CSCE 629-601 Analysis of Algorithms - Project
Routing Algorithms

Gundam Satyabhama Reddy

Abstract

In this project, different routing algorithms to solve the Max-Bandwidth-Path problem have been implemented. These include Dijkstra's algorithm with and without using a heap structure and Kruskal's algorithm using heap sort. An improved Kruskal's algorithm is also proposed and implemented. The algorithm running times are compared for randomly generated sparse and dense graphs.

1. Random Graph Generation

Two types of random graphs are generated to test the algorithms:

Version 1 (G1) : The average vertex degree of the graph is 6. To generate such a graph, we first connect all the vertices into a cycle to make sure the generated graph is connected. The average degree of the graph now would be 2 (as there would be 2 edges on each vertex due to the cycle added). Adding an edge to the graph increases the degree count by 2 as each edge contributes to degree increment of both the vertices it is connected to. Since, the average degree is presently 2, we just need to randomly add **2*num_vertices** more edges without repetition to reach an average degree of 6.

Version 2 (G2) : In this version, each vertex is adjacent to about 20% of the other vertices. We first construct a cycle as done previously to make a connected graph. We then maintain a neighbour_count array indicating the number of edges for each vertex. The values are currently initialized to 2 due to the cycle. A random vertex is chosen based on neighbour_count array and random edges are added to it till it reaches the above condition of 20% neighbours. The neighbour_count array values are incremented for the source and destinations. During this process, we also make sure to check that we don't repeat any edges. Once the neighbour_count of a vertex reaches 1000 (20% of 5000) we pop it from the array. This way we reduce the number of repetitions during random number generation, as the length of the array keeps reducing. Eventually, we would reach a state where we cannot add any more edges; either because there already exists edges between the remaining vertices that we are trying to add an edge to, or because we are left with a single vertex and we cannot add an edge. In either case we stop the algorithm and return the generated graph. On observation, around 15 vertices would end up having close to 1000 edges while the rest have a degree of 1000. This is a brute force approach for the generation and has scope for improvement. The current running time is around 2 minutes for 5000 vertices.

The weights for all the edges in both the versions range from 1 to 100 and are assigned randomly.

The resulting graph is stored as an adjacency list of size num_vertices. The indexes indicate the source and the values indicate the class **Neighbour** which has the destination and the edge weight to that destination. Since we are dealing with undirected graphs, each add edge also adds another entry from destination to source.

2. Max Heap

A Max heap structure has been implemented for use in Dijkstra's algorithm. The three arrays H, P and D are described in the project description. H holding the vertices names, P holding the positions of the vertices in the heap and D, the values of the vertices in H. The three operations insert, delete and maximum are implemented as described below:

insert(vertex, value)
Insert function takes in the vertex and its corresponding value. The D array is updated with the value and the vertex is inserted at the end of the heap. This is then moved up to its correct position by swapping until it satisfies the max heap property. The size of the heap is incremented by 1. The positions in P array are also adjusted accordingly during swaps. Since the maximum height of the tree cannot exceed log(n), the time complexity of insert operation is **O(log(n)).**

delete(vertex)
Delete function takes in the vertex to deleted from the heap. The position of the vertex is determined using the P array which is a constant time operation. Optionally, the value D array can be set to -1. During delete, we get the last element from the heap, place it in the position where the vertex is supposed to be deleted, and decrement the size of the heap by 1. This node is then pushed up or sifted down to satisfy the max heap property that the parent should always be greater than its children. The positions in P array are also adjusted accordingly during swaps. Similar to insert, the time complexity of delete operation is **O(log(n)).**

maximum()
This is a constant time operation and returns the root of the heap (H[0]) if not empty. Hence, the time complexity being **O(1).**

3. Routing Algorithms

We implement 3 algorithms to solve the Max-Bandwidth-Path problem:

a. Dijkstra's algorithm without using a heap structure

As a first step we mark all the status of all vertices as unvisited, and b_width values as 0 and dad as 0. We then set the source's status as in-tree, update b_width to infinity and dad to -1. All of it's neighbours' are marked as fringers and the b_width and dad arrays are updated accordingly. We then iteratively get the maximum fringer using the status and b_width array. For each of these fringers v, we mark it as intree and get their neighbours w's, if w is unvisited, we mark it as a fringer and update the bandwidth to be **min(b_width[v], w.edge_weight)** (1)**.** And also update the dad. If w is already a fringer, we try to correct it's b_width value if it is less than (1). The dad is updated in this case as well. This algorithm continues until there are no more fringers and all vertices become intree. We then construct the BW path using the dad array and b_width[destination] would give us the BW value.

