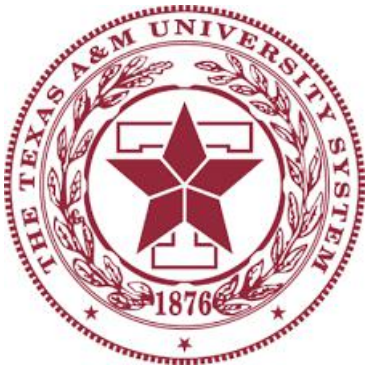


Dept. Computer Science & Engineering

CSCE-629

Searching For Maximal Bandwidth

Submitted **To:**
Jianer Chen
Computer Science &
Engineering Department

Submitted **By :**
Chen Chen
625008676

Contents

1 Abstract

The path of Maximal Bandwidth is an important topic in Computer Science and Electrical Engineering. Usually, for a specific graph, which consists of many vertices and undirected edges between 2 vertices with arbitary weight, we may be able to find different paths from one source vertex to terminal we choose. However, since the bandwidth of one path depend on its smallest weight, and many path can share that edge, which make the path with the maximal bandwidth not be unique. Finding such a maximal bandwidth path would be useful for graph researches and analysis of signal.

We can use some typical algorithm to find the Maximal Bandwidth path, such like Dijsktra and Kruskal. Besides the use of algorithm, we can select specific data structure to enhance the efficiency of our program. In this project, I'm going to use Heap sort to achieve Dijsktra and Kruskal algorithm in order to find the path we want in random graphs we create.

2 Introduction

The main tasks in this project includes the creation of two kinds of random graph, designing the heap structure and applying Dijsktra, Kruskal to achieve finding the Maximal Bandwidth path.

According to the number of vertices and edges, we can classify graphs into *Dense Graph* and *Sparse Graph*. Usually we can use A*djacent List* to store the Sparse Graph, while using *Adjacent Matrix* to store Dense Graph. In C++, we can design a structure to realize the Adjacent List, and use two-dimension array to achieve matrix.

A heap is a specialized tree-based data structure that satisfies the heap property: If A is a parent node of B, then the key (value) of node A is ordered with respect to the key of node B with the same ordering applying across heap. We can classify heap into max-heap and min-heap. In this project, I decided to use an array to represent the heap, and develop *Insert, Delete* and *Max* functions of heap.

Basically, the Dijsktra and Kruskal are used for finding the shortest path between two vertices. However, we can change some steps in these algorithms to finish the goal for finding the Maximal Bandwidth Path. Both of these two algorithms contains sorting algorithm, here we can use heap sort to get the value we want, which means that the structure of heap would influence efficiency of algorithms.

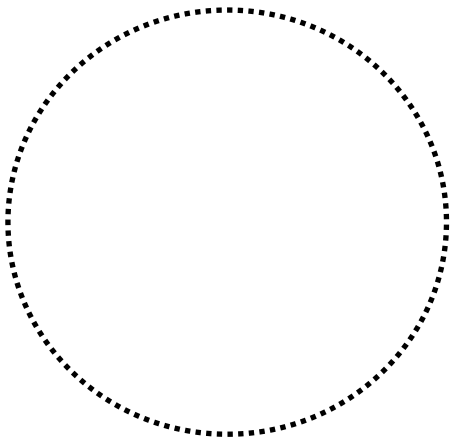
After finishing the program, generating 5 different graphs for Dense Graph and Sparse Graph, respectively. For each pair of graph, selecting 5 pairs of source and terminal vertices randomly, and finding the Maximal Bandwidth path between them.

3 Random Graph Generation

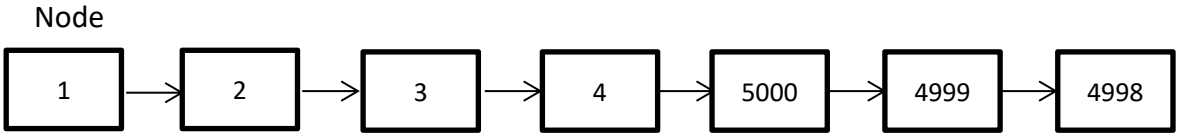
Before we apply and verify our algorithms, we need to create 2 graphs randomly.

Graph 1: every vertex has degree exactly 6.

The degree of a vertex in undirected graph means that for each vertex, there are only specific number of edges between it and the other vertices. In order to finish that, I decide to use the 'Shuffle' , which means designing the graph before arranging the vertices.



