Texas A&M University

Computer Science & Engineering Department

Analysis of Algorithm

Course Project

Maximum Bandwidth Path Analysis

Name: Li-Hen Chen

UIN: 928003907

1. Abstract

In this course project, I have implemented 2 famous algorithms in computer science for constructing Maximum Bandwidth (MBW) path which are Kruskal and Dijkstra. For this reason, I have implemented 2 types of graphs which are dense and sparse. Then, I have implemented 2 types of Dijkstra Algorithm which: simple and heap based. Final, these implantations are tested on 5 pairs of graphs randomly generated by my subroutines (dense and sparse). For each generated graph, also pick at least 5 pairs of randomly selected source-destination vertices and test the running time of different algorithm implementations.

2 Introduction

Nowadays, Networks have become corner-stone of our life which are used social Network [Ellison et al, 2007], Computer Network [Olifer et al, 2005] and etc.

There are some important parameters in networks that should be optimized. One of the most important parameters in this hot topic area of computer science is Maximum Bandwidth (MBW).

Maximum Bandwidth path problem: Given a source node s and a destination node t in a network G, in which each edge e is associated with a value w(e), construct a path from s to t in G whose bandwidth is maximized (the bandwidth of a path is equal to the minimum edge bandwidth over all edges in the path). Maximum Bandwidth problem is not a new problem. This have been studied for more than 40 years. Modification of Dijkstra [Corman et al, 2009] and Kruskal [Malpani et al, 2002] algorithms have been used for this problem for several years.

In this course project I have been implemented these algorithms in Python. At first, I will illustrate about implementation methods I have used for constructing random graphs. Then, I will explain about Heap structure and my implementation. Section 4 is about Dijkstra and Kruskal algorithms and their modifications and their implementation tricks. At last, there is a result section that I will show the results of my implementation.

3. Graph Generation

Presumption: The data structure I use to represent the Graph is Adjacency-List, which is

shown in figure 1. I assume the weight of edges which are added to the graphs are randomly generated with the mean 100 and standard deviation 20 in Gaussian distribution (normal distribution) in order to resemble the practical maximum bandwidth problem. The probability density for the Gaussian distribution is

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

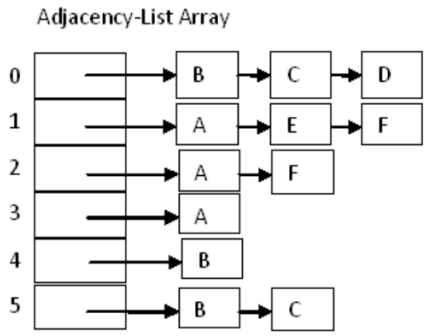


Figure 1. Adjacency list representation of graphs

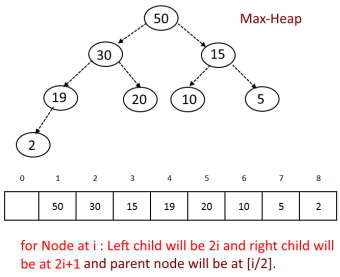
For the first graph G_1, to an undirected graph, because each edge is incident to two vertices and counts in the degree of both vertices, the average degree of an undirected graph is $2 * \frac{|E|}{|V|}$. In this case, if we would like to have a graph with average 6 degree, we would have $6 * \frac{|V|}{2} = 15000$ edges. As a result, I generate 15000 edges and randomly pick 2 different nodes, then add the edge to the nodes. Using this method I am able to generate G_1 where each node has average 6 degree. However, the graph generated by this method may have some nodes (extremely rare) with 0 degree, because the nodes are picked randomly each time. As a result, I changed my method that I first assign each node a random edge then using the above method (also convert 15000 to 10000) to meet the requirement.

For the second graph G_2, in order have each vertex be adjacent 20 % of other vertices randomly, I connect each vertex to other vertices randomly. I run on all 5000 vertices and

randomly choose approximately 500 (also normal distribution with mean 500) to connect. Why is 500? Because it's an undirected graph, we would like to make sure each 2 vertices do not connect twice so originally the 20% of edges is 1000, now we have to cut the number of edges added each iteration into half to be 500.

