CSCE-629 Course Project

Ho Lee
725007592

Objective

We are to utilize algorithms and data structures to solve maximal bandwidth path problems which is a very practical network optimization problem among researches in computer science and engineering. Here we have three approach to implement the max-bandwidth problem :

1. Modification of Dijkstra's shortest path with the use of heap structure for fringes.
2. Modification of Dijkstra's shortest path without the use of heap structure for fringes.
3. Modification of Kruskal's spanning tree algorithm which sort edges by heapsort.

Enable to examine the performance of each approach with different inputs, we therefore designed a program to generate random graphs for inputs. After the graphs are generated, the approaches could be examined fairly.

Implementation

I will break into several implementation steps/parts to better discuss it more detailed. There are of course close relations with each other which I will deep into it later.

1. Linked List Structure
2. Random Graph Generation
3. Heap Structure
4. Three routing algorithm approach we mentioned earlier

Linked List Structure

The linked list structures are defined in both "`linkgraph.hpp`" and "`linked_list.hpp`" which the former is singly linked and the later doubly linked. I use linked lists to mainly store edges of graph in it so that I could check whether a given vertex have an edge toward another vertex and what's the edge weight. For instance, I declare an array of class `list` like " `list * edge_list[TOTAL_VERTICES]` " which TOTAL_VERTICES is set to be equal to the total vertices in the graph. From this array I could easily access to the given vertex `v` and transverse through the list to find or update content. Now I am going to elaborate the functions and structures within the class.

The list is built on the `struct vertices` which includes pointers to previous (only for doubly linked list) and next elements adjacent to the current one in the list, and the data in the structure are the vertex number and this edge's weight. The class functions are :

- `list()` : The constructor of the class call while creating a new class object.
- `newvertex(v,w)` : Add the new given vertex to the tail of the list.
- `find_edge(v)` : Find whether an edge exists, if found return its edge weight.
- `display()` & `display_with_weight()` : Print out the whole list.
- `Delete(v)` : Delete the element in the list contains the given vertex.
- `Update(v,w)` : Update the element of the given vertex with new weight data.
- `list_head()` & `list_tail()` : Return the head/tail of the current list for future use.

Random Graph Generation

After the linked list functionality is created, I can then generate graphs with fixed total vertices number and randomly picked edges weights. Here I fix the total number of vertices to 5000. Larger amount of vertices and edges are required to see performance differences when we start to use these graphs as reference for the algorithm testing in future.

Here I generate two kinds of graph, sparse and dense, to better analyze the performances of our different algorithm approaches. The sparse graphs have average vertex degree 8, i.e. E=4N. The dense graphs have each vertex adjacent to about 20% of the other vertices. The generation of the graphs are generated by "`linkgraph1.cpp`" for sparse graphs and "`linkgraph2.cpp`" for dense graphs. The only difference between these two files are the probabilities to decide whether or not to add the vertex. Since maximal number of possible edges is bound to M = 0.5*(5000)*5000 = 12,500,000, therefore the probability of adding edges to sparse graph is around 0.5*(E/M)*100% = 0.5*(4*5000/M)*100% = 0.8% and for dense graph's probability is obviously around 20%*0.5 = 10%.

The reason I mentioned "around" is because: First, I have to make sure the graph is connected, so I connect the vertices one after another of total 4999 edges to make connected graphs. Second, the graphs are undirected graphs, therefore when adding the an edge, both vertices have adjacent to a new vertex simultaneously. I tweak the probability a little bit to generate graphs with more accurate edges by trial and error since the hidden calculation is more complex than expected.

