**CSCE-629 Analysis of Algorithms**

**Network Routing Protocol**

**Project Report**

Dongwei Qi

230002618

**Part1. Project Description**

In graph algorithms, a MBP (Maximum Bandwidth Path) problem is defined as the problem of finding a path between two designated vertices in a weighted graph, maximizing the weight of the minimum-weight edge in the path.

To implement the theoretical knowledge into a real-world practical computer program, this project fulfills the data structures and algorithms listed below:

Random-generated weighted undirected graph

Max heap structure with subroutines for heapsort, insert and delete

UFS(Union Find Set) algorithm for generation of MST(Maximum Spanning Tree)

DFS(Depth First Search) algorithm

Modified Dijkstra’s algorithm without heap structure

Modified Dijkstra’s algorithm using heap structure

Modified Kruskal’s algorithm using heapsort

**Part2. Implementation Details**

**Experiment Environment**

|  |  |
| --- | --- |
| Software: | Visual Studio Code 1.39.2 |
| Language: | Python 3.7.4 64-bit |
| System: | macOS Mojave 10.14.6 |
| Processor: | 1.4 GHz Intel Core i5 |
| Memory: | 8 GB 2133 MHz LPDDR3 |
| Graphics: | Intel Iris Plus Graphics 645 1536 MB |

**Graph Generation**

For graph generation, this project fulfills a graph class and returns a graph object, with subroutines for seeking all neighbors of a vertice, get weight between two vertices, get all edges, etc. To avoid using given libraries like networkx, the graph uses a dictionary as a storage to maintain all weighted edges. To make sure the graph is connected, the graph builds with a cycle first and gradually adds edges to it. In order to make sure the edges are generated randomly enough, the graph will add edges for each vertex until it reaches a certain degree. The weight range of edges is set between 0~1k to make sure the weight is big enough to avoid repetition. When the degree of vertices set as 1000, it is equivalent in terms of building a graph where each vertex is adjacent to about 20% of the other vertices.

**Max Heap**

The project implemented a max heap structure as components of Dijkstra's algorithm and heapsort subroutine for Kruskal’s algorithm.

In Dijkstra's algorithm, the max heap is used as a storage of fringe nodes for the updates of the current best bandwidth.

Kruskal’s Algorithm uses heapsort to sort the weighted edges for the generation of the Maximum Spanning Tree.

**Modified Dijkstra’s Algorithm without using Heaps**

1. For each v do

2. status[v]=unseen

3. For each edge [src,w] do

4. status[w]=fringe

5. dad[w]=src

6. width[w]=weight[src,w]

7. While there are fringes do:

8. pick a fringe with the largest width

9. status[v]=intree

10. for each edge [v,w] do

11. if(status[w]=unseen) then

12. status[w]=fringe

13. dad[w]=v

14. width[w]=min{width[v],weight[v,w]}

15. elif (status[w]=fringe and width[w]< min{width[v],weight[v,w]})

16. dad[w]=v;

17. width[w]= min{width[v],weight[v,w]}

18. return width[dst]

Modified Dijkstra's algorithm without heap runs in O(n2) time. Pick the best fringe v runs in O(n) time. Traversing the neighbors of the vertex and then calculating the max capacity takes O(n) in Modified Dijkstra's algorithm without heap, while it takes O(logn) in Modified Dijkstra’s algorithm using heap structure.

**Modified Dijkstra’s Algorithm using Heaps**

For the Modified Dijkstra’s Algorithm using Heaps, the difference is that step 8 is fulfilled using max heap structure, thus the time complexity is O(mlogn).

**Modified Kruskal Algorithm using Heap**

1. Sort the edges in non increasing order using heapsort

2. For each v do

3. dad[v]=v; rank[v]=0

4. For i=1 to m do

5. Let ei=[vi,wi];

6. r1=find(vi)

7. r2=find(wi)

8. if(r1!=r2):

9. ei is added to mst

10. Union(r1,r2)

11. return the s-t path in mst

Find(v)

1. while dad[v]!=v do:

2. dad[w]=Find(dad[w])

3. return dad[v]

Union(r1,r2)

1. if(rank[r1]>rank[r2])

2. dad[r2]=r1

3. elif(rank[r1]<rank[r2])

3. dad[r1]=r2;

4. elif(rank[r1]==rank[r2])

5. dad[r2]=r1

6. rank[r1]++

This algorithm takes O(mlogn) time.

