Course Project CSCE 629 Texas A&M University
Reid OBoyle
11/30/2021

1. Introduction

In this project I will be testing two different graph structures on 3 different Maximum-Bandwidth-Path (MBP) algorithms and reporting the results of each. Both graph structures will be undirected and will contain 5000 vertices. In the first graph (G1) edges will be randomly assigned such that the average vertex degree equals 6. In the second graph (G2) edges will be randomly assigned such that each vertex is adjacent to approximately 20% of the other vertices. Both of the graphs will have randomly weighted edges. The first MBP implementation will be created using Djikstra's where fringes are held in an unsorted list. The second MBP implementation will also use Djikstra's algorithm, but instead hold fringes in a Max-Heap. The final MBP implementation will use Kruskal's algorithm to find a Max Spanning Tree while sorting its edges using Heap-Sort and using a DFS to find the MBP.

2. Methods

1.1. Graph Structure

Both the graph and algorithms used will all be implemented and tested using python. In both graphs G1 and G2, vertices are indicated by integers 0-4999. To ensure all vertices are reachable, in both graphs initial edges are created so that there is a cycle connecting all vertices. This is done by creating a list with integers from 0-4999 representing vertices, which is then randomly shuffled and edges are created from index pairs [0,1],[1,2] … [4999,0]. The degree of each vertex in G1 is kept using a list of size 5000 with initial values of 0. G1 continually adds an edge between two random vertices with a random weight between 10,000 and 100,000 until (the sum of vertex degrees)/(5000 vertices) >= 6.0. This ensures that the average vertex degree is 6.0 for the graph. In the second graph G2, the degree of every vertex is held in a list and edges are added for each vertex until its degree >= 750 (15% adjacency). While choosing edges to add to each vertex, the graph only chooses vertices with degree < 1250 (25%) to keep each vertex around 20% adjacency. If the total average adjacency of all the vertices is less than 1000 (20%) more edges are randomly added to the graph until a total average of 20% is obtained.

1.2. Max-Bandwidth-Path Algorithm

For Max-Bandwidth-Path implementation. The first algorithm we will try is Dijskstra's while holding fringes in an unsorted list which is known to take O(n^2) time. This implementation of Dijkstra's takes in a Graph G, starting vertex s, ending vertex t, and n indicating the number of vertices. This implementation initially creates a list of strings the size of n with each string indicating the status of a vertex. They are all initially set to 'unseen' and the algorithm then sets each adjacent vertex of s to 'fringe'. It then iterates until there are no more 'fringe' vertices left (a.k.a. All vertices are 'in-tree' or 'unseen') and each iteration finds the 'fringe' v vertex with the largest bandwidth. It then sets the 'fringe' vertex to 'in-tree' and checks each adjacent vertex from the vertex v. If the adjacent vertex is 'unseen' it is set to fringe and its bandwidth is set to the minimum of v's bandwidth or the weight of the edge from v to itself. If the vertex is already a 'fringe' it repeats the previous step of setting the bandwidth to the minimum only if its bandwidth is already smaller than that minimum. In order to keep track of the path taken, a dad[] array is created in which the value at the index of a vertex is the previous vertex. This allows the algorithm to print the path at the end by using the dad array.

```
Dijkstra's Algorithm (Version I)            Time = O(n^2)
1. for (v = 0; v < n; v++) status[v] = unseen;
2. status[s] = in-tree;
3. for (each edge [s, w])
       status[w] = fringe; dad[w] = s;  bw[w] = cap(s, w);
4. while (there are fringes)
4.1    pick the fringe v of the largest bw[v]; status[v] = in-tree
4.2    for (each edge [v, w])
4.2.1     if (status[w] == unseen)
             status[w] = fringe;   bw[w] = min{ bw[v], cap(v, w) };  dad[w] = v;
4.2.2     else if (status[w] == fringe) & (bw[w] < min{ bw[v] + cap(v, w) })
             bw[w] = min{ bw[v], cap(v, w) };  dad[w] = v;
5.  if (status[t] != in-tree) return ('no s-t path');
    else x = t;
         print(x);
         while (x != s) print(x);  x = dad[x].
```