Heap Structure

Similar to "`linkgraph.hpp`", "`heap.hpp`" describes the class `heap` and the class functions and operations in order to manage the heap structure. The structure which builds up the heap is simply two arrays `D[]` and `H[]` which stores the edge weights and the index within the heap array. The class functions are :

- `heap()` : The constructor of the class call while creating a new class object.
- `extract_max()` : Return the vertex number and weight of the max element in heap.
- `Insert(v,w)` : Add the new given vertex to the heap then rearrange the heap.
- `Delete(v)` : Delete the element in the list contains the given vertex.
- `heapify(bug)` : Rearrange the heap of to follow the heap rule when with a bug.
- `delete_max()` : Delete the max (first) element in the heap then rearrange the heap.
- `Display_heap:` Print out the whole heap in a decreasing order for verifiy.
- `update(v,w)` : Update the element of the given vertex with new weight data.
- `heap_size` : Return the current heap size for future use.

First Approach: Modified Dijkstra's Algorithm (without heap structure for fringe)

I implemented the Dijkstra's algorithm for maximal bandwidth path as described in class, simply put, every node has three statuses: unseen, fringe, and in-tree. The algorithm runs until the target node's status is in-tree.

Here I store the entire input graph into the linked list array with the class I implement before. Once a node is visited, the status of the node changes from unseen to fringe. I store all the fringes' (visited node that are not yet in-tree) information in multiple arrays indexed by their node numbers which stores their labels indicating their current bandwidth from the source, parent node with the max bandwidth path and their statuses. Whenever a maximal fringe is required I scan through the whole array to find the one with the maximal label and return the index/node number of that node. The update of the fringe's label after finding better bandwidth path is simply indexing the array by its node number and change the stored label.

Second Approach: Modified Dijkstra's Algorithm (with heap for fringes)

This implementation is very similar with first approach except now I store the fringe list in the max heap structure which I have described and implemented. Now when I need to find the node in the fringe with maximal label, I simply use the heap class function `extract_max()` to require the needed contents. However, the updating for labels required to scan through the heap array to find the proper element to update.

Third Approach: Modified Kruskal's Algorithm for Maximum Spanning Tree

I also implemented the algorithm the way described in class. First, I created a Maximum Spanning Tree by Kruskal's algorithm and then used the Depth First Search to find the path from source to target in the Maximum Spanning Tree.

The implementation of Kruskal's algorithm required to sort all edges by their weights in a decreasing other Maximum Spanning Tree. For each edges we pick the one with the largest weight by `extract_max()` and added into the Maximum Spanning Tree if adding it won't make a cycle in the tree with three subroutines `find_root(v)`, `merge_root`, and `cycle_detect` which functionalities of these three are described in class. I do implemented the find path compression method to speed up the future look-ups.

After the Maximum Spanning Tree is create, I run the Depth First Search also described in class to find a path from source to target and return the minimal edge weight along this path as the maximal bandwidth.

Test Result

There are total of ten graphs for testing, five are sparse graphs and five are dense graphs as required. All these graphs are to be as input graph to the three algorithm approaches described earlier to find a maximal bandwidth path with five source and target pair for each graph. The five source-target pairs are (228,3741), (1337,3832), (2049,1478), (2518,4705), and (4305, 392). The table below show the test results for each graph running under each approach with the average runtime of all five source-target pair :

Table1. average runtime of a graph input with multiple source-target pair(5)

Run Time (s)	**# of edges**	**Dijkstra**	**Dijkstra w/heap**	**Kruskal**
Dense (20% adjacent)	2,499,993	1.314884	1.338139	0.885201
	2,500,959	1.299181	1.384874	0.882195
	2,502,163	1.412508	1.523828	0.907072
	2,501,118	1.354902	1.519996	0.894361
	2,499,648	1.323593	1.483114	0.901031
Sparse (E = 4N)	20,151	0.061259	0.017338	0.018079
	19,981	0.067073	0.023906	0.019834
	20,028	0.052420	0.017740	0.020176
	20,060	0.064428	0.020482	0.019696
	20,144	0.065222	0.022091	0.019309

Table2. average runtime of a source-target pair with multiple input graphs(5)

