CS 629 Analysis of Algorithms Project

Akshay Rao
UIN:127001397

1 Random Graph Generation

Here we must define two sub routines for random generation of an undirected graph with 5000 vertices.

- For the first graph G_1 we must maintain an average vertex in degree of 6.
- For the second graph G_2 we want to ensure that each vertex is adjacent to about 20% of the other vertices.
- In both cases we will be randomly assigning weights. Within a certain range.

For my implementation of the graph I have used an adjacency list as can be seen in Figure 1 (b).

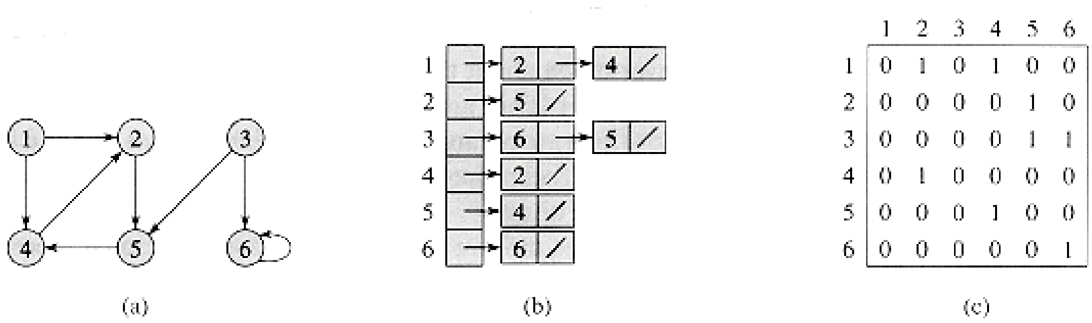

Figure 1: Representations of Graphs [1]

1.1 Graph Datastructure Description

The vertices of the graph are represented by nodes which has attributes as weight and a pointer to the next node. The in-degree is chosen at random using numpy.random.randint() from a range of possible values. For G_1 the average is maintained at 6 while for G_2 the average is maintained at 20% of the number of remaining vertices.

To ensure that the graph is connected, as suggested, there is an edge from each node to the subsequent node. To add adjacent vertices, we start with vertex ID 0 and first assign to this vertex. Then we remove vertex ID 0 from the collection of possible assignments and move on to the next vertex, ie, vertex ID 1. This process is repeated for each vertex ID from 0 to 4999, randomly selecting the in-degree and the adjacent vertices for each vertex. At the end we have an adjacency list representation for each of our graphs G_1 and G_2.

Implementation Details:

- adj_list[i]
- Class node with node.weight,node.nex,node.vert_id

2 Heap Structure

Here we implement a maximum heap structure with subroutines for maximum, insert and delete.

Maximum: Here we just return the first element of the heap. So we return H[0] and D[H[0].

Insert If say a new element is to be added to the heap, it is added initially at the end. Then while the max-heap property with the parent is broken, this element is swapped with its parent. To get parent we can simply use H[math.floor(i/2)]. After adding, we return H [ ] and D [ ].

Delete The element at the index to be deleted is swapped with the last element in the heap, and then the heap last element is deleted from the array. In this case it is removed from the python list H [ ]. Now we have an element at the index that might be breaking the max-heap property. There are two possibilities, either the property is broken with parents or it is broken with one or both children. After removing, we return the updated H [ ] and D [ ].

In the first case, the element is swapped with parent while the max-heap property remains broken.

In the second case, the element is swapped with the larger of its two children. The children of an index i can be obtained as H[2*i] and H[2*i+1].

The heap structure should be able to give the maximum element in constant time while the operations insert and delete should take O(log n) as the height of the heap should be bounded by $log_2(n)$.

Implementation Details:

- H [ ] python list for elements of heap data structure. This will contain the vertex IDs.

- D [ ] of size 5000, numpy array containing values of elements of heap. The value of an element can be accessed by D[H[i]].

3 Routing Algorithms

3.1 Djikstra-based Maximum BW Path without Heap

Dijkstras algorithm for single source shortest path can be modified to solve the problem of maximum bandwidth. The criteria for choosing the next vertex from the fringe to bring into the intree is changed from the one with the smallest edge weight to the one with the largest bandwidth or edge weight. Djikstras algorithm uses a locally greedy approach which turns out to be sufficient for this class of problems. At every iteration, the best vertex from the fringe is taken into the in tree. Its bandwidth and the dad array is updated. New vertices are added to the fringe from the unseen set, and their bandwidths are updated by giving them all another look. It is possible that the vertex whose bandwidth when initially added to the fringe can be increased when looked at again.

Here to choose the element with the largest bandwidth from the fringe, F [ ] is traversed to find the index of the largest element. This takes O(n) and could be one of the more costly operations in this algorithm.