Figure 1. Dijkstra's Pseudocode Algorithm for Max-Bandwidth-Path

1.3. Max-Bandwidth-Path using Max-Heap Implementation

The second algorithm also uses Dijkstra's with the only difference being the fringes are stored in a max heap where the fringe with max bandwidth can be found in constant time and fringes are inserted and deleted into the max heap. This is generally known to be quicker and take time O(mlogn). The max heap works by holding a list H of size 5000 indicating the vertices in the heap tree structure where H[1] is always the vertex with the largest bandwidth. A list D of size 5000 where the indexes represent vertices and the values are the bandwidth of the corresponding vertex. A list P in which the values are the index in the heap (H) of the corresponding vertex (index in P). When inserting a vertex into the heap, the vertex is added to the last position of the heap and swapped with its parent until its bandwidth is less than its parent (up-heap). When finding the maximum fringe, the heap returns its max bandwidth vertex H[1] and replaces it with the last element. It then swaps the replaced vertex with its left or right child until it is larger than both its children (down-heap). When deleting a vertex, the vertex is replaced by the last vertex in the heap and up-heaps or down-heaps until the max heap is fixed.

```
Dijkstra's Algorithm (Version II)            Time = O(m log n)
1.for (v = 0; v < n; v++) status[v] = unseen;
2. status[s] = in-tree;  F = Φ;  \\ F is a set of fringes
3. for (each edge [s, w])
       status[w] = fringe; dad[w] = s;  bw[w] = cap(s, w);   Insert(F, w);
4. while (there are fringes)
4.1    v = max(F);  status[v] = in-tree;  Delete(F, v);
4.2    for (each edge [v, w])
4.2.1     if (status[w] == unseen)
             status[w] = fringe;   bw[w] = min{ bw[v], cap(v, w) };
             dad[w] = v;    Insert(F, w);
4.2.2     else if (status[w] == fringe) & (bw[w] < min{ bw[v] + cap(v, w) })
             Delete(F, w);   bw[w] = min{ bw[v], cap(v, w) };
             Insert(F, w);  dad[w] = v;
5.  if (status[t] != in-tree) return ('no s-t path');
    else x = t;
         print(x);
         while (x != s) print(x);  x = dad[x].
```

Figure 2. Dijkstra's with max heap pseudocode for Max-Bandwidth-Path

1.4. Kruskal's for Max-Bandwidth-Path Implementation

Kruskal's algorithm is known for solving a maximum spanning tree on a given graph G. But, it can also be proved that for any s and t vertices the unique path in the maximum spanning tree T is also a Maximum-Bandwidth-Path from s to t. We can then use this knowledge to implement a third algorithm that solves MBP using a modified version of kruskal's algorithm. For the best performance, we order the edges in kruskal's using heap sort in decreasing order using a max heap and the heapify algorithm. From there kruskal's uses Makeset,Union,and Find to create a maximum spanning tree T. I then created a max spanning tree in a graph representation that would easily be able to handle a DFS. I then could do a DFS on s while tracking the parent of each vertex using a dad[] array to find the path from s to t. We then have obtained a MBP using Kruskal's algorithm in O(mlogn) +O(m+n) for the DFS.

```
1. Sort the edges in non-increasing order using a
maxheap and heapsort:
      e1, e2, ..., em; sorted from heapify
2.    For (v = 0; v < n; v++) MakeSet(v);
3.    for (k = 1; k <= m; k++)
3.1      let ek = [uk, vk];
3.2      r1 = Find(uk); r2 = Find(vk);
3.3      if (r1 ≠ r2)
            T.add(ek)
           Union(r1, r2);
4.    Run DFS(s) on T to find path from s to t using
dad[] array
```

Kruskal's algorithm runs in time O(m log n)

```
MakeSet(v);
  p[v] = -1;
  h[v] = 0.
```

```
Find(v)
1. w = v;
2. while (p[w] ≠ -1)
       w = p[w];
3. return(w).
```

```
Union(r1, r2);
  if (h[r1] > h[r2]) p[r2] = r1;
  else if (h[r2] > h[r1]) p[r1] = r2;
  else \\ h[r2] = h[r1])
      p[r2] = r1; h[r1] = h[r1] + 1.
```

Figure 3. Kruskal's Algorithm for Max-Bandwidth-Path

3. Testing Setup

In order to better confirm any results that we find, the algorithms and graph structures will be tested multiple times. For any given G1 and G2 generated graphs, 5 randomly chosen source and destination vertex pairs will be assigned and the 3 algorithms will be tested on all 5 pairs for both graphs (G1 and G2). Additionally this will be done 5 different times on 5 different G1 and G2 graphs that will be generated by choosing a different seed for the random number generator in python. We will use the time() builtin tools in python to collect the total time taken for each algorithm on a given graph. We can then compare the time taken between graphs and the algorithms as well using python graphing tools to create bar and line charts.