As we can see, the initializations are per vertex and can take **O(n)**. Also, every vertex can be a fringer at most once and when it gets picked, it is made intree and its neighbours are checked. So, this step(neighbours step) runs at most **2*m** times overall as the graph is undirected and the total number of edges is m. Getting the maximum fringer itself takes **O(n)** as we need to iterate over status and b_width arrays. Hence the running time to iterate all the fringers is **O(n * n) + O(m).** Since m <= n^2, the total time complexity of the algorithm is **O(n^2).** Clearly, the calculation of maximum fringer is a bottleneck here which gives the motivation to use heap structure for fringes.

b. Dijkstra's algorithm using a heap structure for fringes

The only difference from the usual approach above is how we calculate the maximum fringer and its storage. We use the Max Heap structure built in **(2)**. The values inserted in the heap are the fringer vertices and its b_width values. This way getting the maximum fringer can be done in constant time. We add the vertex into the heap whenever we set the status as fringer and it is deleted from heap when we get the max fringer. We also delete and insert back the vertex into heap when we are updating the bandwidth value of an already existing fringer. All the inserts and deletes take **O(log(n))** time. So the step that previously took **O(2*m)** now would take

O(m log(n)) overall due to the addition of insert and deletes while iterating neighbours. Iterating over the fringers would take **O(n * log(n))** (because of delete after max) **+ O(m log(n)).** And since the graph is connected, m is atleast n-1. Therefore the time complexity of this algorithm can be considered as **O(mlog(n)).**

c. **Kruskal's algorithm with HeapSort**
   In Kruskal's algorithm, we first sort all the edges according to their edge weights. I used a **Min Heap** structure to sort the edges in non increasing order using heap sort. The elements are in the form **Edge(u,v, weight).** While constructing this structure, i make sure that the duplicates in the adjacency list due to undirected graph are not included repeated in the list by setting the condition **u > v**.The heap sort is performed inplace based on the weight values. We first start from the last non leaf node and call heapify iteratively until we reach the root. This creates the min heap structure required to sort. We then iteratively, swap root node with the last element and decrement the size. We then call heapify on the root node to send the node to its right position. At the end, the root node would have the maximum most value and the last node in the array has the minimum. The total time taken for this is **O(m log(m)).**

   Once this is done, we proceed with the **Kruskal's** algorithm to generate the Maximum Spanning Tree. The Spanning tree T is first initialized to a graph with the same number of vertices as G and no edges. For each edge from the sorted list above, if u and v belong to different pieces in T, we add the edge to T and connect the two pieces. We stop this algorithm when we exhaust the edges or when T has n-1 edges.

   This logic is achieved using makeset union and find operations using two arrays dad and rank. Find is implemented using path compression. In union we try to merge 2 trees and we always attach the shorter tree to the larger tree to avoid increasing the rank. If the two trees are of equal height, the rank is increased by 1 for one of the roots and the tree is attached to it. Makeset and union operations take constant time. Each find operation takes **log(n) time** though this decreases in subsequent finds due to path compression as discussed in class.

   Coming back to Kruskal's algorithm, we use 2 finds for each edge and determine if they are in the same piece based on their roots. If they are not, we merge them using union into a single tree, and also add that edge to the spanning tree. So find is called at most **2*m** times and union is called at most **n-1** times and makeset (used for initializations) is called **n** times. Therefore the running time of the algorithm is given by ~ **O(m log(m))** (sorting) **+ O(m log (n))** (m times find) **+ O(n)** (makeset and union). As m can be at most n^2, log(m) can be at most 2*log(n). Therefore, the time complexity can be approximated as **O(mlog(n)).**

Once the MST is constructed, we find a path in it from source to destination using DFS which takes **O(n+m),** but m here is n-1 as it an MST so it takes **O(n)** time. The bandwidth value is then calculated from the constructed path.

d. **Kruskal's improved algorithm with BuildHeap and popMax**
Due to python's slow execution times(described below), a few modifications were made to Kruskal's algorithm. For the dense graphs there could be as many as 2.5 million edges and we are trying to sort these in the beginning of Kruskal's algorithm. Instead of sorting all the edges at once, a Max Heap was used instead of Min Heap. And we call **buildHeap** (constructs the heap, without sorting). We then use **popMax** to get the maximum element and also delete it from the heap, which internally calls heapify. So instead of running heapify for all the 2.5 million edges during sorting, we only call heapify until we reach **n-1** (i.e, 4999) edges in the spanning tree. This is a significant improvement. The time complexity for this algorithm is still the same **O(mlog(n)),** but the execution times differ drastically for dense graphs.

