SIMPLE TESTS ON TWO POPULAR MAX-BANDWIDTH ALGORITHMS

DAT NGUYEN
331007653

Abstract

With the continuous expansion and evolution of modern communication technologies, network optimization has always been an interesting research topic of computer science. For the Analysis of Algorithms Course Project, let the students gain hands on experience and intuition on one of many profound max-bandwidth algorithms by manually write the algorithms in their lower-level language of choice.

1 Introduction

In this project, two profound max-bandwidth algorithms, Dijkstra (1959) and Kruskal (1956), are to be compared in terms of performance.

Throughout this paper, we are using the term ”graph size” to refer the total number of vertices of a graph. Moreover, some formula being displayed would have two variables n and m. Unless stated otherwise, n will represent the number of vertices of the graph (graph size), and m is the number of edges of the graph.

2 Background

Max-bandwidth problems’ goal is to find the the path with maximal flow in a weighted graph between a pair of source and sink vertices (s and t). Although both algorithms implemented in the project are principled about greedy methods, where it makes locally optimal choices that would lead to potential optimal outcome at the end, each is unique in the supporting structures and steps taken to reach the end goals.

2.1 Dijkstra’s max-bandwidth algorithm

At the high level, Dijkstra’s algorithm works as follows: starting from node s, the algorithm keep a set of fringe nodes (called fringers) which are the border nodes as we are expanding the maximum tree until we reach the sink node t. The greedy decision is made when a fringer node with maximal bandwidth from s is selected to be part of the maximum tree structure that we are constructing. We sums up the general algorithm for Dijkstra in algorithm 1:

Algorithm 1 Dijkstra’s max-bandwidth algorithm

```
 1: procedure DIJKSTRA-BW(G, s, t)
 2:     for v=0 ... (n-1) do
 3:         status[v] = UNSEEN;
 4:         bw[v] = 0; dad[v] = -1;
 5:     end for
 6:     status[s] = INTREE;
 7:     bw[s] = -1; dad[s] = -1;
 8:     for each w adjacent to s do
 9:         status[w] = FRINGER;
10:         bw[w] = bw(s, w); dad[w] = s;
11:     end for
12:     while there are Fringers do
13:         pick fringer v with largest bw[v];
14:         status[v] = INTREE;
15:         for each w adjacent to v do
16:             if status[w] UNSEEN then
17:                 status[w] = FRINGER;
18:                 dad[w] = v;
19:                 bw[w] = min(bw[v], bw(v, w))
20:             else if status[w]=FRINGER & bw[w]<min(bw[v], bw(v, w)) then
21:                 dad[w] = v;
22:                 bw[w]=min(bw[v], bw(v, w))
23:             end if
24:         end for
25:     end while
26:     return dad[0..n-1], bw[0..n-1]
27: end procedure
```

Let us further analyze the complexity of this algorithm. The first two for loops at lines 2 and 8 has O(n) run-time complexity. The while loop at line 12 should run n times for each time a Finger node is added to the tree. Picking fringer with largest bandwidth at line 13 takes time O(n) by linearly scanning the Fringe list. The for loop at line 15 also take time O(m) since each vertex has at most m out-going edges. Therefore, the algorithm's runtime is bounded by O(n^2).

What can be done to improve the algorithm is to avoid linear scanning by using Max Heap structure to select fringer nodes with maximal bandwidth in O(1) + O(log(n)) (Max heap MAXIMUM and DELETE operations).

Algorithm 2 New Dijkstra's max-bandwidth algorithm

```
 1: procedure NEW-DIJKSTRA-BW(G, s, t)
 2:     for v=0 ... (n-1) do
 3:         status[v] = UNSEEN;
 4:         bw[v] = 0; dad[v] = -1;
 5:     end for
 6:     status[s] = INTREE;
 7:     bw[s] = -1; dad[s] = -1;
 8:     MH = new MaxHeap structure
 9:     for each w adjacent to s do
10:         status[w] = FRINGER;
11:         bw[w] = bw(s, w); dad[w] = s;
12:         MH.INSERT(w, bw[w])
13:     end for
14:     while there are Fringers do
15:         v = MH.MAX()
16:         status[v] = INTREE;
17:         MH.DELETE(v)
18:         for each w adjacent to v do
19:             if status[w]  UNSEEN then
20:                 status[w] = FRINGER;
21:                 dad[w] = v;
22:                 bw[w] = min(bw[v], bw(v, w))
23:                 MH.INSERT(w, bw[w])
24:             else if status[w]=FRINGER & bw[w]<min(bw[v], bw(v, w)) then
25:                 dad[w] = v;
26:                 bw[w]=min(bw[v], bw(v, w))
27:                 MH.INSERT(w, bw[w])
28:             end if
29:         end for
30:     end while
31:     return dad[0..n-1], bw[0..n-1]
32: end procedure
```

