

**SANKALP SHIRISH CHAPALGAONKAR**

UIN: 233005696

PROJECT REPORT

ANALYSIS OF ALGORITHMS

CSCE 629

FALL 2022

CSCE 629 Project Report

* Introduction

A Maximum Bandwidth pass problem deals with finding a path between two vertices such that the path contains the maximum bandwidth. This project is implemented using Python and includes the following main functions:

A random graph generation which takes degree of each vertex and number of nodes as input and returns a fully connected random undirected weighted graph.

A Heap Structure responsible for heapsort and also to store the fringe nodes in the case of Dijkstra with Heap implementation.

A Union and Find algorithm which is used in the generation of Maximum Spanning Tree generation

Depth First Search Algorithm to nd the path from one vertex to another in the maximum spanning tree.

Three Routing Algorithm

{ Dijkstra’s without Heap Implementation

{ Dijkstra’s with Heap Implementation

{ Kruskal’s Algorithm

The report is organized as follows; Section 2 involves with Implementation Details, Section 3 deals with Performance Analysis of the algorithms and Section 4 explains about the Conclusion for this project report.

* Implementation

2.1 Random Graph Generation

A random graph generator function is written which takes the number of vertices, degree of each vertex and seed for the random generator as input. This function returns an object of the Graph class. Here the prede ned graph class is being used which is part of the networkx library in python.

UIN: 233005696 Page 1

CSCE 629 Project Report

Firstly a full circle is created with edges connecting all the nodes in a circle with random weights being assigned to each edge. An adjacency list is being maintained to keep track of the edges between the vertices. Next, for each vertex until the degree of the vertex reaches the required degree, edges are being added with random weights and are connected to random vertices. The range of weights is from 1 to 5000 to avoid repetition of edge weights. When the degree being passed to the generator is 1000, it is equivalent to the generation of the graph where each vertex is adjacent to about 20% of the other vertices.

2.2 Heap

The heap being implemented for this project is MaxHeap. The heap structure is used twice, one in Dijkstra with Heap Implementation to store the fringe nodes and in Kruskal’s Algorithm to HeapSort the edges for the generation of the Maximum Spanning Tree. In the Heap implementation, Maximum, Insert, Delete, Swap, Siftup, SiftDown and ReplaceKey subroutines are de ned. For the Dijkstra algorithm usage, ReplaceKey function is very useful to update the bandwidth. This function basically updates the bandwidth of that vertex and puts the vertex in the correct position as per its bandwidth.

2.3 Dijkstra Algorithm

Dijkstra algorithm uses greedy method to nd the maximum bandwidth pass provided all the weights are positive. In this project two implementation of Dijkstra are done i.e., on using Heap implementation to store fringe node and other without using Heap. The main algorithm in both the cases remains the same but only the running time di ers. For the heap implementation of Dijkstra we use the heap function de ned before with Maximum, Delete and ReplaceKey methods being extensively used. The heap is used to store the fringe vertex names and the maxheap is implemented based on the bandwidth of the fringe vertices. The root would then be the fringe vertex with the highest bandwidth. For the implementation of Dijkstra without Heap we run a while loop until all the status of the vertices turns to intree. The time complexity of Dijkstra Algorithm without Heap is more than that of Dijkstra Algorithm with Heap.

UIN: 233005696 Page 2

CSCE 629 Project Report

2.4 Kruskal Algorithm

For the implementation of Kruskal Algorithm a UnionFind class is being de ned. This class helps in path compression. The nd method is used to nd the root of the vertex and union is used to combine various disjoint trees into 1 maximum spanning tree. For the sorting of the edges HeapSort is used. Once a MST is generated, using Depth First Search we nd the path from the given source to the destination.

* Performance Analysis

3.1 Sample Graph

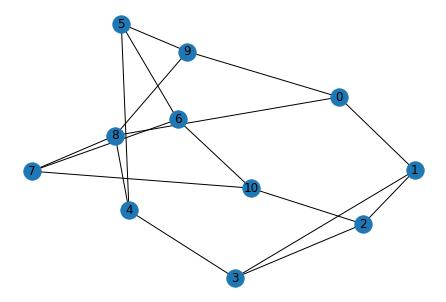


Figure 1: Random Graph with 10 vertices and degree 3

The Fig.1 represents a random graph generated with 10 nodes with degree of each node being 3. This is just of representation showing that the graph is being generated as expected.

