CSCE – 629 Analysis of Algorithms
Fall 2019

Course Project – Network Optimization

DHILTOM PANDARAPARAMBIL ANTONY

UIN – 327008700

November 26th, 2019

In this project, I have implemented a Network routing protocol that calculates the maximum bandwidth path using the following three algorithms:

- Modified Dijkstra's Algorithm without using a heap structure
- Modified Dijkstra's Algorithm using a heap structure for fringes
- Modified Kruskal's Algorithm, in which the edges are sorted by HeapSort

Random Graph Generation

The classes for vertex and graph are implemented. The vertex class maintains a list of neighboring vertices for each vertex. The graph class has both the adjacency list and adjacency matrix and also a function to add edges to the graph. Even though adjacency matrix has higher space complexity, it is used here because it allows for constant time access of edge weight rather than linear time as in the case adjacency lists. In order to have a connected graph, a cycle is created with all the vertices and then we add edges to the graph randomly.

Two type of graphs are generated here:

- G1 (Sparse graph)
  Average vertex degree is 6
  Total_edges = (Avg vertex degree/2) * Num_of_Vertices = 3 * Num_of_Vertices
  After connecting the initial cycle, with each vertex as source, we add two edges while checking that the edge is not already in the list.

- G2 (Dense graph)
  Each vertex is adjacent to about 20% of other vertices which are randomly chosen.
  After connecting the initial cycle, we take every pair of vertices other than the ones already connected and connect it with a probability of 20%. For this, a random function is used to provide values in the range (1,100) and if this value is less than or equal to 20, we connect the edge.

Heap Structure

A max-heap class is implemented with Maximum, Insert, Delete and HeapFy subroutines.

The heap is given by the list H, where each element H[i] gives the name of the vertex in the graph. The vertex values are given in the list D. To find the value of a vertex H[i] in the heap, we can use D[H[i]]. We also use a dict P, that maintains a mapping of vertex to its position in the heap. This is used while calling the Delete subroutine.

Routing Algorithms

The arguments passed to the each of the following algorithms are Graph G, source vertex s and destination vertex t. The output of the algorithm is the Maximum Bandwidth Path from s to t.

Let n be the number of vertices and m be the number of edges.

❖ **Modified Dijkstra's Algorithm without using a heap structure**

1) for v = 1 to n : status[v] = unseen #unseen=-1, fringe=0, intree=1 in code
2) status[s] = intree
3) for each edge [s,w] :
    a) status[w] = fringe
    b) wt[w] = weight(s,w)
    c) dad[w] = s
4) while (there are fringes):
    a) pick the best fringe v
    b) status[v] = intree
    c) for each edge [v,w]:
        i. if status[w] == unseen :
            a. status[w] = fringe
            b. dad[w] = v
            c. wt[w] = min {wt[v],weight[v,w]}
        ii. else if (status[w]== fringe) and (wt[w]<min{wt[v],weight(v,w)}):
            a. dad[w] = v
            b. wt[w] = min{wt[v],weight(v,w)}
5) Output : [dad[n],wt[n]]

Time = O(n^2)

❖ **Modified Dijkstra's Algorithm using a heap structure for fringes**

1) for v = 1 to n : status[v] = unseen #unseen=-1, fringe=0, intree=1 in code
2) status[s] = intree; Heap H
3) for each edge [s,w] :
    a) status[w] = fringe
    b) wt[w] = weight(s,w)
    c) dad[w] = s
    d) Insert(H,w)
4) while (there are fringes):
    a) Maximum(H); Delete(1)
    b) status[v] = intree
    c) for each edge [v,w]:
        i. if status[w] == unseen :
            a. status[w] = fringe
            b. dad[w] = v
            c. wt[w] = min {wt[v],weight[v,w]}
            d. Insert(H,w)

ii. else if (status[w]== fringe) and (wt[w]<min{wt[v],weight(v,w)}):
   a. Delete(H,w)
   b. dad[w] = v
   c. wt[w] = min{wt[v],weight(v,w)}
   d. Insert(H,w)

5) Output : [dad[n],wt[n]]

Time = O((m+n)logn)

Heap Functions:

- **Maximum(H):**
  return H[1]

- **Insert(H,a):**
  n = n + 1
  H[n] = a
  HeapFy(H,n)

- **Delete(H,k):**
  H[k] = H[n]
  n = n – 1
  HeapFy(H,k)

- **HeapFy(A,k):**

1) if k > 1 and A[k] > A[k/2] :
   1.1) h = k
   1.2) while h > 1 and A[h] > A[h/2]:
   1.3) swap A[h] and A[h/2] ; h =h/2
2) Else if k<= n/2 and A[k] < max(A[2k],A[2k+1]):
   2.1) h = k
   2.2) while h <= n/2 and A[h] > min{A[2h],A[2h+1]}:
   2.3) A[d] = max(A[2h],A[2h+1])
   2.4) swap A[h] and A[d]
   2.5) h =d

❖ **Modified Kruskal's Algorithm, in which the edges are sorted by HeapSort**

The heapsort function is defined in class Heap2.