4. Max Heap Implementation

I implement Max Heap in binary tree, which is called binary heap. A binary heap is a complete binary tree which satisfies the heap ordering property. The data structure I choose for binary heap implementation is array which means a complete binary tree can be uniquely represented by storing its level order traversal in an array. The root is the second item in the array. We skip the index zero cell of the array for the convenience of implementation. Consider k-th element of the array, its left child is located at 2*k index, its right child is located at 2*k+1 index and its parent is located at k/2 index. We can see the following figure for clearer implementation.



The MaxHeap I implement the operations listed below.

Time Complexity Analysis:

The heap is constructed by an array and the access is O(1) Search is O(n) Insertion is O(n). The heap's structure is like a tree as following.

- Max: It returns the root element of the Maxheap. Its time complexity of this operation is O(1).
- popMax: It removes the maximum of the heap which is array[0]. When the max value is removed, the heap should maintain its heap property. The worst case it would traverse from root to leaves so the time complexity is O (log n).

- Heapify: We add a new key at the end of the tree. If new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
- Insert: We add a new key at the end of the tree. If new key is greater than its parent, then we don't need to do anything. Otherwise, we need to traverse up to fix the violated heap property. Since we way traverse from leaves to root, the time complexity is O(log n).
- Delete: Because we have recorded the vertex value and vertex name, we can search whole array. When we find the corresponding vertex name, we delete it and then heapify the array again.

5. Routing Algorithms

In this course project, I have implemented modification of Dijkstra and Kruskal Algorithms. In this section I would explain these 2 algorithms and their implementation tricks:

• Dijkstra: Dijkstra is one of the most popular algorithms for finding shortest path. I have implemented 2 versions of this algorithms. These two algorithms different in their implementation in their fringes:

– simple: simple Dijkstra refer to implementation of fringes without special data structure. In this version, complexity is equal to $O(n^2)$.

– heap: Heap Dijkstra refer to implementation of fringes based on MaxHeap which I explained in the previous section. Because of using better data structure for fringes which is Max heap, this algorithm's complexity would be $O(m \log n)$

Maximum Bandwidth Algorithm of Dijkstra is shown in following:

```
Algorithm  DIJKSTRA
 1: procedure DIJKSTRA(G,s,t)
 2:     for for each node v in G do
 3:         P[v] ← 0.
 4:         B[v] ← −∞.
 5:     B[s] ← 0
 6:     F ← ∅
 7:     for for each neighbor w of s do
 8:         P[w] ← s.
 9:         B[w] ← weight(s, w).
10:         add w to F.
11:     repeat
12:         remove the node u of maximum B[u] from F
13:         for each neighbor w of u do
14:             if B[w] == −∞ then
15:                 P[w] ← u.
16:                 B[w] ← minB[u], weight(u, w).
17:                 add w to F.
18:             else if (w is in F) & (B[w] < minB[u], weight(u, w)) then
19:                 P[w] ← u.
20:                 B[w] ← minB[u], weight(u, w).
21:     until (B[t] ≠ −∞) & (t is not in F)
```

If we are only interested in a shortest path between vertices source and target, we can terminate the search earlier at line 12 if u = target. A more general problem would be to find all the shortest paths between source and target (there might be several different ones of the same length). Then instead of storing only a single node in each entry of P[] we would store all nodes satisfying the relaxation condition.