Imagine that putting all of the 5000 vertices around a circle, each vertex connect with 6 vertices, 3 of them behind it and 3 of them after it. For instance, vertex 1 can connect with vertex 2, 3, 4 and vertex 4998, 4999 and 5000. In this way, we can ensure that each vertex is able to reach degree 6, also, can always find a way from 1 vertex to any one of other 4999 vertices. After finishing the basic structure, we can start 'shuffle', which means changing one vertex with the other one, such like we exchange the position of vertex 1 and vertex 2500. Randomly selecting a pair of different vertex. After 2500 times, I got an arbitary graph with 5000 vertices and defree equal to 6.



This is the typical structure of Adjacent List. In C++, we can define Adjacent List like this:

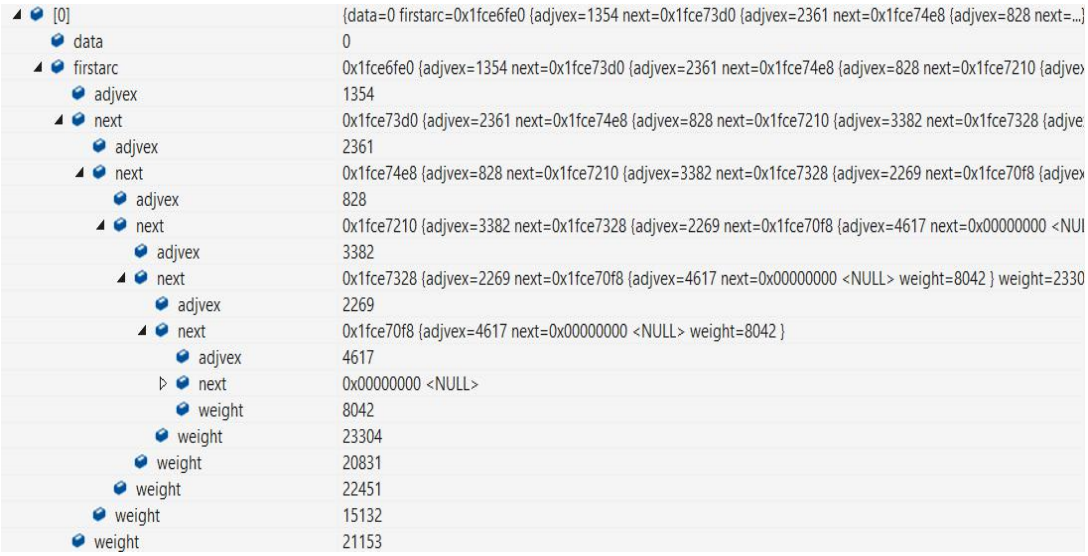
```
typedef struct ArcNode
[
    int             adjvex;       //vertex name
    struct ArcNode  *next;        //next vertex
    int             weight;       //weight
]ArcNode;
typedef struct VNode
[
    int       data;         //source vertex
    ArcNode   *firstarc;    //point to the first vertex
]VNode, AdjList[vn];
typedef struct
[
    AdjList   vertices;
]ALGraph;
```

Structure of adjacent list

When we are applying the 'Shuffle' , we have to find all of the same vertex in this Adjacent List. For instance, if we want to switch 1 to 2500, we have to change all of the No.2500 vertex in the list, also change No.1 vertex to No.2500. But we can not change the searching index at that time since it would mislead us. During the process of shuffling, we have to record the pair of changing vertices. After finishing the shuffling, substituting the index with the information we record during the process.

Now, we have gotten a graph whose degree is 6. However, we may not be able to find a specific vertex and its information according to its index, since Graph[1] may not represent vertex No.1. In order to simplify the process after generating graphs, re-organizing the graphs which could make the Adjacent List more convenient to use in other algorithms.

Structure of searching array



Structure of vertex list (degree =6)

Next step is to generating random weights on each edge. The most important thing is that we have to ensure the edge (s,t) and (t,s) share the same value of weight. Here I used head-in method in my adjacent list. To doing so, I used pointer to find terminal vertex when I visited the source vertex.

Using random seed based on time to generate random number.

```
33      srand((unsigned)time(NULL));
```

Random seed

The random function is applied for generating a pair of exchange vertex and weight. For the random vertex, I need to control the scope of these number from 1 to 5000, and each pair be selected should be marked as a bool value. For weights , it can just be any integer. (swaparray is the bool array to mark vertices' situation)

```
100             k1 = rand() % vn;
101             k2 = rand() % vn;
102                 while (swapArray[k1] == true || swapArray[k2] == true)
103                 [
104                     k1 = swapArray[k1] == true ? rand() % vn : k1;
105                     k2 = swapArray[k2] == true ? rand() % vn : k2;
106                 ]
```

Finding the exchange pair

Graph 2: Each vertex has edges going to about 20% of the other vertices.