Maximum Spanning Tree (T)

1) Sort the edges in non-increasing order: e1,e2,e3…….em using HeapSort
2) Heap2 T
   for v = 1 to n : MakeSet(v)
3) for i = 1 to m :
   a. ei = [ui,vi]
   b. r1 = Find(ui) ; r2 = Find(vi)
   c. if (r1!=r2) :
      Union(r1,r2)
      T = T + ei
4) return T

MakeSet(v)

i) dad[v] = 0
ii) rank[v] = 0

Union(r1,r2)

i) if rank[r1] > rank[r2]:
   dad[r2] = r1
ii) else if rank[r2] > rank[r1]
   dad[r1] = r2
iii) else:
   dad[r2] = r1
   rank[r1] += 1

Find(v)

i) w = v ; s = []
ii) while dad[w] != 0:
   s.append(w)
   w = dad[w]
iii) while s!=None:
   u = s.pop()
   dad[u] = w
iv) return w

Time = O(mlogn)

HeapSort(list): # input is a tuple of (source vertex, destination vertex, bandwidth)

1) Build a min heap from the input data with bandwidth as key

2) At this point, the smallest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

3) Repeat above steps while size of heap is greater than 1

4) Return the list of edges sorted by bandwidth in decreasing order

Do a Breadth First Search from vertex s on T to find the MBP from s to t.