With the use of max heap, the first two for loops at lines 2 and 9 has O(nlog(n)) run-time complexity. While the while loop at line 14 still runs n times, selecting fringer nodes with maximal bandwidth and delete it on the max heap takes O(log(n)). The for loop at line 18 repeats m times, and the content of the loop run in time O(log(n)). Therefore, time complexity of this new Dijkstra is O(nlog(n) + nlog(n) + nmlog(n)) = O((n+m)log(n)).

We would expect the newer Dijkstra algorithm to run faster than the first version for more parse graphs where $m \geq n^2/2$.

2.2 Kruskal's max-bandwidth algorithm

Take a different approach from Dijkstra's algorithm, Kruskal build its maximal tree by continuously add largest edges to an empty graph until all vertices in the graph are all connected. We present how we implement it at high-level with the following pseudocode.

Algorithm 3 Kruskal's max-bandwidth algorithm

```
 1: procedure KRUSKAL-BW(G, s, t)
 2:     MH = new MaxHeap structure
 3:     Maxtree = Graph G without any edge
 4:     for each edge e = u, v in G do
 5:         MH.INSERT(e, weight(u, v));
 6:     end for
 7:     while MH is not empty do
 8:         e = $u, v$ = MH.MAX();
 9:         if u and v are in different parts of the graph then
10:             Maxtree.AddEdge(u, v);
11:         end if
12:         MH.DELETE(e);
13:     end while
14:     Perform BFS or DFS on Maxtree to fill up dad[] and bw[]
15:     return dad[0..n-1], bw[0..n-1]
16: end procedure
```

Line 4 is where we sort all the edges. This step should take O(mlog(m)). The while loop at line 7 is secured to repeat m times. The operations inside the loop are max heap operations which take O(log(m)), while sequence of MakeSet, Find, and Union takes time O(log(n)). Thus the content of the while loop takes O(log(m) + log(n)). Graph traversal at the end takes O(m + n). Thus, the bottleneck is the while loop at line 7 which takes time O(m (log(n) + log(m))) overall. So that, when graph is parse, m=O(n), Kruskal should take

O(mlog(n)). However, for dense graph, m=O(n^2), Kruskal can take time O(mn) or O(mlog(m)).

3 Implementation

3.1 Benchmarking standard

Following the project instruction, we will be testing three algorithms stated in the previous section with two type of graph, parse (G1), and densely connected (G2). Specifically, in G1, average vertex degree is 6. We constructed this graph by firstly add edges to make a cycle that connect all vertices first (to make the graph connected). Then, edges of random weights are added randomly to the graph until the graph reach a target vertex count. Since for every added edge to the graph, the total degree of all vertices is creased by 2 (vertices at both ends of the newly added edge both have their degree increased by 1), we have the formula:

$$d = 2 \times e \tag{1}$$

with d being the total degree of all vertices in the graph and e being edge count of graph. Combine this piece with average vertex degree formula:

$$avgDeg = \frac{d}{n} \tag{2}$$

the target number of edges can be calculate using:

$$e = \frac{avgDeg \times n}{2} \tag{3}$$

Algorithm 4 demonstrates how G1 is created.

In constructing G2, similarly to G1, we firstly create a loop that connect all vertices. Then, for each vertex v in G, we add edges branching out of v until v is connected to 20 percent of the other vertices in the graph. Let us demonstrate it with the pseudocode in algorithm 5.

