TEXAS A&M
UNIVERSITY

ANALYSIS OF ALGORITHMS

Course Project

Author: Kamalakshitha Aligeri

UIN: 128006765

Prof. Jianer Chen

April 25

Abstract

In this course project, two algorithms are implemented to compute the maximum bandwidth path, modified Dijkstra and Kruskal's algorithm. The algorithms are tested on two randomly generated graphs and test results are recorded.

We demonstrate Dijkstra with heap structure performs better for both sparse and dense graphs. Kruskal's algorithm is simpler, easier in implementation, more flexible but performs worse for dense graph.

Introduction

Given a source node s and a destination node t in a network G, in which each edge is associated with a weight w, construct a path from s to t in G whose bandwidth is maximized (the bandwidth of a path is equal to the minimum link-bandwidth that is weight over all links in the path).

Random Graph Generation

Networks can be shown with a graph in which every node in networks are vertices in graphs and every link in networks are edges in graphs. The project tests the algorithm on two types of random graphs:

1. The first graph has an average vertex degree of 6 and a pair of vertices that are adjacent is chosen randomly. (sparse graph). The average vertex of a degree means on an average each vertex is connected to 6 edges. Since it is an undirected graph number_of_edges is (average-vertex degree/2)*number_of_vertices
2. In the second graph, each vertex is adjacent to about 20% of other vertices, which are randomly chosen. (dense graph). After ensuring that the graph is connected by making an initial cycle, each pair of vertices are connected with a probability of 20%.

Heap Structure

A max-heap is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node. For the calculation of the maximum bandwidth path, Dijkstra uses a max-heap data structure to store fringe nodes.

Kruskal's algorithm uses the heap sort algorithm to sort the edges in non-increasing order.

Heap Structure Algorithms are:

Algorithm Max Heapfy

```
1: procedure MAX HEAPFY(A,i)
2:     l ← LEFT(i)
3:     r ← RIGHT(i)
4:     if l ≤ A.heap_size and A[l] > A[i] then
5:         laregest ← l
6:     else largest ← i
7:     if r ≤ A.heap_size and A[r] > A[largest] then
8:         laregest ← r
9:     if largest ≠ i then
10:        exchange A[i]with A[largest].
11:        MAX HEAPFY(A,largest).
```

Algorithm BUILD MAX HEAP

```
1: procedure BUILD MAX HEAP(A)
2:     A.heap_size ← A.length
3:     for i ← [A.lenght/2] to 1 do
4:         MAX HEAPFY(A,i).
```

Algorithm HEAP SORT

```
1: procedure HEAP SORT(A)
2:     BUILD MAX HEAP(A).
3:     for i ← A.lenght to 2 do
4:         exchange A[1] with A[i].
5:         A.heap_size ← A.heap_size − 1
6:         MAX HEAPFY(A,1).
```

Algorithm INSERT

```
1: procedure INSERT(A,key)
2:     A.heap_size ← A.heap_size + 1
3:     A[A.heap_size]key
4:     MAX HEAPFY(A,A.heap_size).
```

Algorithm HEAP MAX

```
1: procedure HEAP MAX(A)
2:     return A[1]
```

Routing Algorithms

Dijkstra

There are two versions of Dijkstra in this course project.

a. Simple (Without Heap)

In this version, the fringe nodes are just stored in a Linked List (Adjacency List). Time complexity is $O(n^2)$. The array dad[v] records the father of node v in the maximum bandwidth

tree and the array bw[v] records the bandwidth of the path from source node s to the node v in the maximum bandwidth tree.

b. Modified (With Heap)

In the modified Dijkstra's algorithm, the array dad[v] records the father of node v in the maximum bandwidth tree and the array bw[v] records the bandwidth of the path from source node s to the node v in the maximum bandwidth tree. The set of fringes are implemented by a max-heap data structure. Since the time complexity for insert and delete operation on a max-heap is O(logn), the total time complexity of Dijkstra's algorithm is O(mlogn), where m is the number of edges and n is the number of vertices in a graph G.

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)
```

Kruskal:

Kruskal's algorithm to find the maximum bandwidth path is based on a theorem that gives a relation between maximum bandwidth path and maximum spanning tree.

The theorem states that: Let G be a network, in which each link e has a bandwidth value b, let T be a maximum spanning tree in G (with respect to link bandwidth). Then for any two nodes s and t in G, the unique path P in T from s to t is a maximum bandwidth path from source s to a vertex t in G. Max Spanning tree is a spanning tree whose sum of edge's weight is the maximum. Its advantage is that once MST is constructed, we can find the path between any two ends in linear time.

Algorithm 7 MAKE SET

```
1: procedure MAKE SET(v)
2:     Dad[v] ← 0.
3:     Rank[w] ← 0.
```

Algorithm 8 FIND RANK