4. Results and Discussion

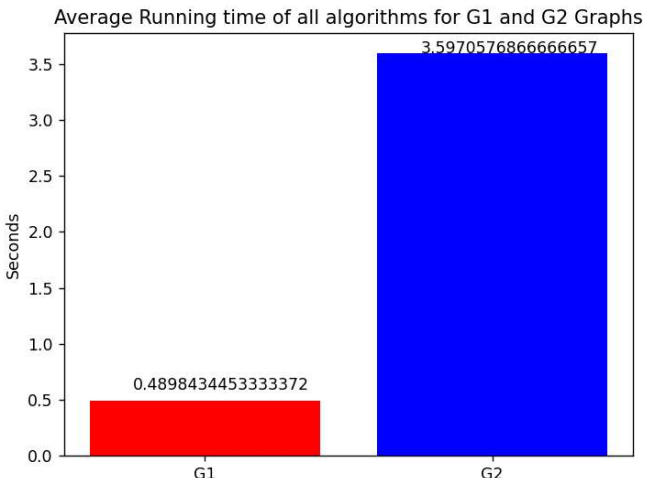

Figure 4: The average running time

Among all three algorithms for G1 and G2

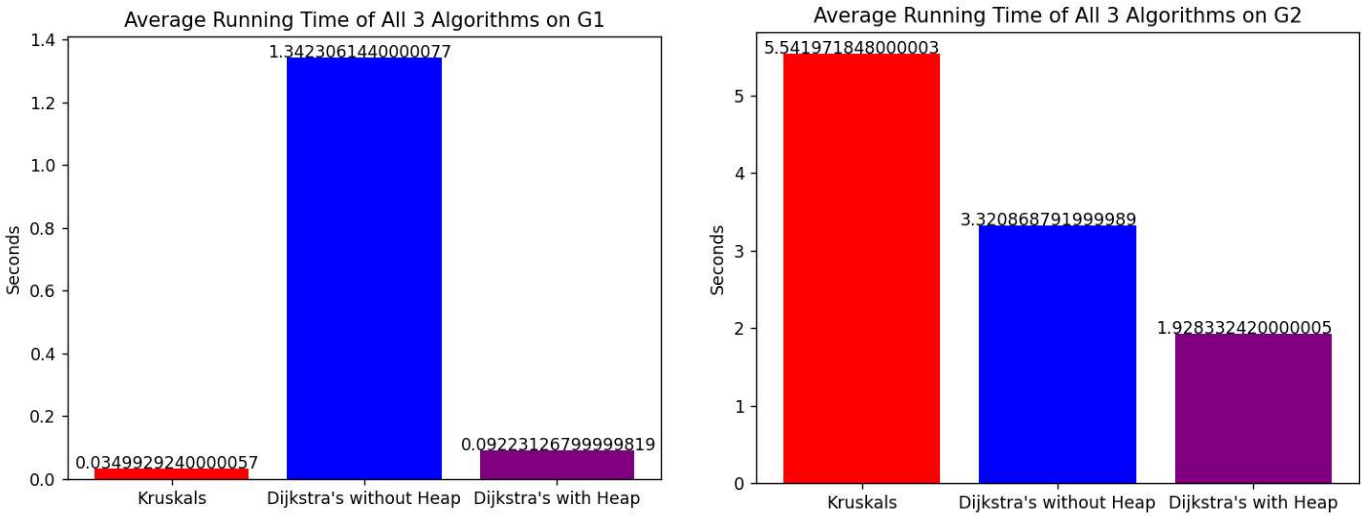

Figure 5. Average running time of Kruskals, Dijkstra's using an unsorted array to store fringes, and Dijkstra's using MaxHeap to store fringes.