Implementing algorithm 1 and 2 are very straight forward. For algorithm 3, since the number of edges in dense graph G2 is so much larger than the number vertices n, array-based max heap has to allocate huge memory resource to perform heap sort. To be precise, in G2, each vertex is connected to 1000 other vertices, which is also the average degree of each vertex. Applying it to formula 3, there should be roughly 2.5 million edges in G2 (compared to about 15.000 edges in G1). Additionally, working towards the details of Kruskal's algorithms, disjoint set structure and operations are required to identify if two nodes are from different parts of the graph (Algorithm 3 line

Algorithm 4 Algorithm to create G1

```
 1: procedure CREATE-G1(G, s, t)
 2:     avgDeg = 6;  ▷ Average degree of vertices
 3:     targetNumEdges = avgDeg × graphSize / 2;
 4:     edgeCount = 0;
 5:     G = new graph of size $graphSize$;
 6:     for each vertex v do  ▷ cycle of vertices
 7:         w = (v + 1) mod $graphSize$;
 8:         weight = random weight value;
 9:         G.addEdge(v, w, weight);
10:         edgeCount++;
11:     end for
12:     while edgeCount ≠ targetNumEdges do
13:         get random vertices v and w (v ≠ w);
14:         if v is not adjacent to w then
15:             weight = random weight value;
16:             G.addEdge(v, w, weight);
17:             edgeCount++;
18:         end if
19:     end while
20:     return G;
21: end procedure
```

Algorithm 5 Algorithm to create G2

```
 1: procedure CREATE-G2(G, s, t)
 2:     adjPercent = 20;
 3:     targetNumVertex = (graphSize × adjPercent / 100)-1;
 4:     G = new graph of size $graphSize$;
 5:     for each vertex v do  ▷ cycle of vertices
 6:         w = (v + 1) mod $graphSize$;
 7:         weight = random weight value;
 8:         G.addEdge(v, w, weight);
 9:     end for
10:     for each vertex v in graph G do
11:         adjCount = degree of vertex v
12:         while adjCount < targetNumVertex do
13:             w = random vertex so that (w ≠ v)
14:             if v is not adjacent to w then
15:                 weight = random weight value;
16:                 G.addEdge(v, w, weight);
17:                 adjCount++;
18:             end if
19:         end while
20:     end for
21:     return G;
22: end procedure
```

9). With the three primitive disjoint set operators: MakeSet, Find, and Union, line 9 Kruskal's algorithm can be expanded with more details which can be seen in algorithm 6.

We run the experiment 5 times, and for each of 5 iterations, the algorithms (1 2 3) are iteratively applied on the newly generated pair of G1 and G2 graphs. For each pair of G1, G2, we randomly select 5 pairs of source and sink vertices (s, t). We the record the run time and give some analysis in the next section.

Algorithm 6 Detailed Kruskal's max-bandwidth algorithm

```
1:  procedure DETAILEDKRUSKAL-BW(G, s, t)
2:      MH = new MaxHeap structure
3:      DS = new DisjointSet structure
4:      Maxtree = Graph G without any edge
5:      for each edge e = u, v in G do
6:          MH.INSERT(e, weight(u, v));
7:      end for
8:      while MH is not empty do
9:          e = $u, v$ = MH.MAX();
10:         if DS.Find(u) == -1 then
11:             DS.MakeSet(u);
12:         end if
13:         if DS.Find(v) == -1 then
14:             DS.MakeSet(v);
15:         end if
16:         if DS.Find(u) $\neq$ DS.Find(v) then
17:             Maxtree.AddEdge(u, v);
18:             DS.Union(u, v)
19:         end if
20:         MH.DELETE(e);
21:     end while
22:     Perform BFS or DFS on Maxtree to fill up dad[] and bw[]
23:     return dad[0..n-1], bw[0..n-1]
24: end procedure
```

4 Results and Analysis

	Dijkstra's	NewDijkstra's	Kruskal's
G1	0.0434378	0.0021104	0.290287
G2	0.49623	0.461093	2.41676

Table 1: Average runtime in seconds of three max-bandwidth algorithm on parse(G1) and dense(G2) graphs