Run Time (s)	**(S, T)**	**Dijkstra**	**Dijkstra w/heap**	**Kruskal**
Dense (20% adjacent)	(228, 3471)	1.376872	1.468463	0.895211
	(1337,3832)	1.400492	1.485469	0.892773
	(2049,1478)	1.209513	1.452758	0.717506
	(2518,4705)	1.315818	1.416391	0.902763
	(4305, 392)	1.350259	1.402450	0.902173
Sparse (E = 4N)	(228, 3471)	0.065000	0.019788	0.018207
	(1337,3832)	0.068081	0.018863	0.023489
	(2049,1478)	0.037199	0.015309	0.018287
	(2518,4705)	0.064428	0.016969	0.018089
	(4305, 392)	0.075702	0.030627	0.018995

Table3. runtime variance regarding different E with fixed N = 5000, path (s,t) = (2518,4705)

Run Time (s)	**# of edges**	**Dijkstra**	**Dijkstra w/heap**	**Kruskal**
Fixed Vertices # : 5000	5,013,009	2.816910	2.671694	1.634043
	4,562,291	2.552364	2.474628	1.542909
	4,094,749	2.321401	2.400907	1.343214
	3,467,291	1.766239	1.851315	1.163282
	2,957,775	1.861061	1.860849	0.980218
	2,466,957	1.360624	1.467458	0.827262
	1,966,944	1.163997	1.254522	0.707236
	1,479,830	0.894410	0.853046	0.511484
	1,000,376	0.602369	0.665683	0.360722
	499,246	0.368418	0.403434	0.179541

Table.4 runtime variance regarding to N for fixed E = 1million, path(s,t) = (0,N/2)

# of edges	**# of vertice**	**Dijkstra**	**Dijkstra w/heap**	**Kruskal**
994,597	10,000	0.934183	1.213662	0.397255
1,017,494	8,000	0.731319	1.015238	0.388177
997,142	6,000	0.677397	0.861610	0.380672
993,301	4,000	0.525177	0.568452	0.374627
1,006,460	2,000	0.507202	0.509313	0.344728

Table.5 runtime variance regarding to N for both dense and sparse graph, path(s,t) = (0,N/2)

# of edges	**# of vertice**	**Dijkstra**	**Dijkstra w/heap**	**Kruskal**
9,949,323	10,000	6.887486	6.571589	3.428256
6,366,100	8,000	3.519929	3.989361	2.203421
3,582,564	6,000	2.033149	2.033811	1.245757
1,592,301	4,000	0.929612	0.996396	0.564580
398,803	2,000	0.235572	0.242478	0.151873
39,768	10,000	0.212762	0.096110	0.049913
32,530	8,000	0.140349	0.042034	0.031565
23,937	6,000	0.025346	0.013636	0.024468
16,000	4,000	0.020246	0.017256	0.014572
7939	2,000	0.017437	0.008268	0.006974

Discussion

As discussed in class, the time complexity of these three approaches are:

- Dijkstra: O(mn) when the fringe list is stored in the linked list
- Dijkstra w/heap: O(mlogn) when the fringe is stored in the heap data structure which could be access in logn time
- Kruskal: O(mlogn) time algorithm since DFS also takes linear time

From Table.1 and Table.2 we can see that the difference of source-target pair doesn't affect runtime as much as the graphs' own properties. From Table.1 we can see that if the graph is intrinsically more complex it would require more runtime no matter what source-target pair you choose to find the max-bandwidth path.

Now we can take a look at the test results to see if they correlated with the theoretical runtime we listed above. Table.3, which we fixed the total number of vertices N = 5000 and fixed source-target path = (2518,4705), shows clearly that the runtime of all approaches grows linearly and proportionally to the total number of edges, which fits the theoretical runtime that all three approach have runtime linear related to E (O(m*x), x is other variables). Table.5 simply shows that Kruskal have better performance no matter in what circumstances.

For Table.4, which we fix the number of edges approximately equal to one million and source-target path = (0,N/2) for different N, theoretically it should show that the Dijkstra should still be proportional to N while Dijkstra w/heap and Kruskal should be proportional to logN. However, the Kruskal and Dijkstra without heap performs as expected, the Dijkstra w/heap doesn't have overall better performance compare to the one without.