Test Results

```
C:\Users\dhilt\Desktop\Courses\DS\Git Proj>python main.py

Number of Vertices :  5000


Iteration :  1

Sparse Graph edge count :  15000
Dense  Graph edge count :  2504268
Time to create Sparse Graph :  0.9853982925
Time to create Dense  Graph :  66.4802882671


RESULTS for SPARSE GRAPH :  1

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[3072, 271]     0.1935181618          0.0877709389          0.7220339775
[645, 170]      0.0229377747          0.0189476013          0.2253992558
[3547, 4228]    1.3274161816          0.1515929699          0.2293875217
[4874, 3117]    0.6921494007          0.0987362862          0.2194168568
[4569, 119]     0.4168848991          0.0817813873          0.2283895016
Avg             0.5305812836          0.0877658367          0.3249254227


RESULTS for DENSE GRAPH :  1

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[3072, 271]     0.3430836201          1.3643462658          73.8674654961
[645, 170]      3.0578277111          1.260666132           74.2255649567
[3547, 4228]    2.4803709984          0.4188802242          73.6431248188
[4874, 3117]    2.5601923466          1.6625549793          78.9668540955
[4569, 119]     0.3829762936          1.6625552177          72.8971610069
Avg             1.764890194           1.2738005638          74.7200340748
```

```
Iteration :  2

Sparse Graph edge count :  15000
Dense  Graph edge count :  2502423
Time to create Sparse Graph :  1.1709008217
Time to create Dense  Graph :  71.7960329056


RESULTS for SPARSE GRAPH :  2

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[4478, 3089]    0.755979538           0.1765625477          0.4617664814
[197, 168]      1.2466666698          0.0838112831          0.2293462753
[4310, 2088]    0.725063324           0.0488724709          0.236345768
[448, 3563]     1.3284835815          0.0658230782          0.2273914814
[1701, 1101]    0.4946436882          0.0688509941          0.2364070415
Avg             0.9101673603          0.0887840748          0.2782514095


RESULTS for DENSE GRAPH :  2

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[4478, 3089]    2.7815642357          0.4258608818          72.5340559483
[197, 168]      1.1519193649          0.6971421242          70.0248315334
[4310, 2088]    1.9228599072          1.5159442425          70.1035745144
[448, 3563]     1.7812371254          0.4218788147          76.3363053799
[1701, 1101]    1.4870262146          0.3699743748          72.0114450455
Avg             1.8249213696          0.6861600876          72.2020424843
```

```
Iteration :  3

Sparse Graph edge count :  15000
Dense  Graph edge count :  2504800
Time to create Sparse Graph :  1.1449725628
Time to create Dense  Graph :  69.5928430557


RESULTS for SPARSE GRAPH :  3

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[4970, 1330]    0.6153478622          0.1047201157          0.3939454556
[461, 679]      0.9096026421          0.1476054192          0.2293820381
[4907, 746]     0.8338086605          0.0997402668          0.22539711
[4415, 2728]    1.1818807125          0.1346437931          0.3380959034
[4526, 2797]    0.74401021            0.0488696098          0.2234027386
Avg             0.8569300175          0.1071158409          0.2820446491


RESULTS for DENSE GRAPH :  3

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[4970, 1330]    0.5525228977          1.4910533428          76.6550374031
[461, 679]      3.2093853951          1.3613948822          79.7936518192
[4907, 746]     2.9132111073          1.6934297085          69.5650274754
[4415, 2728]    3.604364872           2.2788646221          69.9121007919
[4526, 2797]    4.3244392872          1.9049060345          72.2418720722
Avg             2.9207847119          1.745929718           73.6335379124
```

```
Iteration :  4

Sparse Graph edge count :  15000
Dense  Graph edge count :  2504732
Time to create Sparse Graph :  1.1256766319
Time to create Dense  Graph :  66.5850276947


RESULTS for SPARSE GRAPH :  4

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[46, 527]       1.4650840759          0.0548520088          0.2952105999
[1186, 4770]    0.9145526886          0.0488669872          0.2253880501
[2779, 1059]    0.802852869           0.1236691475          0.3251304626
[4190, 1978]    0.098739624           0.1216726303          0.2224521637
[3248, 2404]    0.0638237             0.0139577389          0.2094419003
Avg             0.6690105915          0.0726037025          0.2555246353


RESULTS for DENSE GRAPH :  4

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[46, 527]       2.7925717831          1.1170129776          71.1906843185
[1186, 4770]    3.1775078773          0.7190897465          70.6281487942
[2779, 1059]    0.5136270523          0.3331100941          69.966984272
[4190, 1978]    1.0541861057          1.5139155388          66.7994158268
[3248, 2404]    1.3114998341          1.4899790287          74.9575278759
Avg             1.7698785305          1.0346214771          70.7085522175
```

```
Iteration :  5

Sparse Graph edge count :  15000
Dense  Graph edge count :  2501345
Time to create Sparse Graph :  1.2440023422
Time to create Dense  Graph :  65.6505243778


RESULTS for SPARSE GRAPH :  5

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[2344, 993]     0.5834085941          0.070810318           0.4009251595
[4644, 3734]    1.0621602535          0.0787894726          0.2373652458
[2831, 4687]    1.4860677719          0.01496315            0.336101532
[218, 1381]     1.1479682922          0.1325995922          0.2363600731
[895, 4835]     1.6087338924          0.1416208744          0.2253978252
Avg             1.1776677608          0.0877566814          0.2872299671


RESULTS for DENSE GRAPH :  5

Nodes           Dijkstra              Dijkstra_Heap         Kruskal
[2344, 993]     0.225435257           1.5229330063          79.0754513741
[4644, 3734]    2.85835886            0.564491272           79.3608863354
[2831, 4687]    2.9002420902          0.8397169113          69.5680196285
[218, 1381]     0.1685578823          0.1975114346          75.4306271076
[895, 4835]     3.0847187042          1.6735260487          72.0753185749
Avg             1.8474625587          0.9596357346          75.1020606041
```

```
Consolidated RESULTS for SPARSE GRAPHS :

Graph           Dijkstra              Dijkstra_Heap         Kruskal
1               0.5305812836          0.0877658367          0.3249254227
2               0.9101673603          0.0887840748          0.2782514095
3               0.8569300175          0.1071158409          0.2820446491
4               0.6690105915          0.0726037025          0.2555246353
5               1.1776677608          0.0877566814          0.2872299671
Avg             0.8288714027          0.0888052273          0.2855952168

Consolidated RESULTS for DENSE GRAPHS :

Graph           Dijkstra              Dijkstra_Heap         Kruskal
1               1.764890194           1.2738005638          74.7200340748
2               1.8249213696          0.6861600876          72.2020424843
3               2.9207847119          1.745929718           73.6335379124
4               1.7698785305          1.0346214771          70.7085522175
5               1.8474625587          0.9596357346          75.1020606041
Avg             2.0255874729          1.1400295162          73.2732454586
```

*All the above values are in seconds

Conclusion

For graph G1 (Sparse Graph)

The performance of Dijkstra's with heap > Kruskal's > Dijkstra's without heap

For graph G2 (Dense Graph)

The performance of Dijkstra's with heap > Dijkstra's without heap > Kruskal's

Therefore, Dijkstra's with heap gives best performance (least time) for both kind of graphs. Even though the time taken by Kruskal on dense graphs is significantly higher, I noted that sorting the edges in Kruskal's takes around 90% of the total execution time. So Kruskal can be used to find the maximum bandwidth path if we already have a sorted set of edges as input.

GitHub Link : https://github.tamu.edu/dhiltom07/Algo_MBP