3.2 Sample Output

* S o u r c e : 600

2 D e s t i n a t i o n : 2410

UIN: 233005696 Page 3

CSCE 629 Project Report

* Path w i t h o u t Heap :

|  |  |  |
| --- | --- | --- |
| 4 | [2410 , 3118 , 411 , 4701 , 2378 , 653 , 2135 , 2822 , 3300 , 1000 , 2974 , 1316 , 204 , 140 , 1869 , | |
|  | 1602 , 1829 , 3858 , 2471 , | 469 , 4986 , 4204 , 285 , 1056 , 489 , 1656 , 3651 , 2832 , 2876 , 1323 , |
|  | 4300 , 4965 , 2150 , 600] |  |
| 5 | Maximum Bandwidth D i j k s t r a : Without Heap 4989 | |
| 6 | Path with Heap : |  |
| 7 | [2410 , 3118 , 411 , 4745 , 35 , 3209 , 3712 , 2507 , 1267 , 4143 , 545 , 389 , 2475 , 2693 , 3648 , | |
|  | 3550 , 4354 , 3608 , 690 , 3496 , 3858 , 2471 , 469 , 4986 , 4204 , 285 , 1056 , 489 , 1656 , 3651 , | |
|  | 2832 , 2876 , 1323 , 4300 , | 4965 , 2150 , 600] |
| 8 | Maximum Bandwidth D i j k s t r a : | Heap 4989 |

* Kruskal Path :

10 [2410 , 3118 , 411 , 4701 , 2378 , 653 , 2135 , 2822 , 3300 , 1000 , 2974 , 1316 , 204 , 140 , 1869 ,

|  |  |  |  |
| --- | --- | --- | --- |
| 1602 , | 1829 , | 3858 , | 2471 , 469 , 4986 , 4204 , 285 , 1056 , 489 , 1656 , 3651 , 2832 , 2876 , 1323 , |
| 4300 , | 4965 , | 2150 , | 600] |

11 Maximum Bandwidth Kruskal : 4989

1. Timing A n a l y s i s
2. Graph G e n e r a t i o n : 6 0 . 5 2 3 4 3 0 3 4 7 4 4 2 6 3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 14 | Time | f o r | D i j k s t r a | Without Heap : 1 1 . | 5385103225708 |
| 15 | Time | f o r | D i j k s t r a | With Heap : 8 . 4 4 7 5 | 73900222778 |

1. T o t a l Time Kruskal : 1 6 . 6 0 5 8 1 0 6 4 2 2 4 2 4 3

This is a sample output giving Path from destination to source for all 3 algorithms along with the maximum bandwidth. We can observe that the maximum bandwidth returned by all 3 algorithms is the same but the maximum bandwidth pass di ers for each algorithm. The output also has the running time of each algorithm being displayed.

3.3 Timing Analysis for Sparse Graph

3.3.1 Graph Generation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Graph | Graph1 | Graph2 | Graph3 | Graph4 | Graph5 |
|  |  |  |  |  |  |
| Duration | 0.090954304 | 0.105186224 | 0.097897053 | 0.096430302 | 0.164721966 |
|  |  |  |  |  |  |

UIN: 233005696 Page 4

CSCE 629 Project Report

3.3.2 Dijkstra without Heap

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Duration |  | Pair1 |  | Pair2 |  | Pair3 |  | Pair4 |  | Pair5 | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph1 | 3.933715105 | | 4.233586311 | | 4.1036129 | | 4.259529591 | | 4.113918066 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph2 | 4.810223103 | | 4.12529707 | | 4.211987019 | | 4.224545956 | | 4.291259289 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph3 | 4.284241915 | | 4.184536695 | | 4.249397993 | | 4.199856043 | | 3.965998411 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph4 | 4.029652119 | | 4.242292881 | | 4.183971405 | | 4.038101435 | | 4.398355961 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph5 | 4.069397449 | | 4.224159241 | | 4.128736973 | | 4.198780298 | | 4.319262266 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3.3.3 Dijkstra With Heap | | | | |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Duration |  | Pair1 |  | Pair2 |  | Pair3 |  | Pair4 |  | Pair5 | |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph1 | 0.243809223 | | 0.268352985 | | 0.265566111 | | 0.26983285 | | 0.2698946 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph2 | 0.319811106 | | 0.258093357 | | 0.266513586 | | 0.270230293 | | 0.270822287 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph3 | 0.310579538 | | 0.2722826 | | 0.253706217 | | 0.270066261 | | 0.226960897 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph4 | 0.267971992 | | 0.281337023 | | 0.275892258 | | 0.270857811 | | 0.325409174 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph5 | 0.273793697 | | 0.288200617 | | 0.267091036 | | 0.274477482 | | 0.26757288 | |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3.3.4 Kruskal Algorithm | | | | |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Graph |  | Graph1 |  | Graph2 |  | Graph3 |  | Graph4 |  | Graph5 |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  | Duration |  | 0.125549126 |  | 0.123049593 |  | 0.119797039 |  | 0.130824137 |  | 0.12173295 |  |
|  |  |  |  |  |  |  | |  |  |  |  |  |  |
|  |  |  |  |  |  | MST Build Time | | |  |  |  |  |  |