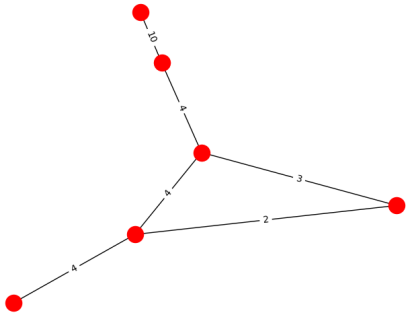
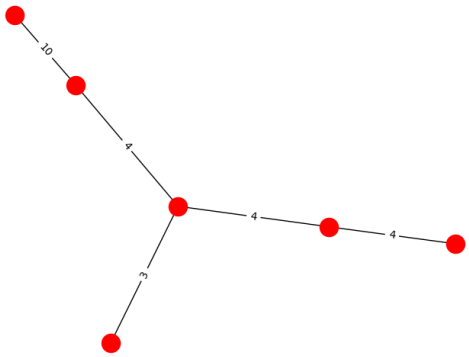
• Kruskal: Kruskal algorithm is useful for finding maximum spanning tree in a given graph G. Implementation of Kruskal is based on Union, Find and Make Set Operations. The time complexity of the algorithm is $O(E \log V)$.

```
Algorithm  MAX BANDWIDTH KRUSKAL
 1: procedure MAX BANDWIDTH KRUSKAL(G)
 2:     sort the links of G in terms of link bandwidth in non-increasing order: {e_1, e_2, ...., e_m}.
 3:     for each node v of G do
 4:         MAKE SET(v).
 5:     T ← ∅.
 6:     for i = 1 to m do
 7:         let e_i = [u_i, v_i].
 8:         r_1 ← FIND(u_i)
 9:         r_2 ← FIND(v_i)
10:         if r_1 ≠ r_2 then
11:             add e_i to T
12:             UNION(r_1,r_2)
```

6. Testing Results

First of all, I use a graph with small amount of edges to test the correctness of my three

algorithms. All 3 algorithms got same result as shown in figure below.

Before Algorithms	*After Algorithms*
	

I have tested my four algorithms on the 5 pair of graph in which 5 pair (s, t) randomly is chosen. For every algorithm, I would have 25 samples for dense and sparse graphs. I would calculate average of these running time and show them in the below Table. Each graph, I calculate the average running of every 5 samples, and list the 5 graphs separately. Then calculate the average running time of 5 graphs.

Graph 1

Algorithm	*Sparse*	*Dense*
Naïve Dijkstra	2.03 sec	8.59 sec
MaxHeap Dijkstra	0.10 sec	5.30 sec
Kruskal	0.06sec	18.12 sec

Graph 2

Algorithm	*Sparse*	*Dense*
Naïve Dijkstra	2.05 sec	9.50 sec
MaxHeap Dijkstra	0.12 sec	3.10 sec
Kruskal	0.06 sec	18.53 sec

Graph 3

Algorithm	*Sparse*	*Dense*
Naïve Dijkstra	2.08 sec	10.01 sec

MaxHeap Dijkstra	0.18 sec	4.49 sec
Kruskal	0.07 sec	18.59 sec

Graph 4

Algorithm	*Sparse*	*Dense*
Naïve Dijkstra	1.93 sec	10.36 sec
MaxHeap Dijkstra	0.12 sec	5.41sec
Kruskal	0.06 sec	18.19 sec

Graph 5

Algorithm	*Sparse*	*Dense*
Naïve Dijkstra	1.97 sec	8.36 sec
MaxHeap Dijkstra	0.07 sec	3.13 sec
Kruskal	0.06 sec	16.26 sec

Average Time of 5 Graphs for different Algorithms

Algorithm	*Sparse*	*Dense*
Naïve Dijkstra	2.012 sec	9.364 sec
MaxHeap Dijkstra	0.118 sec	4.286 sec
Kruskal	0.062 sec	17.938 sec

- **Discussion**

Let's first list the time complexity of each algorithms and have some discussions.

	Naïve Dijkstra	*MaxHeap Dijkstra*	*Kruskal*
Time Complexity	$O(\|V\|^2)$	$O((\|E\| + \|V\|) \log V)$	$O(E \log E)$ (in these cases) *or* $O(E \log V)$

For Sparse Graph

We can tell from 5 sparse graphs Kruskal and Dijkstra with max heap data structure has great performances and the Naïve Dijkstra has relatively bad performance. In fact, the results correspond to the time complexity of algorithms. The sparse graph, where each node has average 6 degree, has relative less edges where $|V|^2 \gg |E|$. For sparse graphs, that is, graphs with far fewer than $|V|^2$ edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a max heap as a priority queue to implement extracting maximum efficiently. To Kruskal, Kruskal first sort the edges by weight using a comparison sort in O(E log E) time; Next, we use a disjoint-set data structure to keep track of which vertices are in which components. We need to perform O(V) operations, as in each iteration we connect a vertex to the spanning tree, two 'find' operations and possibly one union for each edge. Even a simple disjoint-set data structure such as disjoint-set forests with union by rank can perform O(V) operations in O(V log V) time. Thus the total time is O(E log E) = O(E log V). As a result, Kruskal algorithm is able to reach slightly better performance to Dijkstra with max heap data structure in sparse graph according to the experiment results and analysis of time complexity. To sum up, the results match the time complexity analysis as shown below.

