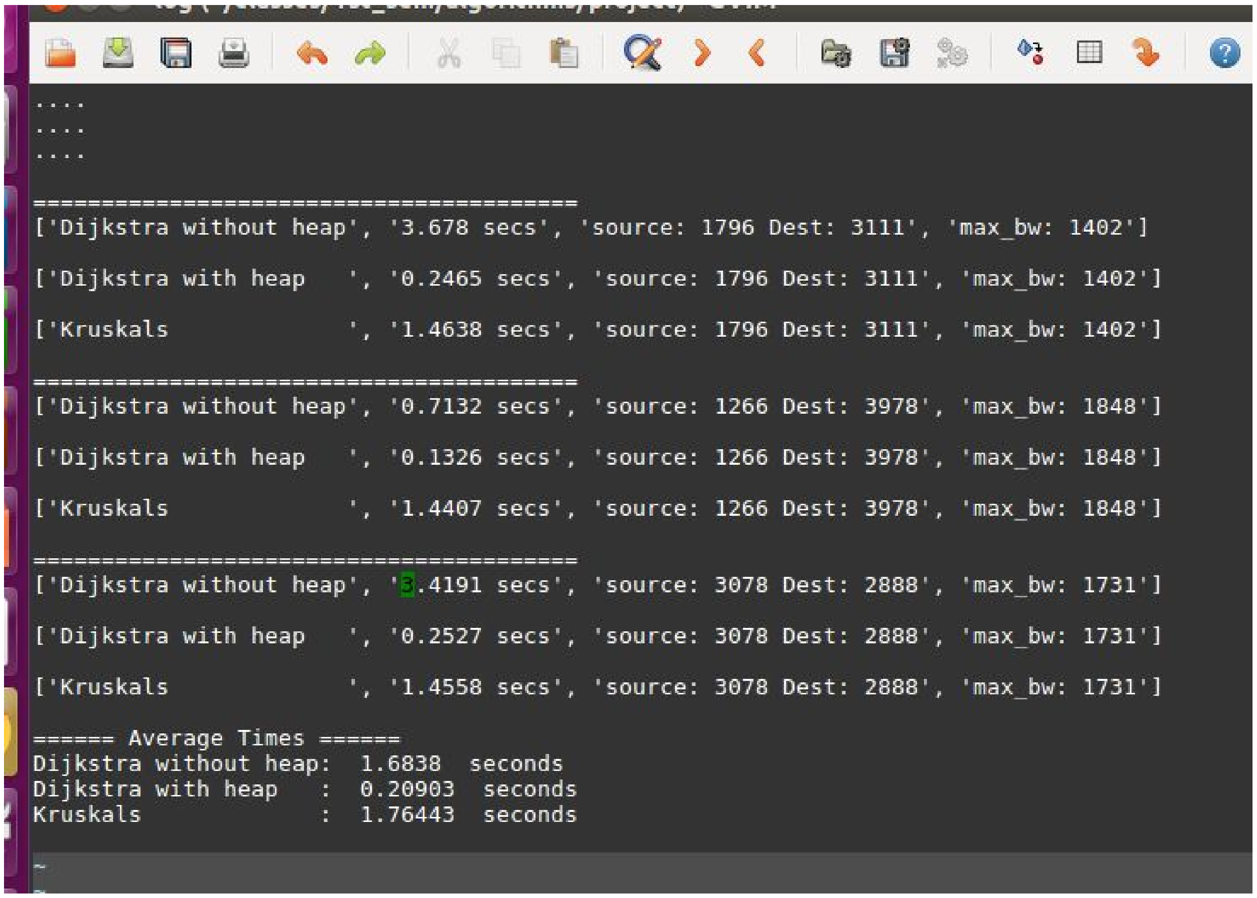
## References

- Class Notes
- CLRS Textbook
- http://www.geeksforgeeks.org/heap-sort/

## Appendix

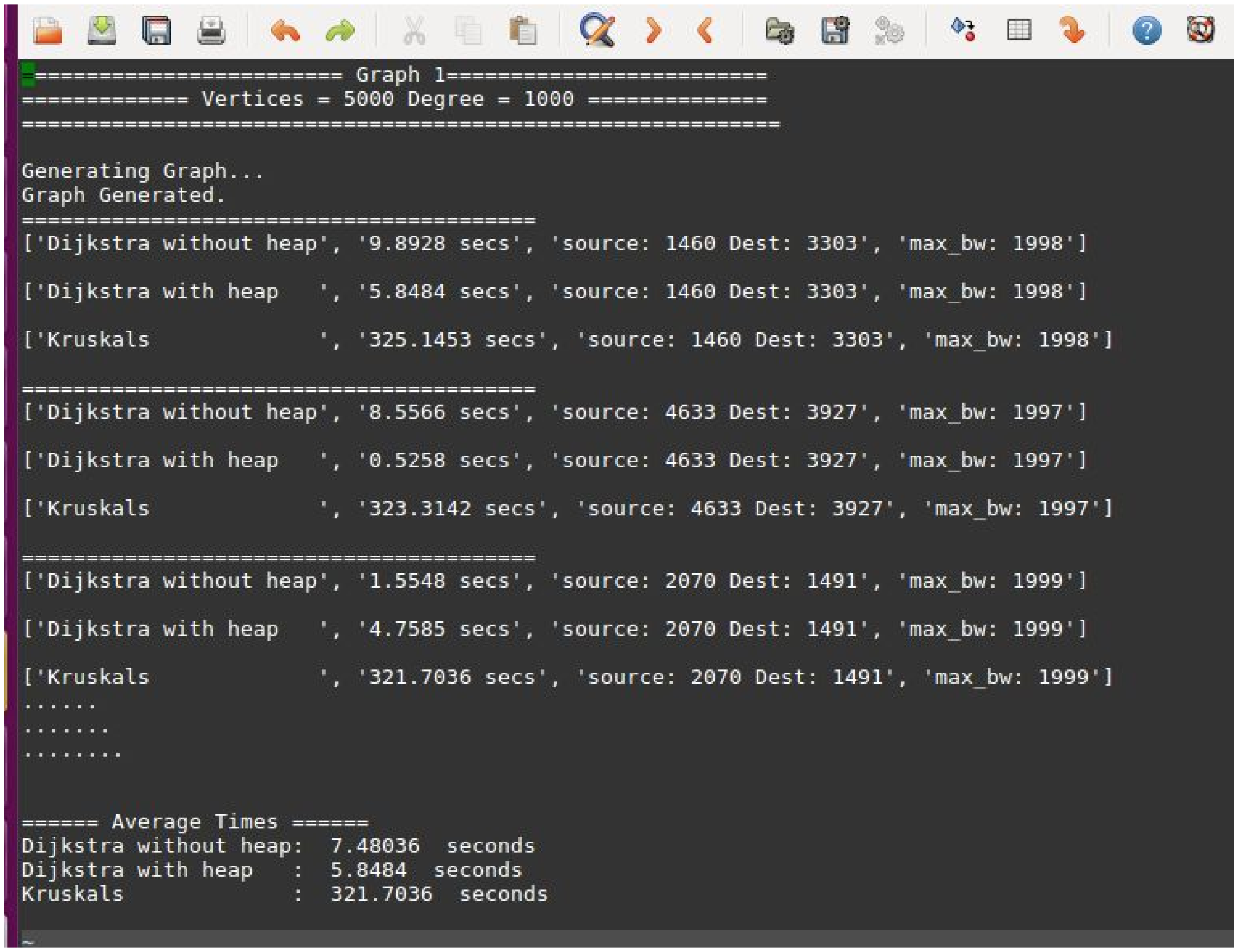
Note: This section was also updated with the new snapshots as per the updated columns above.

1. Sparse graph output snapshot



```
....
....
....

========================================
['Dijkstra without heap', '3.678 secs', 'source: 1796 Dest: 3111', 'max_bw: 1402']

['Dijkstra with heap   ', '0.2465 secs', 'source: 1796 Dest: 3111', 'max_bw: 1402']

['Kruskals             ', '1.4638 secs', 'source: 1796 Dest: 3111', 'max_bw: 1402']

========================================
['Dijkstra without heap', '0.7132 secs', 'source: 1266 Dest: 3978', 'max_bw: 1848']

['Dijkstra with heap   ', '0.1326 secs', 'source: 1266 Dest: 3978', 'max_bw: 1848']

['Kruskals             ', '1.4407 secs', 'source: 1266 Dest: 3978', 'max_bw: 1848']

========================================
['Dijkstra without heap', '3.4191 secs', 'source: 3078 Dest: 2888', 'max_bw: 1731']

['Dijkstra with heap   ', '0.2527 secs', 'source: 3078 Dest: 2888', 'max_bw: 1731']

['Kruskals             ', '1.4558 secs', 'source: 3078 Dest: 2888', 'max_bw: 1731']

====== Average Times ======
Dijkstra without heap:  1.6838   seconds
Dijkstra with heap   :  0.20903  seconds
Kruskals             :  1.76443  seconds