UIN: 233005696 Page 5

CSCE 629 Project Report

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
|  |  |  |  |  |  |
| Graph1 | 0.025768757 | 0.059503078 | 0.080115795 | 0.06618309 | 0.094055176 |
|  |  |  |  |  |  |
| Graph2 | 0.097825289 | 0.058978558 | 0.0885396 | 0.081089973 | 0.151168108 |
|  |  |  |  |  |  |
| Graph3 | 0.093871117 | 0.030666113 | 0.080920696 | 0.088834524 | 0.077366114 |
|  |  |  |  |  |  |
| Graph4 | 0.057703972 | 0.076043844 | 0.074041605 | 0.096882582 | 0.11514473 |
|  |  |  |  |  |  |
| Graph5 | 0.050774813 | 0.108473301 | 0.046176672 | 0.045380116 | 0.101308823 |
|  |  |  |  |  |  |
|  |  | DFS Path Time | |  |  |
|  |  |  |  |  |  |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
|  |  |  |  |  |  |
| Graph1 | 0.180866003 | 0.156195402 | 0.239415646 | 0.161786079 | 0.247546196 |
|  |  |  |  |  |  |
| Graph2 | 0.278862715 | 0.150232792 | 0.180722237 | 0.235738993 | 0.246048927 |
|  |  |  |  |  |  |
| Graph3 | 0.200816631 | 0.194551945 | 0.177006721 | 0.243169785 | 0.171486616 |
|  |  |  |  |  |  |
| Graph4 | 0.149168491 | 0.242821217 | 0.165160418 | 0.254041672 | 0.224181414 |
|  |  |  |  |  |  |
| Graph5 | 0.143975735 | 0.205111504 | 0.206474543 | 0.13775444 | 0.258444548 |
|  |  |  |  |  |  |

Total Time

3.4 Timing Analysis for Dense Graph

3.4.1 Graph Generation

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Graph | Graph1 | Graph2 | Graph3 | Graph4 | Graph5 |
|  |  |  |  |  |  |
| Duration | 58.98736835 | 60.90540886 | 59.47075152 | 58.49494171 | 60.234025 |
|  |  |  |  |  |  |

UIN: 233005696 Page 6

CSCE 629 Project Report

3.4.2 Dijkstra Without Heap

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
|  |  |  |  |  |  |  |
|  | Graph1 | 11.2545j867 | 10.45073462 | 11.01199841 | 12.51237202 | 10.5702095 |
|  |  |  |  |  |  |  |
|  | Graph2 | 11.15364408 | 11.06088352 | 11.17029309 | 11.38336825 | 11.29249835 |
|  |  |  |  |  |  |  |
|  | Graph3 | 11.05792451 | 11.75624418 | 12.04945111 | 11.56264257 | 12.54632187 |
|  |  |  |  |  |  |  |
|  | Graph4 | 12.58334613 | 11.48749709 | 10.75620127 | 11.10832119 | 11.39596725 |
|  |  |  |  |  |  |  |
|  | Graph5 | 11.31051612 | 11.51912165 | 11.55561805 | 12.05622864 | 12.25940847 |
|  |  |  |  |  |  |  |
| 3.4.3 Dijkstra With Heap | | |  |  |  |  |
|  |  |  |  |  |  |  |
|  | Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
|  |  |  |  |  |  |  |
|  | Graph1 | 8.054227829 | 7.950505733 | 8.114288568 | 8.534822464 | 9.047904253 |
|  |  |  |  |  |  |  |
|  | Graph2 | 8.10699749 | 8.219291449 | 8.423859358 | 8.308233261 | 8.176138639 |
|  |  |  |  |  |  |  |
|  | Graph3 | 8.033660173 | 8.233247757 | 8.329352856 | 8.515664101 | 8.026985645 |
|  |  |  |  |  |  |  |
|  | Graph4 | 8.069142818 | 8.356496334 | 8.370652676 | 8.274383545 | 8.225543976 |
|  |  |  |  |  |  |  |
|  | Graph5 | 8.277152538 | 7.964969158 | 8.432705879 | 8.372105837 | 8.449433804 |
|  |  |  |  |  |  |  |
| 3.4.4 Kruskal Algorithm | | |  |  |  |  |
|  |  |  |  |  |  |  |
|  | Graph | Graph1 | Graph2 | Graph3 | Graph4 | Graph5 |
|  |  |  |  |  |  |  |
|  | Duration | 17.43509665 | 17.55303402 | 17.77852368 | 16.74237704 | 16.17882357 |
|  |  |  |  |  |  |  |
|  |  |  | MST Build Time | |  |  |