Implementation Details:

- bw [ ] a python list containing the bandwidth of the path from s to each vertex ID 0,1,2..4999
- dad [ ] a numpy array, to get the s-t path
- adj_list [ ] for graph representation
- status [ ] to determine whether a vertex is in tree, fringe or unseen.
- F [ ] a python list containing the IDs of fringe vertices

3.2 Djikstra-based Maximum BW Path with Heap

Here the collection of fringe vertices is stored in a heap. This allows for constant time access to the 'best' member of the fringe to bring into the intree in the next iteration. This could help improve performance compared to the previous algorithm. Other than this detail the two are identical.

Implementation Details:

- bw [ ] a python list containing the bandwidth of the path from s to each vertex ID 0,1,2..4999
- dad [ ] a numpy array to obtain the s-t path
- adj_list [ ] for graph representation
- status [ ] to determine whether a vertex is in tree, fringe or unseen.
- F_H [ ] the heap with the IDs of fringe vertices
- D [ ] a numpy array containing the values of elements in the heap.

3.3 Kruskal-based Maximum BW Path with Heap-sort

Here first obtain a collection of all the edges. The edges are sorted into decreasing order. Then one by one we take the largest edge and join the vertices of that edge to form a sub tree. Two vertices are only joined if they belong to separate sub trees. This process is facilitated by Union and Find. A dad array is used to determine which subtree a vertex belongs to. Initially all values are set to -1, ie, each is a separate subtree. As we progress the dad of vertices are updated. Finally once we have progressed through all the edges, we will have a maximum spanning tree that will allow us to obtain a maximum bandwidth path for any pair of s and t. For this implementation a new graph was used to represent the maximum spanning tree as an adjacency list. To get the s-t path we can perform a dfs from either s or t.

the edges are sorted by heap sort which runs in O($nlogn$). For this we use the heap we have defined from earlier. To sort the collection of edges, the edges are first inserted into the heap and heap is created. Next, we carry out maximum ( ) and delete ( ) successively in iterations until the heap is empty.

Implementation Details:

- edges [ ] contains the bandwidth and end vertices of all edges in our graphs G_1 and G_2
- dad [ ] gives us information about which subtree the vertex belongs to. Initially it is set to -1
- adj_listk [ ] for graph representation of spanning tree
- rank [ ] to determine the depth of a tree to help decrease the time spent in Find ( ) operation
- E_H [ ] the heap for sorting the edges

4 Testing

The time elapsed for each of the algorithms on 5 generated pairs of graphs for 5 pairs of s-t are given in the tables below. The time elapsed is measured as the time from initialization until a dad array is obtained, i.e, we have the s-t path. For kruskals, this time elapsed has also included the time taken for dfs to obtain the s-t path. For each of the five iterations, the two graphs G_1 and G_2 are generated and then five pairs of s-t are chosen at random. Each of the algorithms are run one after the other while measuring the time elapsed for each.

Time elapsed measurement:

- Djikstra no heap and Djikstra with heap: start here is defined at the step before we begin the first step, i.e, before we add the vertices adjacent to the source vertex. The end time is the step after we have obtained the final dad array.
- kruskal: start is defined as the step before sorting of edges. The end time is defined at the step after carrying out dfs from s or t.

4.1 Observations

- Across all algorithms, the time elapsed on the much denser G_2 is greater by a significant factor.
- For G_1 The quickest algorithm is Djikstra with heap followed by Kruskal followed by Djikstra without heap.
- For G_2 The quickest algorithm is Djikstra with heap followed by Djikstra without heap followed by Kruskal.
- In G_1 it can be seen that using the heap data structure for Djikstra's yielded an improvement in running time by nearly a factor of 10 or an absolute difference of around .9 s. On the other hand in G_2 the same yielded an improvement of around 1 s but the factor was less.
- Kruskal's algorithm in G_1 performs about midway between the other two algorithms, but in G_2 it is by far the slowest by around 50 s.

4.2 Discussion

We can see that Kruskal's algorithm scales poorly as the graph becomes more dense,ie, as we add more edges. Kruskals must first sort all of the edges so this will be O($nlogn$) by heap sort. Maximum ( ) is constant time but deletion is O($logn$) and so this becomes O($nlogn$) as we must sort n elements. The large number of edges can greatly increase the running time. Kruskal's has a running time of O($ElogE$) for sorting all the edges and this may help explain the increased running time. For the dense graph G_2 there are a very large number of edges ($E_2 \sim 2.5 * 10^6$). On the other hand for the graph G_1 there are far fewer edges ($E_1 \sim 2.2 * 10^4$) and this may help to explain the improved performance compared to kruskal on G_2 as well as compared to Djikstra with no heap on G_1.

