CSCE 629: Analysis of Algorithm Project Report

Samyuktha Sankaran(UIN: 327006025)

Network routing protocol using data structures and algorithms

Abstract

Implementing the Dijkstra's algorithm to find the Maximum bandwidth path, with and without heaps and to find the Maximum Spanning Tree using Kruskal's algorithm.

1 Introduction

Network optimization refers to a set of technologies and techniques that are geared towards improving the network performance. The project has the following components.

- Generating a sparse graph G1, with an average vertex degree of 6 and a dense graph G2, with each vertex connected to 20% to the other vertices. The graphs are assigned random weights.
- Heap structure - Max heap. Implementing the MAXIMUM, INSERT and DELETE routines. This will later be used with the Dijkstra algorithm implementation using heaps.
- Implementing routing algorithms to find the **Maximum bandwidth path**. There are three different implementations: Modified version of Dijkstra's algorithm to find the maximum bandwidth path with and without heaps and finding the **Maximum Spanning tree** which gives the Maximum bandwidth path.
- Testing with multiple samples to verify the performance of these algorithms and summarizing the results to draw an inference.

2 Implementation

I have implemented two main Graph algorithms, with modifications, to retrieve the maximum bandwidth paths. The pseudo codes of these implementations are as follows.

2.1 Maximum-heaps

The maximum heap is used in Dijkstra's algorithm to extract the maximum bandwidth. The heap is implemented as a pair of integers, the <v, BW[v]> where v is the vertex and BW[v] is its corresponding bandwidth. The Insert, delete and extract-max routines are implemented.

- The MAX-HEAP property is the value of the node is greater than its left and right child. This property is to be maintained throughout the implementation. Each time an insertion or a deletion takes place, suitable subroutines are called to ensure this balance.

- INSERT is to push values to the heap. The heap is stored as an array. The heap is initialized to the number of vertices in the graph. The new entry is added at the end of the array and heapify up is called to place the value at the correct position.
- DELETE is to delete a value from the heap. Since the elements are stored as a pair, the key is unique. The key(that is, the vertex) is passed to the function, which is located in the array. It is swapped with the root of the heap and a pop() operation is carried out.
- EXTRACT MAX, which gives the maximum in the heap, i.e., the root of the heap, the first element in the array. Each time an extract max is carried out, the top element is returned to the function, the root is replaced with the last element of the heap and size is reduced by 1. heapifyDown routine is called to restore the heap property.
- heapifyDown - here, for a given index, its parent left and right child are taken. It is compared with the left and right child, and swapped if either of them are larger than the value at the current index. The heapifyDown routine is called recursively till the value is placed in the correct position.
- heapifyUp - here, the value at the given index is compared to its parent's value. If it is larger than its parent's , then it is swapped and called recursively till it reaches it correct position.

2.2 Dijkstra's algorithm(without heaps)

Dijkstra's algorithm is used to find the single shortest path between the source and destination. Here, I implement a modified version of the same, only to find the Maximum Bandwidth path. There are three arrays : BW[], the bandwidth of each vertex from the source; parent[], it updates the parent of each vertex to track the path back to the source, Here I implement without the heaps, so after initializing part, I find the vertex with the maximum bandwidth value, which is the source.

Time complexity: For modified Dijkstra's algorithm, without using the heap structure, finding the vertex with maximum BW takes linear time. Therefore total time complexity is $O(V^2 + E)$, where V is the number of vertices and E is the number of edges.

2.3 Dijkstra's algorithm(With heaps)

This is the same modified version of Dijkstra like in the previous section, with usage of **maximum-heaps** for retrieving the vertex with maximum bandwidth value. In max-heap structure, it takes constant time to retrieve this value, hence it is an improvement in the performance.

Time Complexity: It takes constant time to extract maximum and $O(ln(V))$ times to get the minimum value, insertion and delete operation, whereas previous case, it took constant time for insertion. The total time complexity is $O((V + E) * (ln(V)))$.

2.4 Kruskal's algorithm

Kruskal's algorithm is used to find the minimum spanning tree - which is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. Here, Iuse modify the Kruskal algorithm to find the maximum spanning tree, from which I can compute the maximum bandwidth path between two points.

Lemma: The s-t path obtained in Tmax is the maximum bandwidth path for s-t pair in the graph.

Time Complexity: In previous Dijkstra's algorithm, I compute the path between the source and the destination. Here, I find the entire maximum spanning tree and extract the source-destination path. It contains the MBP for any pair of vertices. The sorting is carried out using heapsort and Kruskal inserts all the E edges into the heap. It takes $O(E * (ln(V)))$ time and the unionSet operation takes constant time, but the findSet operation can take $O(ln(V))$ time. And from the MaxST, to obtain the s-t path using DFS takes $O(V + E)$ time.

2.5 Implementation details

I have implemented all the programs using C++. I have used vectors in STL, to store the heap structure and maintain the bandwidth, parents and the status arrays. This is an efficient way to store the dynamic arrays, on the contrary of using static array would prove to be very expensive given the size of graph is not known beforehand. As for the status array - 0 represents UNSEEN, 1 represents a FRINGE and 2 represents an INTREE. The graph is stored in the Adjacency list format as vector of vector of pairs. Each vector corresponds to one vertex and the each pair in the vector are the adjacent vertices with the corresponding edge weights. For maximum spanning tree, there is a slight alteration, where I have used vector of pair of pairs, each element in the vector is of form (weight, (u-v)). This makes sorting using heapsort more convinient and accessing the necessary values are more easier.

3 Testing

The above algorithms are successfully implemented and the time taken to compile and execute are presented here. Three test samples for sparse and dense graphs with nodes 10, 100 and 2000 are presented here.