4. Implementation Details

The algorithms are implemented in python and a README.md is provided along with the source code explaining the file structure.

5. Testing and Observations

5 pairs of graphs G1 and G2 with 5000 vertices are constructed as described in **(1).** For each graph we choose 5 random source and destination pairs to calculate the maximum bandwidth path. Therefore, there are a total of 25 graphs each for G1 and G2 types. The overall average execution time of the graphs is given below:

Overall average times for **G1** type graphs - sparse graphs :

Dijkstra **V1**	Dijkstra **V2**	**Kruskal**	**Kruskal Improved**
0.89471secs	**0.09992secs**	0.10043secs	0.10108secs

Overall average times for G2 type graphs - dense graphs :

Dijkstra **V1**	Dijkstra **V2**	**Kruskal**	**Kruskal Improved**
2.08393secs	**1.32767secs**	25.86055secs	3.64459secs

Dijkstra V1 being without heap and Dijkstra V2 being with heap.

Number of edges in the sparse graph would be 15,000 and in dense graph it would be around 2.5 million.

Looking at the execution times, we can observe that Dijkstra V2 is better than Dijkstra V1. Which can also be seen in the time complexities O(n^2) and O(m log(n)). When the number of edges are increased, the execution time of V1 increased by ~2 fold where as the execution time of V2 increased by ~13 fold. This can be explained by the fact that as m moves closer to n^2, O(m log(n)) will tend to be worse than O(n^2). While for smaller values of m, O(m log(n)) is better than O(n^2). Hence different versions of Dijkstra's algorithm can be chosen based on the values of m and n.

Coming to Kruskal's algorithm, it performs better than Dijkstra **V1** for sparse graph, but for dense graph it takes almost 26 seconds. Looking into the execution time of the algorithm, I found that the major bottleneck is heap sort:

Heap sort time : 24.966 seconds
MST construction time : 0.017 seconds
Path construction time : 0.002 seconds

The same code written in C++ drastically changes the performance of heap sort (< 2 seconds). As I am using Python, the heap sort seems to be taking a lot of time. Moreover, the increase is also due to the 2.5 million edges in the dense graph O(mlog(m)).

The improved version of Kruskal's algorithm was thus implemented. For sparse graphs, there is no significant change in execution times, but for dense graphs, the execution time reduced ~7 fold when compared to normal Kruskal's algorithm. Looking into the execution time again, we can see that build heap doesn't take as much time as sorting.

Build Heap time : 1.211 seconds
MST Construction time with popMax : 0.326 seconds
Path construction time : 0.0018 seconds

Kruskal's algorithm works better with sparse graphs than with dense graphs which is evident from the algorithm. And, out of all the algorithms Dijkstra's algorithm with Heap seems to be the best**.**

6. Future Improvements

a. The random graph generation algorithm for G2 can be improved further. Currently, a brute force approach is implemented.
b. We can be intelligent in adding the edges in the adjacency list, like inserting them in a sorted order, etc. That way, the sorting in Kruskal’s can be done in linear time.
c. In Dijkstra’s algorithm, instead of deleting and inserting the vertex back to the heap if its already a fringer; we could use “update” to update the band width value and call heapify. This way, we would not need to call heapify twice(once in delete and once in insert).
d. Other sorting algorithms can be explored for sorting edges in Kruskal’s algorithm.
e. A priority queue instead of Max Heap could also be experimented with in Dijkstra’s algorithm.

7. Conclusion

2 versions of Dijkstra’s and Kruskal’s algorithms each were implemented and their analysis has been provided. Testing was done on both sparse and dense graphs for all the algorithms and average execution times were provided. Out of the 4 algorithms, Dijkstra’s algorithm with Heap structure seems to perform the best.