Theoretically, Dijkstra’s algorithm without heap takes more time compared to Dijkstra’s algorithm with heap and Kruskal’s algorithm.

**Part3. Experiment Results**

**Snapshot example:**

****

For the testing of the project, 5 pairs of graphs are generated and each was test by 5 pairs of random source and destination vertice. All the testing records are maintained in the tables below.

For sparse graph:

|  |  |  |  |
| --- | --- | --- | --- |
| Sparse | #vertices | #edges | avg\_degree |
| G1 | 5000 | 17816 | 7.1264 |
| G2 | 5000 | 17845 | 7.138 |
| G3 | 5000 | 17857 | 7.1428 |
| G4 | 5000 | 17849 | 7.1396 |
| G5 | 5000 | 17884 | 7.1536 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G1 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 586 | 1229 | 0.56890273 | 0.278632879 | 0.39680505 |
| Round2 | 294 | 49 | 0.28299809 | 0.347150087 | 0.36382699 |
| Round3 | 1360 | 57 | 0.73659492 | 0.301295996 | 0.47113776 |
| Round4 | 4272 | 3952 | 0.62801123 | 0.401356936 | 0.38699102 |
| Round5 | 982 | 1411 | 0.64082789 | 0.380090237 | 0.39571905 |
| avg\_time |  |  | 0.57146697 | 0.341705227 | 0.40289598 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G2 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 4188 | 276 | 0.24287391 | 0.079884052 | 0.37731886 |
| Round2 | 3369 | 1102 | 0.26848578 | 0.135657072 | 0.34605122 |
| Round3 | 3964 | 2137 | 0.67060399 | 0.343032122 | 0.36418509 |
| Round4 | 3648 | 207 | 0.08128309 | 0.055934906 | 0.3414979 |
| Round5 | 2716 | 4692 | 0.83962297 | 0.199316978 | 0.38832426 |
| avg\_time |  |  | 0.42057395 | 0.162765026 | 0.36347547 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G3 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 3858 | 3156 | 0.11287093162536621 | 0.09406900405883789 | 0.42124104 |
| Round2 | 1552 | 1118 | 0.23722600936889648 | 0.18684101104736328 | 0.39745998 |
| Round3 | 4357 | 4987 | 1.5460140705108643 | 0.43166303634643555 | 0.40153503 |
| Round4 | 2073 | 1519 | 0.1499619483947754 | 0.05182504653930664 | 0.39348602 |
| Round5 | 2700 | 1479 | 1.3822247982025146 | 0.405647755 | 0.40700793 |
| avg\_time |  |  | 0.42057395 | 0.197462559 | 0.38196207 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G4 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 1359 | 4277 | 0.588413 | 0.17408990859985352 | 0.3745811 |
| Round2 | 4087 | 3989 | 0.1854243278503418 | 0.3823411464691162 | 0.40923214 |
| Round3 | 2071 | 4654 | 1.1898810863494873 | 0.1254439353942871 | 0.37521791 |
| Round4 | 4996 | 1997 | 0.8218021392822266 | 0.110689878 | 0.39970899 |
| Round5 | 2027 | 1438 | 0.03284597396850586 | 0.024519920349121094 | 0.40258574 |
| avg\_time |  |  | 0.588413 | 0.110689878 | 0.39226518 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G5 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 3583 | 937 | 0.13356304168701172 | 0.023155212 | 0.39332795 |
| Round2 | 3836 | 3719 | 1.5884060859680176 | 0.4721949100494385 | 0.42104602 |
| Round3 | 550 | 688 | 1.465602159500122 | 0.437347174 | 0.40068984 |
| Round4 | 151 | 1225 | 0.78125072 | 0.469072104 | 0.42035413 |
| Round5 | 4595 | 3727 | 0.5306100845336914 | 0.338190794 | 0.38414693 |
| avg\_time |  |  | 0.78125072 | 0.316941321 | 0.40391297 |

