Project: Maximum Bandwidth Path

TING-WEI SU (UIN:231003210)
willytwsu@tamu.edu
CSCE 629 Analysis of Algorithm

1. Introduction

In our class discussion, we discussed a problem with finding the maximum bandwidth path from a source vertex to a destination vertex in a given undirect weighted graph. The bandwidth of the path is defined by the smallest edge of a selected path between the source and destination. This algorithm is well-known in all kinds of network communication systems such as routing protocols and network traffic control. Two main famous algorithms have been discussed along with the suitable data structure. Using Dijkstra without heap data structure, Dijkstra with heap data structure, and Kruskal's algorithm. Dijkstra is famously used in finding the shortest path problem in a graph, and we have a slightly adjust to apply Dijkstra to our problem. In the following section, I will address the implementation details of the graph generation, the max-bandwidth-path algorithm approach and the result I gather. At last, some future optimization to make the program runs even faster than the current version is provided.

2. Design and Implementation

Applying modern software engineering techniques by using objected oriented programming, such as the c++ class, could not only extend the code reusability but also the robustness of the class private property. The ultimate idea is to separate each feature, such as the generation of graphs, the finding of the maximum bandwidth path, and the data structure applied to the algorithm into different classes. The benefit of this structure is to separate the graph generation from the max-bandwidth algorithm. Thus, once we generate the graph, we could pass the graph to the algorithm object, which is the MaxBWDijkstra or MaxBWKruskal class. Then authorize the class to read data from the graph property and apply any algorithm on the graph to achieve our goal. Therefore, we will need three types of files for these three main classes. And by adding a makefile I have made the compilation easier in the future. The version of the c++ compiler I use is c++11.

Class structure

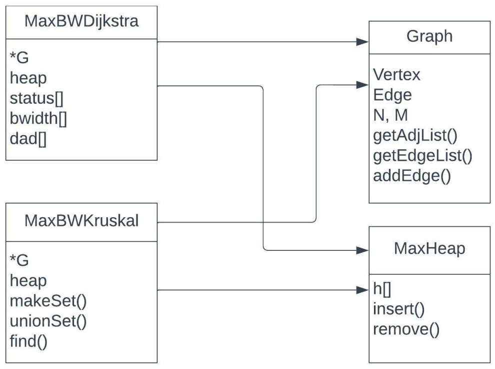

Figure 1: Class hierarchy

Dijkstra algorithm without heap

Dijkstra's algorithm is based on a greedy approach to find the locally optimal solution. In this implementation, I use three arrays to keep track of the vertex status, the parent of the vertex and the bwidth of the vertex. The status is an enum type of three states which are unseen, fringer and intree. Whenever a source vertex is selected, we will mark the vertex as intree. Then find its neighbor vertex and marked those as fringer. Then use a routine to continue to check the vertex bwidth value of the vertex neighboring to the fringer vertex to be the smallest of all.

Dijkstra algorithm with heap

The heap data structure can be used to store the vertex in non-increasing order. However, we have to implement insertion and deletion by ourselves. The insert operation could be divided into two actions. The first is to swap the last element in the heap with the element intended for deletion. The second is max_heapify_down which will reconstruct the elements' position in order to maintain heap property after the swap operation. The time complexity for a heap structure is O(log n) to heapify the structure either up or down. For insertion, we attach the new element at the end of the heap, then do the max_heapify_down to maintain the heap property.

```
class MaxHeap
{
private:
...
    void swap(struct node &a, struct node &b);
    void max_heapify_up(int i);
    void max_heapify_down(int i);
public:
...
    // Insert new element to heap
    void insertNode(int idx, int bw);
    // Remove an element from heap
    void removeMax();
};
```

Kruskal algorithm

Kruskal approach required us to first construct the maximum spanning tree in the graph then do a DFS explore on the MST to find the maximum bandwidth path. Therefore, in order to construct the MST, a Union-Find-Makeset model is require to handle the set operation for we view the each vertex as independent set at the beginning. Constantly find the vertex from the largest bwidth, and do the union with neighbor vertex if not already added into the set. When all vertex united in a single set, the exploration has done, we have successfully construct the maximum spanning tree.

```
// Disjoint set operation:
// Create a set whose only member is x
void makeSet(int x);
// Unites the two set that contain x and y
// Say S = Sx U Sy
void unionSet(int x, int y);
// Find the root of x
int find(int x);
```