If we take the complexity of kruskals as $ElogE$ and compare the estimated times on G_1 and G_2 we get a ratio of $\sim \frac{2.5*10^6 log(2.5*10^6)}{2.2*10^4 log(2.2*10^4)} = 167.4$ while the actual ratio is $\sim \frac{5}{0.1} = 166.7$ if we take time elapsed as .36s and 60s for the two graphs respectively. This is in good agreement with the expectation.

For djikstra the time complexity is $O(VlogV + ElogV)$ and for connected graphs $O(ElogV)$ with a binary heap. Taking the number of vertices as 5000 and the number of edges as mentioned earlier we get a ratio of $\sim \frac{2.5*10^6}{2.2*10^4} = 113.63$ while the measured ratio is $\sim \frac{5}{0.1} = 50$. Time complexity is $O(Elogv+V^2)$ for djikstra without a binary heap. So we estimate $\sim \frac{2.5*10^6 log5000+5000^2}{2.2*10^4 log5000+5000^2} = 9.44$ while the measured ratio is $\sim \frac{6.3}{0.92} = 5.76$.

So we have for the time elapsed ratios:

Dijkstra_noheap

$$Expected \sim \frac{2.5 * 10^6}{2.2 * 10^4} = 113.63 Measured \sim \frac{5}{0.1} = 50$$

Dijkstra_heap

$$Expected \sim \frac{2.5 * 10^6 log5000 + 5000^2}{2.2 * 10^4 log5000 + 5000^2} = 9.44 Measured \sim \frac{6.3}{0.92} = 5.76$$

Kruskals

$$Expected \sim \frac{2.5 * 10^6 log(2.5 * 10^6)}{2.2 * 10^4 log(2.2 * 10^4)} = 167.4 Measured \sim \frac{5}{0.1} = 166.7$$

For djikstra with out heap, to obtain the maximum bandwidth element in the fringe it will be O($logn$) as we must traverse the entire data structure containing the fringes. Using a heap on the other hand yields the largest element in constant time. This may help explain the improved performance compared to without using a heap.

Possible improvements included using a Fibonacci heap instead of a binary heap as it allows for constant time decrease key instead of O($logn$). This would help improve performance for the dijkstras with heap implementation. With binary heap the complexity is $O(VlogV + ElogV)$ but using a fibonacci heap this becomes $O(VlogV + E)$. This could be especially helpful with graphs with many edges as the second term (which is dominating as E is very large) would be decreased by a factor $logV$. [2]

4.3 Tables

Time elapsed (s) for Graph G_1			
No.	Djikstra No Heap	Djikstra Heap	Kruskal
1a	.87	.09	.35
1b	.96	.09	.36
1c	.88	.09	.36
1d	.85	.09	.36
1e	.90	.09	.36
2a	.94	.10	.36
2b	1.04	.10	.36
2c	.93	.10	.36
2d	.93	.10	.36
2e	.87	.10	.36
3a	.96	.09	.36
3b	.82	.09	.36
3c	1.01	.10	.36
3d	.93	.09	.37
3e	.97	.10	.36
4a	.91	.10	.40
4b	.96	0.10	.36
4c	1.01	.11	.37
4d	.97	0.11	.37
4e	.90	0.10	.38
5a	.93	.10	.36
5b	.95	0.10	.37
5c	.95	.11	.36
5d	.87	0.10	.36
5e	.96	0.10	.38

Time elapsed (s) for Graph G_2			
No.	Djikstra No Heap	Djikstra Heap	Kruskal
1a	6.02	5.05	57.24
1b	6.18	5.04	57.17
1c	6.09	5.04	56.50
1d	5.93	5.00	56.43
1e	6.35	5.14	59.41
2a	6.11	5.08	57.76
2b	6.01	4.99	56.88
2c	6.19	4.99	57.03
2d	6.31	5.00	56.91
2e	5.86	5.32	58.15
3a	6.52	5.35	59.07
3b	6.39	5.06	58.01
3c	6.31	5.13	58.18
3d	5.98	5.13	58.14
3e	6.16	5.04	57.72
4a	6.90	5.32	61.90
4b	6.18	5.22	62.56
4c	6.76	5.45	61.55
4d	6.60	5.12	61.37
4e	6.08	5.42	60.68
5a	6.35	5.47	60.86
5b	6.35	5.11	58.75
5c	6.45	5.14	60.37
5d	5.85	5.09	58.70
5e	6.30	5.11	58.21

References

[1] USTC Intro to Algorithms: CHAPTER 23: ELEMENTARY GRAPH ALGORITHMS.

[2] Cormen T Leiserson C Rivest R Stein C. Introduction to algorithms second edition.