For the dense graph Graph 2, we can use adjacent matrix to represent it. Since the number of each dimension in this matrix is 5000, we have to avoid the stack overflow. Therefore, I declared the matrix (2-D array) as a global variable.

It is easy to put weight into this dense graph. Matrix[s][t] denote an edge which starts from s to t, the number in matrix is the weight of that edge. Only need to focus on half of the matrix because matrix[s][t]=matrix[t][s]. In this graph, matrix[s][s]=0, which means every vertex go to itself would always take 0 steps. If there are two vertices never meet each other, I set the number in matrix as INT_MAX, which means we can not reach t from s.

As for the probability 20%, I generated a number from 1 to 10 randomly. If this number is less than 2, then return 1, otherwise, return 0. After that, using this function times a random integer, then arranging the result be the weight of edge.

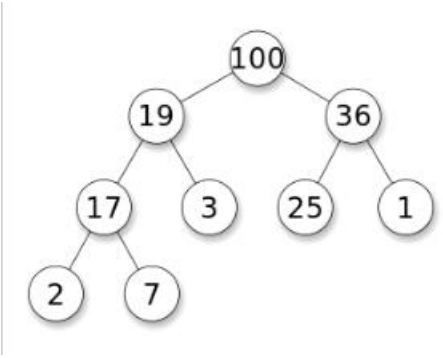
```
175     int probability()//20%
176     [
177         int m = rand()%10;
178         if (m < 2)
179             return 1;
180         else
181             return 0;
182
183     ]
```

20% probability

4 Heap Structure

There are many sorting algorithm, such like quick sorting and insert sorting. Heap sorting is a kind of sorting depending on the heap structure. Here, I put all of the element into an array, and use a counter to calculate the size of heap.

For an array heap={100 19 36 17 3 25 1 2 7}, the heap structure would be:



Heap Structure

The property of max heap is that heap[i] would always be larger than heap[2i] and heap[2i+1], however we can not know whether heap[2i] is larger than heap[2i+1] or not. After we adjust the max heap, the first element in this heap would be the largest one among this array.

Besides getting the maximal value, insert and delete operation are also important. When applying Insertion algorithm, putting a new element in the end of the current heap (a sub array), then the counter++. After that, I got a new heap with size = (|heap|+1) , and we have to adjust the heap again to find the proper position of this new element. As for deletion, switching the element we want to delete with the last element in heap, then counter--. In this (|heap|-1) sub array, adjusting it again.

Start adjusting the heap from at [|heap|/2], from bottom to top. The comparing condition can be changed according to different situation, such like bandwidth and weight.

5 Routing Algorithm

(1) Dijsktra's Algorithm

Dijsktra's Algorithm is usually used for finding the shortest path in a graph between 2 vertex.

```
Dijkstra(G, s, t)
1. For each vertex v do
        status[v]=un-seen;
2. status[s]=in-tree;
   Dad[s]=-1;
   bw[s]=INT_MAX;
3. For each edge [s,w] do
        status[w]=fringe;
        Dad[w]=s;
        bw[w]=weight[s,w];
4. While there are fringe do
        pick a fringe v of maximal bw[v]
        status[v]=intree;
        For each edge [v,w] do
                if status[w]=un-seen;
                then status[w]=fringe;
                     Dad[w]=v;
                     bw[w]=min{bw[v],weight[v,w]};
                else if status[w]=fringe and bw[w]<min{bw[v],weight[v,w]};
                then Dad[w]=v;
                     bw[w]=min{bw[v],weight[v,w]};
5. Output (Dad[1...n])
```

Normally, Dijsktra algorithm would takes O(V^2).

To achieve this algorithm, I used several array and vector to denote different information. (vn is the number of vertex)

Status[vn]: Recording each vertex's status. *Unseen* means we have not visited it yet. *Fringe* means we may consider to mark this vertex as the path. *In-tree* means the vertex has been in the max bandwidth path.

Dad[vn]: Recording the parent node of each vertex. For instance, Dad[w]=v denotes the parent vertex of w in this path is v.

Bw[vn]: Recording the maximal bandwidth from source to each vertex.

Vector<int> storefringe: a vector used for storing the fringe vertex. Vector is an efficient data structure for inserting and deleting.