For dense graph:

|  |  |  |  |
| --- | --- | --- | --- |
| Sparse | #vertices | #edges | avg\_degree |
| G1 | 5000 | 3088903 | 1235.5612 |
| G2 | 5000 | 3088561 | 1235.4244 |
| G3 | 5000 | 3089675 | 1235.87 |
| G4 | 5000 | 3088798 | 1235.5192 |
| G5 | 5000 | 3089416 | 1235.7664 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G1 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 2679 | 4865 | 4.68225288 | 1.40020895 | 112.589328 |
| Round2 | 1969 | 4840 | 0.27909493 | 4.72043705 | 114.239219 |
| Round3 | 3016 | 4714 | 0.47347307 | 5.334584 | 114.063201 |
| Round4 | 2616 | 3364 | 2.20565915 | 3.78967237 | 111.678135 |
| Round5 | 3136 | 67 | 1.65641284 | 4.38189888 | 112.544042 |
| avg\_time |  |  | 1.85937858 | 3.92536025 | 113.022785 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G2 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 2192 | 2946 | 0.38618803 | 2.15917206 | 105.609204 |
| Round2 | 2531 | 4657 | 4.79877901 | 5.18606997 | 113.792888 |
| Round3 | 3823 | 324 | 2.8475101 | 4.47613001 | 119.668766 |
| Round4 | 2466 | 4891 | 4.38554478 | 6.40479779 | 113.409705 |
| Round5 | 3073 | 3854 | 0.23593497 | 5.01296473 | 112.117572 |
| avg\_time |  |  | 2.53079138 | 4.64782691 | 112.919627 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G3 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 4785 | 3306 | 1.38714814 | 4.4269979 | 110.24452 |
| Round2 | 3279 | 3863 | 1.85592985 | 3.44314122 | 117.79631 |
| Round3 | 3034 | 4179 | 2.98675299 | 6.17442203 | 115.365446 |
| Round4 | 3580 | 371 | 3.69705701 | 3.54680991 | 114.675608 |
| Round5 | 1146 | 4397 | 3.13789296 | 5.21577215 | 118.461519 |
| avg\_time |  |  | 2.61295619 | 4.56142864 | 115.308681 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G4 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 3749 | 2734 | 3.35522795 | 2.30962181 | 119.057939 |
| Round2 | 661 | 4787 | 1.76219797 | 5.51570415 | 119.379579 |
| Round3 | 625 | 1555 | 4.65791821 | 5.55546904 | 100.045111 |
| Round4 | 4234 | 386 | 4.8010211 | 4.01666093 | 107.632783 |
| Round5 | 4279 | 359 | 3.53140473 | 5.4268496 | 112.21418 |
| avg\_time |  |  | 3.62155399 | 4.56486111 | 111.665919 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| G5 | src | dst | Al1 | Al2 | Al3 |
| Round1 | 4530 | 2399 | 2.02332902 | 2.30998588 | 115.010392 |
| Round2 | 4866 | 2422 | 1.41840482 | 4.30468702 | 114.199901 |
| Round3 | 4972 | 2384 | 0.24075913 | 5.92626619 | 100.031824 |
| Round4 | 96 | 875 | 0.23407817 | 5.64327717 | 117.272001 |
| Round5 | 4720 | 156 | 1.0872128 | 4.37782598 | 107.552492 |
| avg\_time |  |  | 1.00075679 | 4.51240845 | 110.813322 |

**Part4. Performance Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| avg\_time(s) | Al1 | Al2 | Al3 |
| Sparse Graph | 0.556455717 | 0.2259128 | 0.38890233 |
| Dense Graph | 2.325087385 | 4.44237707 | 112.746067 |

Theoretically, these algorithms should run on faster on sparse graph than in dense graph. And the experiment results corresponds with that.

**Analysis and Discussion**

From the test data above, we can see that the running time for each algorithm differs from each other. While in the sparse graph, the contrast of running time is not that obvious. However, in the dense graph, although Kruskal’s algorithm and Dijkstra’s algorithm using heap have same time complexity, the traversing of the 2,500,000+ edges made it much slower than expected due to the density of the graph.

Based on different situations, the algorithms all have their own advantage. For example, Kruskal's algorithm enables generating an MST for once and then we can use it to search for any path in the future. Dijkstra’s algorithm has better performance facing the condition when we only need to find one single path between source and destination vertices.

**Further Improvements**

In this project, the graph structure is completed using dictionary structure, which is better in search and operations compared with the adjacent matrix as it takes O(1) time for basic operations.

I think there's still some improvements can be done in using the max heap structure, more various subroutines should be implemented for more convenient operations.

**Further Research**

Professor Chen mentioned that we can use the 2-3 tree structure to improve the performance of sorting and searching for the best fringe node in Dijkstra's algorithm.

Also if we put Divide-and-Conquer thought into building the MST and execute certain operations like Prim's algorithm or other more efficient algorithms, we can greatly reduce the scale of the problem in the dense graphs and then lower the time complexity into linear time.

**Project Source Code**

Source code is pushed on <https://github.com/tomatoJr/629project>