This could be explained by my implementation variance from the theoretical one. One of the most expensive operation for heap structure is search since there is no storing preferences for children nodes. The suggested building of the heap structure is to have two array: H[] one represents the heap indexes and structure shape which stores the vertex number, the other D[] stores the data of vertices which is index exactly by vertex numbers. By this implementation, although we still could not directly access the index number in the heap structure by given a vertex number in constant time, we could modified the data of the given vertex name directly without finding the storing position in the heap structure, i.e. D[H[]].

This is extremely useful and efficient for the implementation of the Dijkstra w/heap. The time consuming dominate steps in Dijkstra's algorithm are the choosing of maximal labels among fringes and updating fringes' labels. For the Dijkstra without heap, although the updating could be done in constant time by directly access the fringe array by indexing the vertex number as I mentioned in the implementation part, the max-fringe-picking step could not avoid to scan through all fringes to find one which takes O(n) time. This is why we uses heap to improve Dijkstra, since most heap operations could be done in O(logn) time. The ones Dijkstra need for max extract can even be done in constant time, but the problem comes down to the updating labels which requires a search operation. Luckily the implementation trick of that two arrays H[] and D[] avoid the updating of data to require the heap position. Since my implementation could not properly make use of it, the performance of my Dijkstra with heap simply doesn't perform better than without heap—while sometimes even worse.

Here is the reason I thought to be possible that the Dijkstra with un-perfect heap implementation has worse performance from time to time. This need to come down to the discussion of the relations between N and E (total number of vertices and edges in the graph). We can see from Table.1 and Table.2 that the heap-Dijkstra have better performances in sparse graph while worse in dense graph compared to non-heap Dijkstra. As we mentioned, the difference runtime details in this two approach is that Dijkstra needs O(n) time to pick the max label fringe to visit next but almost constant time to update the labels, while Dijkstra w/un-perfect heap only needs constant time to extract the max fringe but O(n) time to update the labels. It seems like they are now equal in regards of O(n) runtime for this two procedure. However, the maximal times Dijkstra without heap needs to extract max is N, while Dijkstra w/un-perfect heap needs at most near E times to update.

This is why the Dijkstra w/ heap performs better in sparse graphs since there are smaller Es and performs worse in dense graph with larger E. But from Table.3 we see that Dijkstra w/un-perfect heap performs better again in some even denser graphs. My explanation mat be that the difference isn't too significant and it is not an average runtime since I only ran once. Another possible reason is that for denser graphs, the fringe list would be filled almost close to N very fast, which make Dijkstra without heap takes almost exact N time for each extract, while the updating of labels isn't as frequent as expected.

The discussion about Kruskal's algorithm approach to find max-bandwidth path is extremely detailed, I am not going to mention here again. The first time I implemented the Kruskal approach have significant longer runtime which confuses me. The extract max of all the sorted edges takes a great amount of time therefore make the creation of Maximum Spanning Tree extremely long. However, because Maximum Spanning Tree is a "tree" without a doubt, thus the maximum number of edges is bounded by N-1. So there is no need to extract all edges from the sort heap list, the procedure would only need run until the edges added to the tree equals to N-1. This modification boosted my runtime by almost four times faster.

Improvement

The most obvious improvement to implement is to implement the correct heap structure mentioned in previous section. This could undoubtedly speed up the Dijkstra with heap from O(nm) to O(mlogn).

Also, the sorting of all edges in Kruskal takes great amount of time as mentioned. I thought an interest point is that this sorting step takes O(m) time, which means that it takes longer whenever the graph is denser, however we only need N-1 edges at last for the Maximum Spanning Tree. We might not need to sort or insert all edges into the heap because the smaller edges are never going to extract to add in the tree anyway, since it is a dense graph, it is unlikely that the deleted edges would make the final graph unconnected. This required more study to examine the correctness of this idea, and it is somewhat similar to the idea that discussed in class: the one includes Median-Finding and Connected-Component.