UIN: 233005696 Page 7

CSCE 629 Project Report

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
|  |  |  |  |  |  |
| Graph1 | 0.054023504 | 0.151445866 | 0.097405434 | 0.052339077 | 0.072049856 |
|  |  |  |  |  |  |
| Graph2 | 0.101567507 | 0.08053565 | 0.059229851 | 0.11239624 | 0.107078314 |
|  |  |  |  |  |  |
| Graph3 | 0.105113745 | 0.137387991 | 0.053827763 | 0.052511454 | 0.081958294 |
|  |  |  |  |  |  |
| Graph4 | 0.067979336 | 0.073587179 | 0.107208252 | 0.042102814 | 0.079879761 |
|  |  |  |  |  |  |
| Graph5 | 0.092291594 | 0.056953192 | 0.079370975 | 0.089389086 | 0.06570816 |
|  |  |  |  |  |  |
|  |  | DFS Path Time | |  |  |
|  |  |  |  |  |  |
| Duration | Pair1 | Pair2 | Pair3 | Pair4 | Pair5 |
|  |  |  |  |  |  |
| Graph1 | 22.47302389 | 21.13967872 | 22.54795623 | 18.17994547 | 17.23893452 |
|  |  |  |  |  |  |
| Graph2 | 17.40424252 | 15.39398742 | 17.96434426 | 17.39454412 | 15.4269433 |
|  |  |  |  |  |  |
| Graph3 | 16.23631096 | 17.26502752 | 14.41450381 | 16.89396977 | 16.5409832 |
|  |  |  |  |  |  |
| Graph4 | 15.99625611 | 17.09899616 | 18.15328217 | 16.89459133 | 16.9670155 |
|  |  |  |  |  |  |
| Graph5 | 15.48662543 | 17.36739016 | 16.20957422 | 14.69757318 | 15.12691569 |
|  |  |  |  |  |  |

Total Time

3.5 Discussion on the Outputs

3.5.1 Theoritical Time Compelexity

|  |  |
| --- | --- |
| Algorithm | Time Complexity |
|  |  |
| Dijkstra Without Heap | O(n2) |
|  |  |
| Dijkstra with Heap | O((n + m)logn) |
|  |  |
| Maximum Spanning Tree Build Time | O(mlogn) |
|  |  |
| Path Find Time | O(m) |
|  |  |

UIN: 233005696 Page 8

CSCE 629 Project Report

3.5.2 Dijkstra’s Algorithm

From the theoritical Complexities, we can observe that Dijkstra Implementation with Heap is faster than that without Heap and this can also be seen in the output table. This is because insertion and deletion in Heap takes O(logn) time. In non-heap implementation, it takes O(n) time to nd the fringe vertex with the maximum bandwidth. Therefore, the performance of Dijkstra with heap is better than Dijkstra without heap. This results holds true for both sparse as well as dense random graphs. We can observe that the trend remains the same as the number of nodes increases the running time increases. This is because we have more number of edges to be examined now.

3.5.3 Kruskal’s Algorithm

Maximum Spanning Tree build time is O(mlogn) as per the theory. Since building of an MST involves many operations like to nd the edges in decreasing order and creating an tree, we can see that the time to build a MST is larger. And, the time to nd the source to destination path is very less than the build time as it just involves going through the parent array got from DFS which takes linear time. We can observe than Kruskal is faster than Dijkstra with Heap for graph with 6 degree. And, for the dense graph, Dijkstra with Heap is faster than that of Kruskal Algorithm. The reason behind this might be that since more number of edges are to be inserted in the heap and are to be sorted each time in the case of Kruskal Algorithm.

* Conclusion

From the outputs, we can observe that although the theoretical time complexity are the same, the running time for each algorithm di ers for each case. This is because the graphs are random in nature and each graph wont let the algorithm reach its worst case scenario which is being represented by the theoretical complexities. Dijkstra performs better when we need to nd a single path between two vertices. Kruskal algorithm gives us an advantage of generating a given MST once and determining the path based on that MST in linear time.

UIN: 233005696 Page 9