3.1 Implementation

The subroutines implemented here, generates two types of graphs - G1: sparse graph and G2: Dense graph. Both the graphs are undirected, weighted graphs. To ensure connectivity, all the vertices are connected to their adjacent numbered vertices, i.e., i is connected to i-1 and i+1.
G1: It is a randomly generated graph, with an average vertex degree of six. A count array is maintained, which holds the degree of the vertex.
G2: Here, each vertex is connected to 20% of the adjacent neighbours. It is similar to flipping a biased coin, with a chance of getting 20%head.
The neighbours and weights are assigned randomly using the rand() and srand() functions, available in c++. For each set of sparse and dense graph, 5 pairs of (source, destination) vertices are chosen and tested.

3.2 Observations

The following are the time taken in seconds to execute the sub-routines. The results are an average value for 5 queries per graph.

Input Graph(G)	Dijkstra with heap	Dijkstra without heap	Kruskal
G1 (N = 10)	0.003649	0.006	0.004414
G2 (N = 10)	0.003990	0.004389	0.00321
G1 (N = 50)	0.006474	0.006978	0.007552
G2 (N = 50)	0.006588	0.007197	0.00525
G1 (N = 100)	0.007026	0.009493	0.006
G2 (N = 100)	0.008025	0.00874	0.007626
G1 (N = 1000)	0.031731	0.032885	0.028142
G2 (N = 1000)	0.030347	0.030405	0.059971
G1 (N = 5000)	0.039692	0.079199	0.05497
G1 (N = 5000)	0.364192	0.417262	0.910522

- Looking at the time values, I can infer that the Dijkstra algorithm without using heaps is the worst amongst the three. But the interesting part is, the usage of these algorithms depends on the type of application. Here the analysis has been carried out on two types of graphs - dense and sparse.

- Kruskal's algorithm performed worse for dense graphs, especially for the case with higher number of nodes. This is because, it computes the entire tree which is redundant - more so, if the (s-t) vertices are in close proximity to each other. Most of the time taken here, is in computing the maximum spanning tree. Finding the path takes comparatively lesser time. This is quite contrary to the theoretical statement, that Kruskal performs 4 times better than the Dijkstra.

- While looking at graphs with comparatively lesser number of nodes, the Kruskal approach is comparable to the Dijkstra's approach. In case where there is 100 nodes per graph, the Kruskal's performs better than Dijkstra.

- For sparse graphs, heap approach produces faster results. Dijkstra takes considerable time due to construction of the heap, and each time a value is inserted or maximum is popped out, the heapify routine has to be called to restore the heap property.

- An interesting observation was, although Kruskal takes more time in computing the Maximum spanning tree, once the tree is computed retrieving the s-t paths are considerably faster than that of Dijkstra. In case of Dijkstra, for each s-t pair, the entire routine has to be carried out and then the path is computed. Whereas in the former, it computes the tree and the path for any pair can be easily obtained.

3.3 Inference

The following inference can be drawn from the observations made

- There can be more than one Maximum Bandwidth path for a given pair. Three approaches are capable of providing the same or three different paths. However, the bandwidth of the path must be the same in all these approaches.

- The Dijkstra's and the Kruskal's algorithms are greedy approaches and not necessarily gives the optimal solution.

- Although, both Dijkstra and Kruskal have same order of complexity (say if Kruskal takes 2Elog(V) times, Dijkstra takes 10Elog(V) times), the application determines which algorithm is to be used.
- For sparse graph, there is not much change in the Kruskal algorithm, except for the number of s-t pairs that are being computed. The heap based Dijkstra has been performing consistently for all the cases.
- If there is a dense graph and not too many s-t pairs, it is absolutely redundant to use the Kruskal algorithm. The too much time consumed in finding the MaxST can't be efficiently compensated by computing the s-t paths in lesser time.
- Also, if there is a scenario where the graph is constantly changing, it is best to avoid Kruskal's algorithm.
- The first approach performs similar to the second one, only where it is a very small graph. In other cases, it is more time consuming than the latter. The naive approach, does not provide fruitful results in any the cases. It can probably used where there are frequently changing graphs with too few queries to compute.
- The Dijkstra with heap approach does not depend on the graph size or number of queries. On an average, it performs better than the other two methods.

There are however some challenges in implementing these algorithms in real time. Insertion and updating heaps is a costly operation in case of highly connected large graphs. While working with these algorithms, most of the errors were logical errors, like overflow issues, etc. While generating random graphs, at most care has to be taken to avoid cycles and forming multiple edges between the same pair of vertices. Weights need to be allocated at random. There is a tendency of repetition of weights, so the range of the random number generation needs to be chosen carefully. Working with smaller graphs are considerably easy, debugging gets a lot harder with bigger and heavier graphs.

3.4 Further Improvement

The kruskal algorithm can be further improved using the A* approach and the MakeSet-Find-Union approach. A* is an extension of the Dijkstra's algorithm. It is more efficient, however it gives a good route, but not the best route. It takes up a grid structure and computes the path. Another one is using M-F-U. Here it explores the concept of boundary and interior nodes and is almost in linear time. However it is quite hard to implement the same.

4 Conclusion

Based on this project, I would suggest to go with **Dijkstra's algorithm using heaps**, especially when there is no previous knowledge of the graph or number of queries. Some of the factors to consider while picking an algorithm to work with would include, the ease of implementation, the type of graph,whether it is dense or sparse, the application (a crucial factor to be considered), the number of queries that are to be computed, system limitations,etc. In most of the cases, it is better to have a good trade off between the performance of the algorithm and the time it takes to compute the results.