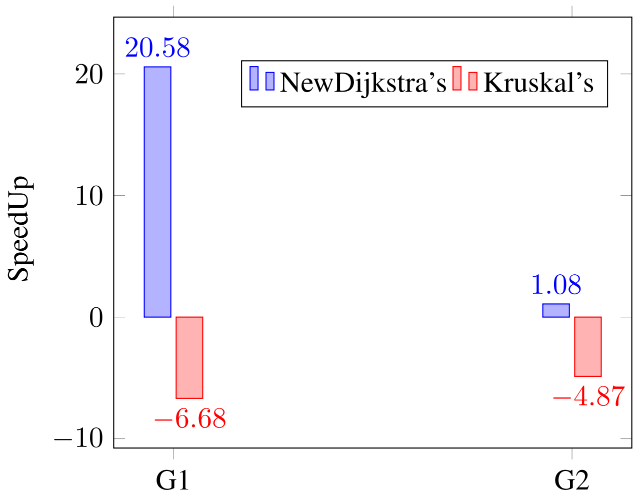

Figure 1: Speedup of NewDijkstra's, Kruskal's algorithms when compared to Dijkstra's

Results We show our experiment result in Table 1, Figure 1, 2.

Analysis As expected, New-Dijkstra's algorithm which make use of MaxHeap is an improvement over the version that use linear scan to look for maximal fringers. What stands out in the data is the runtime of Kruskal's algorithm, which is significantly slower than two version of Dijkstra's.
Figure 1 compares the algorithms directly when run on either G1 or G2. For parse graph G1, NewDijkstra's algorithm which use max heap performs up to 20 times better linear scan Dijkstra's, while we can see there is only a slight improvement in dense graph G2. This is due to the fact that in both version of Dijkstra, the step where we reach all neighbors of each vertex would take O(m). Therefore, for parser graph G1 where m ¡¡ n, NewDijkstra's algorithm perform much better; and in G2, the cost of this step has used up all the savings came from the use of max heap.
Since Kruskal's algorithm is O(mlog(m)) as explained in section 2.2, performing it on dense graph degrade its performance by roughly 8 folds as seen in figure 2. However, when performed on parse graph G1, Kruskal's performance is comparable to Dijkstra's algorithm (without max heap usage), which can be seen in figure 1.

5 Future Improvement and Conclusion

My code and progress on this project can be found at my Github repository.
We have performed some simple runs of some well-known max-bandwidth algorithms, and com-

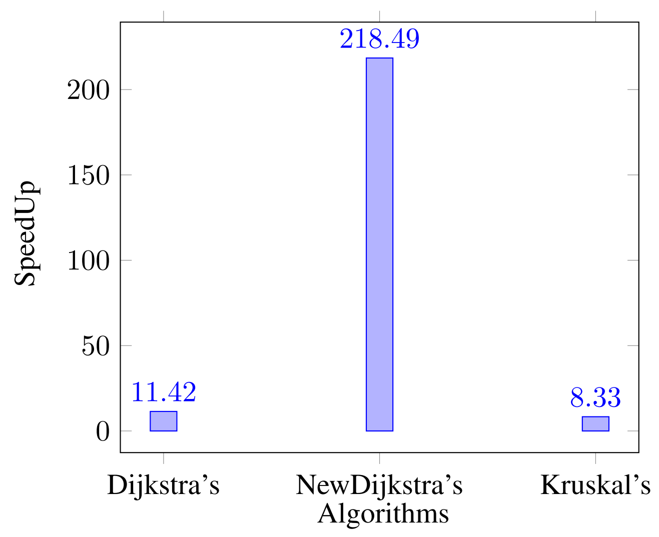

Figure 2: Speedup of algorithms when run on G1 compared to G2

pared them on two types of graphs, parse and dense. We have also successfully illustrate the significant of max heap structures and operations, and their affects on the performance of each algorithm. The experiments conducted for this project mainly focus on the run time as a measure of performance. However, memory usage is also a concern when it comes to effective algorithm. The use of array for instance access can be a waste of memory resource, especially for Kruskal's algorithm which uses array-based heap sort to store the edges. For the future work, I would love to see how integrate other data structures can save memory and how run time is affected by it.

References

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik 1*(1), 269–271.

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society 7*(1), 48–50.