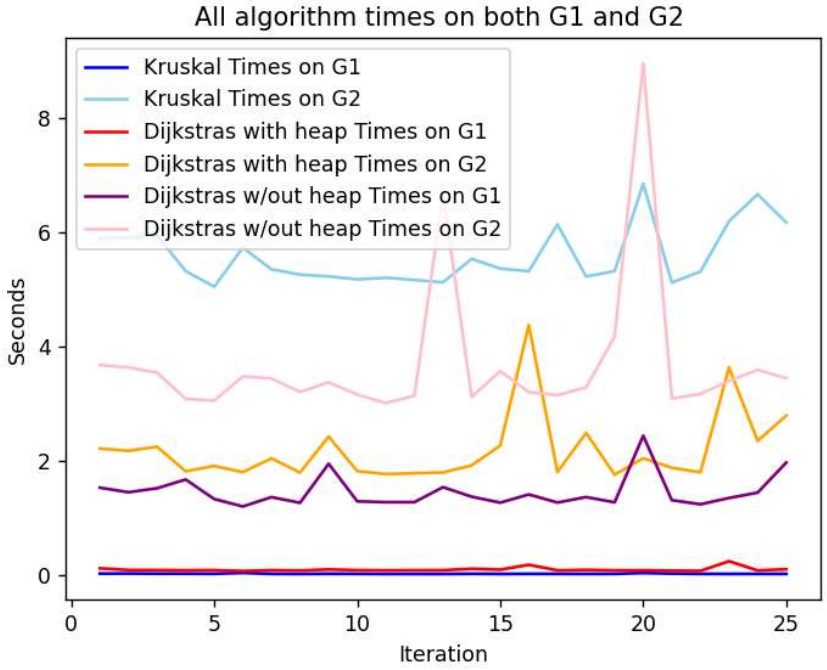

Figure 6. Algorithms times on each algorithm for both G1 and G2 over all 25 iterations.

From Figure 4 we can see that generally these algorithms take much longer on G2 than G1. That is not completely unexpected because even though the time complexity does not change for the algorithms we are running from G1 to G2, G2 has significantly more edges. G1 and G2 have the same amount of vertices but in G2 each vertex averages about 1000 edges on a graph with 5000 vertices. But in G1 the average vertex degree is only 6 so there are a lot less edges in G1. Kruskal's runs in O(mlogn) and Dijkstra's O(n^2) and O(mlogn) respectively. But in practice

the number of edges m makes for a large difference in computation time especially with the large amount of edge difference we have from G1 to G2.

In Figure 5 we can see that Dijkstra's was significantly faster when using a max heap to store fringes which reinforces what we know about their running times. Kruskal's was faster on G1, but saw a very large increase in computation time when running on G2. I found it very surprising that Kruskal's algorithm was slower than Dijkstra's without a heap in G2. I expect that doing a DFS after using Kruskal's adds some runtime with the large number of edges in G2 as DFS takes O(m+n) time. From these results I would say that it is best to use Kruskal's on graphs with smaller number of edges and Dijkstra's with max heap when a graph has a large amount of edges.

In figure 6 we can see that G1 is much faster than G2 for all three algorithms and Kruskal's is fastest for G1 but by a small margin relative to the other algorithm times. For G2 we can see that Dijkstra's with a heap was fastest and there is a much larger difference in the times from the algorithms. But, because there are much more edges and all three algorithms take longer time it is easier to see the difference between them and we can still see from figure 4 that there is still a large difference but just at a smaller scale.

5. Conclusion

I expected Kruskal's to be slightly faster than Dijkstra's with a heap and Dijkstra's without a heap to be the slowest. In G2 I first expected that the slowdown in Kruskal's came from doing heapsort and heapify on the edges because there were so many of them. But it does not make sense because Dijkstra's also uses a max heap and should take about the same amount of time doing extractMax and insert operations that heapify takes in Kruskal's. I expect that the extra time realistically came from doing find operations and DFS in Kruskal's on G2. I also did not use the exact same code for heap implementation in Kruskal's and Dijkstra's. So there is a chance that my heapify was not efficient in Kruskal's and I did not realize that, but the results from G1 being faster contradict that theory.

From these specific results, It is justifiable to say that in practice, networks where a node has large degree and many edges should use Dijkstra's with max heap to find a Max-Bandwidth-Path from one node to another. But if nodes have a relatively low degree Kruskal's is more efficient for Max-Bandwidth-Path and should be used. We can see that Kruskal's was 3x faster than Dijkstra's with max heap when average degree of vertices was 6. We can also conclude that if possible, it is best to keep network nodes to a limited number of edges between them to get better performance. Maybe if possible, nodes with large numbers of edges could be split into multiple nodes with fewer edges.

I would expect that if we try these same two graph structures but differ the amount of vertices we might see different results. In all of our tests, the amount of vertices was constant (5000) for both G1 and G2. In future work, a good way to strengthen our results and claims would be to try

differing amounts of vertices with constant vertex degree and vertex adjacency. Then we could see if the number of vertices has any effect on the algorithm performance.

In practice, a possible good way to deal with best performance would be to use Dijkstra's or Kruskal's based on the situation. A threshold based on the edge adjacency average or vertex degree could be built to decide in constant time whether to use Kruskal's or Dijkstra's based on the network. That way if a network has a large amount of edges per vertex, we could decide to use Dijkstra's instead of Kruskal's to save the extra computation time from Kruskal's in that scenario.