By doing this algorithm, we can output the Dad array, from which getting the max bandwidth path.

(2) Dijsktra's Algorithm (with heap)

```
Dijkstra(G, s, t)
1. For each vertex v do
        status[v]=un-seen;
2. status[s]=in-tree;
  Dad[s]=-1;
  bw[s]=INT_MAX;
3. Heap H=NULL;
4. For each edge [s,w] do
        status[w]=fringe;
        Dad[w]=s;
        bw[w]=weight[s,w];
        Insert(H,w);
5. While H != NULL
        pick a fringe v of maximal bw[v], v=Max(H);
        Delete(H,v);
        status[v]=intree;
        For each edge [v,w] do
                if status[w]=un-seen;
                then status[w]=fringe;
                     Dad[w]=v;
                     bw[w]=min{bw[v],weight[v,w]};
                     Insert(H,w);
                else if status[w]=fringe and bw[w]<min{bw[v],weight[v,w]};
                then Delete(H,w);
                     Dad[w]=v;
                     bw[w]=min{bw[v],weight[v,w]};
                     Insert(H,w);
6. Output (Dad[1...n])
```

Here, Dijsktra's algorithm with heap would takes O($(m+n)\log n$). m is the number of edges and n is the number of vertex.

(3) Kruskal Algorithm (Heap sort)

We can apply Kruskal algorithm for finding the max-spanning tree.

```
Kruscal(G, s, t)
1. Sort the edges increasingly
        E1, E2, E3. . . . Em
2. max-spanning tree T=NULL
3. for i=1 to m do
        let Ei=[ui, vi]
        r1=Find(ui);
        r2=Find(vi);
        if r1 != r2, then
                T=T+Ei;
                Union(r1, r2);
4. make T a new graph
5. Dad[1. . . vn]=BFS(T, s, t)
```

Find(v): Find the parent node of vertex v.

Union(r1,r2): For 2 pieces of sub-graph, combine them together

Kruskal Algorithm usually takes $O((E+V)\log V)$

```
MakeSet(Vi)
1. D[Vi]=0;
2. rank[Vi]=0;

Find(Vi)
1. w=Vi
2. while D[w] != 0 do
        w->stack s
        w=D[w];
3. while s != NULL
        u<-stack s;
        D[u]=w;
4. return w

Union(r1, r2)
If (rank[r1] > rank[r2])
        then D[r2]=r1;
else if (rank[r1] < rank[r2])
        then D[r1]=r2;
     else //rank[r1]==rank[r2]
        D[r2]=r1;
        rank[r1]++;
```

D[vn]: Parent array for each vertex.

Rank[v]: the length of path from vertex to its source node.

In Find(v), we can use stack to store all of the vertex in a path, and set their parent be the source vertex. In this way, we could eliminate redundant operations.

However, after the Kruskal, we can only get a Max Spanning Tree which contains several edges. Using these edges to re-build a graph, and applying BFS on this graph to find the path of maximal bandwidth from s to t.

```
BFS(G, s, t)
1. For each vertex i do
        color[i]=white;
2. color[s]=gray
   Dad[s]=-1;
   queue.push(s);
3. while queue != NULL
   t=queue.pop();
   for each vertex w that near t do
        if(color[w]==white)
                color[w]=gray;
                Dad[w]=t;
                queue.push(w);
4. color[s]=black;
5. return Dad[1...vn]
```

In Kruskal Algorithm, I used a structure array to store the information of each edges, including their weight, source vertex and terminal vertex.

```
4   typedef struct Edge
5   {
6       int weight;
7       int s;
8       int t;
9   }edge;
```

Information of an edge

In BFS or DFS, we have to mark the status of each vertex, thus I used the color[vn]. *White* means not being visited yet, *gray* means being processing while *black* means having be visited.

Heap sorting here is aimed to finding the edge with maximal weight each iteration. For graph 1, there are 15,000 edges, while the expected number of edges in graph 2 may be 2.5 millions. Therefore, it could take longer time in Kruskal to find the MST for a Dense graph.

6 Testing and Conclusion

To verify the program, randomly generating 5 graphs for Graph 1 and Graph 2, respectively. For each pair of graphs, testing 5 pairs of source and terminal vertex. Recording the running time for each algorithm.

In Graph 1, because of the 'Shuffle' theory, there would always be a path from one vertex to any other one. For Graph 2, I have to add a path from source to terminal which across all of the other vertices in order to make sure there is a path between them.