Graph generation

The goal of our graph generation is to compose an undirected weighted graph with vertices and edges. For each vertex, I use an array to store each vertex key value. For each edge, I use the adjacent list to store the neighbor vertex and are pointed by the head vertex array. When we are to find a vertex neighbor, we could use the head array to locate the vertex's key value and then extract the linked list attached from the head pointer to find the adjacency list vertices.
In addition, the randomness of the graph is handled by the random number generator in the standard library c++ provides. I separate the generation into two stages to ensure that the graph is not only randomly enough but also connected. In the first stage, to ensure connectivity we will iterate the assigning routine for a specific amount of edges. And each newly generated vertex is connected to the previous vertex so that we could ensure that the (i+1)th edge is connected to i-th vertex. Once we generate a certain amount of edges that will connect each vertex like a chain, I will attach the last vertex to the beginning vertex to form a loop. Several functions including a isEdgeValid and connectVertex are provided to check whether the new edge is valid or not. And if the edge generated has a duplicate value as the previous generation, I will re-generate again for the same index. If the edge is valid, I will update the edge information to the head array.
The second stage of graph generation is to choose the rest of the not-connected vertices with the already connected vertices. Two functions to find the alone vertex and to find the connected vertex is provided. The searchAlonedVertex vertex function will linearly search for vertices whose adjacency list is still empty. The searchConnectedVertex will randomly pick a vertex in the connected list to be selected as the next dest of the edge.

```
// Function to search for alone vertex
// By alone, I mean those vertex that has not been connected
```

```
int searchAlonedVertex();
// Function to search for connected vertex
int searchConnectedVertex();
```

3. Test and Result

In the result table, we could see that Dijkstra with heap is faster on finding the max-bw path with a better time complexity. However. Kruskal can do even better in some of the case. While in the denser graph, Kruskal did not perfom so well.
The following result is been compiled and tested on a Linux machine in the TAMU ECE Apollo Server.
The specification of the linux machine:

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    2
Core(s) per socket:    10
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
Stepping:              4
CPU MHz:               3099.926
CPU max MHz:           3600.0000
CPU min MHz:           1200.0000
BogoMIPS:              5599.70
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              25600K
```

	Dijkstra w/o heap	Dijkstra w/ heap	Kruskal
Type 1	14729.6	621.972	20706.7
Type 2	21047.1	18133.1	847501

	Dijkstra w/o heap	Dijkstra w/ heap	Kruskal
Graph0	14829.7	296.495	20668.4
Graph1	14646.8	1320.72	19929.8
Graph2	14158.2	241.053	19996.8
Graph3	15239.4	871.659	22435.4
Graph4	14774.1	379.927	20503

	Dijkstra w/o heap	Dijkstra w/ heap	Kruskal
Graph0	24599.9	22802	1082361
Graph1	18328.5	16347.6	1450445
Graph2	17528.3	12294.5	606239
Graph3	24604.2	23006.7	545135
Graph4	20174.6	16214.6	553329

```
Dijkstra w/o heap on type 1 graph (s, t) = (81, 1149)
Bandwidth: 84
Time: 15356.2 us.

Dijkstra w/ heap on type 1 graph (s, t) = (81, 1149)
Bandwidth: 12
Time: 386.191 us.

Kruskal on type 1 graph (s, t) = (81, 1149)
Bandwidth: 1
Time: 20393.2 us.

Dijkstra w/o heap on type 2 graph (s, t) = (81, 1149)
No element in heap
Bandwidth: 58
Time: 29701.3 us.

Dijkstra w/ heap on type 2 graph (s, t) = (81, 1149)
No element in heap
Bandwidth: 58
Time: 28867.2 us.

Kruskal on type 2 graph (s, t) = (81, 1149)
Bandwidth: 3
Time: 632555 us.
```

4. Future Optimization

Graph generation speedup

The current implementation generate the graph in a truly randomnes means, which I also create the randomly selection for every edges after the connected stages, i.e. after all vertex is connected. This will create too much run time, if the random number generator keep finding a set that has been generate before. The waste of regenerate the new edge is a burden. Therefore, a graph matching to select which vertex is to connect at the last stage of the graph generatioon is an means to avoid the conflict time spend on previous approach.

Better data structure

A 2-3 tree might be a good idea on optimizing largest bandwidth fringer extraction. The 2-3 tree can extract the maximum in $O(logn)$.

Applied different approach according to the type of graph

We know that the time complexity of the Dijkstra without heap is $O(n^2)$, while Dijkstra with heap is $O(mlogn)$. In our experiment, the result shows that Dijkstra with heap has a better performance of the two even on a denser graph. However, it would not always be the case. For example, in the type 2 graph, if m is enourmously larger enough than n to create a larger factor leading to the $O(mlogn)$ is worst than $O(n^2)$ on the dense graph. In the algorithm field, there is not such best algorithm that could apply to every problems in the world. We have to find the suitable alrogithm to different scanario. We could create a graph evaluation first by comparing the m and n factor before applying the algorithm. We could achieve $minO(n^2), O(mlogn)$ by combining the both algorithm.