~
~
```

2. Dense graph output snapshot



```
======================== Graph 1========================
============= Vertices = 5000 Degree = 1000 ==============
===========================================================

Generating Graph...
Graph Generated.
========================================
['Dijkstra without heap', '9.8928 secs', 'source: 1460 Dest: 3303', 'max_bw: 1998']

['Dijkstra with heap   ', '5.8484 secs', 'source: 1460 Dest: 3303', 'max_bw: 1998']

['Kruskals             ', '325.1453 secs', 'source: 1460 Dest: 3303', 'max_bw: 1998']

========================================
['Dijkstra without heap', '8.5566 secs', 'source: 4633 Dest: 3927', 'max_bw: 1997']

['Dijkstra with heap   ', '0.5258 secs', 'source: 4633 Dest: 3927', 'max_bw: 1997']

['Kruskals             ', '323.3142 secs', 'source: 4633 Dest: 3927', 'max_bw: 1997']

========================================
['Dijkstra without heap', '1.5548 secs', 'source: 2070 Dest: 1491', 'max_bw: 1999']

['Dijkstra with heap   ', '4.7585 secs', 'source: 2070 Dest: 1491', 'max_bw: 1999']

['Kruskals             ', '321.7036 secs', 'source: 2070 Dest: 1491', 'max_bw: 1999']
......
.......
........


====== Average Times ======
Dijkstra without heap:  7.48036  seconds
Dijkstra with heap   :  5.8484  seconds
Kruskals             :  321.7036  seconds
```