After returning the Dad[1...n] from each algorithms, visiting all of these array from source to terminal, which could help to trace the Maximal Bandwidth path. Also we can check the bandwidth array to verify these algorithms.

Here are the testing result:

Test 1	**Graph 1 (millisecond)**			**Graph 2 (second)**		
Function / **(s,t)**	**Dijsktra**	**Dijsktra(heap)**	**Krusucal**	**Dijsktra**	**Dijsktra(heap)**	**Kruskal**
(1598,4456)	141ms	10ms	299ms	4.689s	0.856s	53.178s
(489,876)	134ms	10ms	295ms	4.646s	0.866s	53.095s
(985,654)	237ms	9ms	297ms	4.643s	0.872s	53.471s
(12,8)	231ms	10ms	316ms	4.585s	0.871s	53.029s
(4862,3215)	240ms	11ms	294ms	4.605s	0.855s	53.058s

Test 2	Graph 1 (millisecond)			Graph 2 (second)		
Function (s,t)	Dijsktra	Dijsktra(heap)	Krusucal	Dijsktra	Dijsktra(heap)	Kruskal
(369,489)	240ms	10ms	292ms	4.878s	0.853s	53.504s
(756,86)	229ms	10ms	296ms	4.865s	0.861s	53.761s
(88,3354)	240ms	10ms	295ms	4.988s	0.858s	53.850s
(2569,471)	214ms	9ms	295ms	4.874s	0.850s	53.878s
(1321,627)	301ms	10ms	293ms	4.835s	0.858s	53.526s

Test 3	Graph 1 (millisecond)			Graph 2 (second)		
Function (s,t)	Dijsktra	Dijsktra(heap)	Krusucal	Dijsktra	Dijsktra(heap)	Kruskal
(3259,343)	319ms	9ms	294ms	4.382s	0.865s	55.951s
(63,1994)	318ms	9ms	296ms	4.401s	0.863s	55.485s
(1120,215)	257ms	11ms	298ms	4.435s	0.856s	55.865s
(997,106)	393ms	9ms	300ms	4.390s	0.861s	55.549s
(519,23)	257ms	10ms	296ms	4.403s	0.862s	55.489s

Test 4	Graph 1 (millisecond)			Graph 2 (second)		
Function (s,t)	Dijsktra	Dijsktra(heap)	Krusucal	Dijsktra	Dijsktra(heap)	Kruskal
(883,1988)	306ms	10ms	295ms	4.420s	0.853s	53.419s
(125,2245)	355ms	11ms	295ms	4.419s	0.876s	53.360s
(1993,236)	337ms	9ms	301ms	4.403s	0.865s	53.685s
(160,7)	261ms	9ms	300ms	4.399s	0.846s	53.738s
(172,29)	364ms	10ms	300ms	4.465s	0.863s	53.749s

Test 5	Graph 1 (millisecond)			Graph 2 (second)		
Function (s,t)	**Dijsktra**	**Dijsktra(heap)**	**Krusucal**	**Dijsktra**	**Dijsktra(heap)**	**Kruskal**
(27,748)	221ms	10ms	293ms	4.557s	0.859s	53.158s
(55,257)	212ms	10ms	295ms	4.571s	0.852s	52.63s
(3985,639)	316ms	10ms	292ms	4.574s	0.867s	52.993s
(4877,485)	300ms	10ms	295ms	4.597s	0.863s	52.928s
(853,3751)	343ms	10ms	296	4.566s	0.859s	52.682s

As we can see in the testing result, for a sparse graph Graph 1, the running time of each algorithm would be faster than any other for a dense graph Graph 2. Specifically, Dijsktra Algorithm with heap is the fastest since the efficiency of heap sort. In the normal Dijsktra, need to go through all of the fringes in *storefringe* vector, and find the maximal bandwidth, which would take much more time than heap.

For a dense graph Graph 2, since its complex structure, there are nearly 2.5 millions of edges in this graph. When we are using the Kruskal, I have to push all of these edges into the heap (an array and the heap is only a part of it. Controlling the length of this sub array). After that, I have to find 4999 edges with larger bandwidth among all of these edges, that's why Kruskal for graph 2 can take lots of time.

7 Improving

As I conclude in the testing result, Kruskal for graph 2 takes over 50 seconds, which is much longer than any others. That may be resulted by the bad structure of heap and functions of heap operation. Instead of using an extremely long array (over 2.5 millions), we can try an arraylist which is easy for insertion and deletion to reach the target we want. By doing so, it can save lots of time in insertion and deletion, which could be used many time in algorithms