$$O(E \log E) > O((|E| + |V|) \log V) \gg O(|V|^2)$$

For Dense Graph

However, in the dense graph, where each node connects to 20% of other nodes, has large number of edges. Let's first look at Kruskal Algorithms and the table. We can see that Kruskal runs extremely slow in the dense graph because Kruskal algorithm's time complexity is bounded by $O(E \log E)$. In dense graph, the number of edges is relatively large so Kruskal algorithm runs slow. Then we can look at 2 Dijkstra algorithms, the Naïve Dijkstra is approximately 2 times slower than Dijkstra with max heap data structure. Indeed, for large E we can simplify the time complexity of Dijkstra with max heap data structure to $\mathrm{O(E \log V)}$. As a result, Dijkstra with max heap data structure is without a doubt faster than Naïve Dijkstra algorithm. How about Kruskal and Naïve Dijkstra algorithms? In this case, the number of edges is 1000 times than vertices so $O(E \log E) \gg O(|V|^2)$. We can conclude that Kruskal is not suitable for the dense graph.

Conclusion

I think we can have few conclusions from this experiment.

1. Data Structure indeed has great impact on the time complexity and the actual running time so we have to carefully choose the proper data structure.

2. Naïve approach without heap is not the one that we should choose in any scenario.

3. Kruskal can run efficiently on the sparse graph but not on the dense graph. In fact, that's another issue for maximum (minimum) spanning tree that Kruskal algorithm is better for sparse graph and Prim's algorithm is better for dense.

Finally, my recommendation from the entire analysis would be to use 2nd - heap based approach if we are completely unsure about how frequently the graph will change, how many s, t queries will be there per graph.

7. Further Research

There is Median-Finding Maximum Bandwidth Path algorithm with running time O(n) which would be much better than Dijkstra and Kruskal. Its basic idea is that the weights of edges in one set are not less than that in the other. If a spanning tree exists in subgraph composed solely with edges in larger edges set, it then computes a MBP in the subgraph, a MBP of the subgraph is exactly a MBP of the original graph. If a spanning tree does not exist, it combines each disconnected component into a new super vertex, then computes a MBP in the graph formed by these super vertices and edges in the smaller edges set. A forest in each disconnected component is part of a MBP in original graph. Repeat this process until two (super) vertices are left in the graph and a single edge with largest weight between them is to be added. A MBP is found consisting of all the edges found in previous steps.

Time Complexity:

The algorithm is running in O(E) time, where E is the number of edges. This bound is achieved as follows:

- dividing into two sets with median-finding algorithms in O(E)
- finding a forest in O(E)

- considering half edges in E in each iteration $O(E + E/2 + E/4 + \cdots + 1) = O(E)$

As shown above, the time complexity of the Median-Finding MBP algorithm is O(n).

Pseudo-code

```
function MBP(graph G, weights w)
    E ← the set of edges of G
    if | E | = 1 then return E else
        A ← half edges in E whose weights are less than the median weight
        B ← E - A
        F ← forest of GB
        if F is a spanning tree then
            return MBP(GB,w)
        else
            return MBP ((GA)η, w)
```

8. Reference

1. Ellison, Nicole B. "Social network sites: Definition, history, and scholarship." Journal of ComputerMediated Communication 13.1 (2007): 210-230.

2. Olifer, Natalia. Computer networks. John Wiley & Sons, 2005.

3. Malpani, Navneet, and Jianer Chen. "A note on practical construction of maximum bandwidth paths." Information Processing Letters 83.3 (2002): 175-180.

4. Cormen, Thomas H. Introduction to algorithms. MIT press, 2009.