```
1: procedure FIND RANK(v)
2:     w ← v.
3:     while Dad[w] ≠ 0 do
4:         w ← Dad[w].
       return w
```

Algorithm UNION

```
1: procedure UNION(r1,r2)
2:     if Rank[r1] > Rank[r2] then
3:         Dad[r2] ← r1.
4:     else if Rank[r2] > Rank[r1] then
5:         Dad[r1] ← r2.
6:     else
7:         Dad[r1] ← r2.
8:         Rank[r2] + +.
```

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: {e1, e2, ...., em}.
3:     for each node v of G do
4:         MAKE SET(v).
5:     T ← ∅.
6:     for i = 1 to m do
7:         let ei = [ui, vi].
8:         r1 ← FIND(ui)
9:         r2 ← FIND(vi)
10:        if r1 ≠ r2 then
11:            add ei to T
12:            UNION(r1,r2)
```

We can modify the Kruskal's algorithm by sorting the edges in non-increasing order and then make use of union-find tree such that a sequence of m Makeset, Union and Find operations takes time O(mlog*n). Therefore, the time complexity of Kruskal's algorithm is s(m)+O(mlog*n), where s(m) = O(mlogn) is the time for sorting m elements.

To find the maximum bandwidth path between a pair of nodes source s and a vertex v, Kruskal's algorithm is combined with BFS (Breadth First Search). In BFS, to discover the path from source s to a vertex v, a dad array is used to store the parent element for each node that discovered it first.

Test Results:

Sparse Graph		
Dijkstra (without Heap)	**Dijkstra(with Heap)**	**Kruskal**
1.93509697914	0.132798910141	0.595219135284
1.88797187805	0.103161096573	0.601490020752
1.07702612877	0.0649120807648	0.60706782341
1.30424785614	0.0066180229187	0.602252960205
1.31696009636	0.0656268596649	0.606412172318
1.20779514313	0.0496277809143	0.58575797081
1.63203406334	0.0867540836334	0.590782880783
1.63214898109	0.0887620449066	0.591782808304
1.19095182419	0.0432720184326	0.588964939117
1.88243412971	0.102613925934	0.665091991425
1.74470305443	0.088907957077	0.578813791275
1.22865104675	0.000710010528564	0.61988902092
1.70652413368	0.087042093277	0.595973968506
0.759693145752	0.0378329753876	0.591243982315
1.82567119598	0.100995779037	0.592422008514
0.360714912415	0.0661211013794	0.596405982971
0.973249912262	0.0752868652344	0.588301181793
0.228778123856	0.0367748737335	0.592617988586
1.92604899406	0.111702919006	0.600602865219
0.763195991516	0.0979809761047	0.598325967789
0.0400140285492	0.0653581619263	0.611168861389
1.0854241848	0.0445899963379	0.594657897949
0.0298290252686	0.0318379402161	0.59890794754
0.383954048157	0.0394630432129	0.595320940018
0.0423641204834	0.032534122467	0.594056129456
Average: 1.1266193198846	Average: 0.066451425552347	Average: 0.59934124946592

Dense		
Dijkstra (without Heap)	**Dijkstra(with Heap)**	**Kruskal**
2.96985006332	1.20294594765	141.879948139
1.12971591949	1.06386780739	142.893793106
0.107562065125	0.476558923721	148.381335974
4.90294504166	1.7627530098	143.371143103
3.05009293556	1.15530395508	145.090692997
2.38339591026	1.6539850235	141.489309788
3.51530480385	0.708363056183	143.56873703
0.424914836884	2.00800299644	144.337137938
3.84247422218	0.538542985916	144.512848139
4.70666003227	0.475757837296	146.018211126
3.60088205338	1.49399399757	156.471145868
4.73853302002	0.708343029022	145.530344009

0.771317958832 3.34556078911 1.0708398819 4.97044801712 4.45124983788 4.5774641037 3.21909809113 0.652168035507 2.80651497841 2.53509092331 1.98870396614 4.28087902069 0.740937948227	0.954060077667 1.43795204163 1.20822811127 2.14316010475 1.48650789261 1.6368970871 2.1830868721 2.04632019997 1.27707099915 1.65089511871 1.44702601433 1.17584204674 1.64133906364	145.280456066 146.082574129 144.772565126 163.656599045 149.818802118 150.020639896 150.792756081 149.296576023 150.505810976 152.147017002 150.761084795 152.570994139 151.306718826
Average: 2.8313041782382	Average: 1.3414721679694	Average: 148.02228965756

Analysis

Algorithm	**Sparse Graph**	**Dense Graph**
1. Dijkstra(without heap)	1.1266193198846	2.8313041782382
2. Dijkstra(with heap)	0.066451425552347	1.3414721679694
3. Kruskal's	0.59934124946592	148.02228965756

For both sparse and dense graphs, Dijkstra with heap performances better than the other two algorithms.

From the above results Kruskal's performs worse for a dense graph.