Performance Analysis of Max-Bandwidth-Path Algorithms

CSCE-629 : Analysis of Algorithms

By :

NFN Nimisha

UIN : 932002518

Texas A&M University, College Station

November 30th, 2021

Introduction

The objective of this project is to implement different algorithms used in the MAX-BANDWIDTH-PATH problem and analyse their running time. The algorithms are coded in **Java** programming language. The algorithms shown in this project are:-

1. An algorithm for Max-Bandwidth-Path based on a modification of Dijkstra's algorithm without using a heap structure.
2. An algorithm for Max-Bandwidth-Path based on a modification of Dijkstra's algorithm using a heap structure for fringes.
3. An algorithm for Max-Bandwidth-Path based on a modification of Kruskal's algorithm, in which the edges are sorted by HeapSort.

Project File Structure

Edge.java - This file contains a user-defined class to represent weighted edges of the generated graph. src is source vertex and dest is destination vertex of the edge.

Graph.java - This file contains a user-defined class to store the generated graph. It contains private variables to store the number of vertices in the graph and adjacency list for the graph.

GraphGenerator.java - This file contains code to generate random sparse and dense types of graphs for a given number of vertices and graph density..

DijkstraAlgo.java - This file contains code implementation for Dijkstra algorithm for finding Max Bandwidth Path without using a heap structure.

DijkstraAlgoWithHeap.java - This file contains code implementation for Dijkstra algorithm for finding Max Bandwidth Path using a heap structure.

MaxHeap.java - This file contains implementation of Max Heap data structure used in DijkstraAlgoWithHeap algorithm. It contains H[], D[], P[] arrays that store the names, values and position of elements in the Max Heap.

KruskalAlgo.java - This file contains code implementation for Kruskal algorithm for finding Max Bandwidth Path in which edges are sorted by HeapSort.

MakeUnionFind.java - This file contains code for the Union-Find algorithm used in Kruskal algorithm for performance optimization.

Test.java - This file contains test code that generates 5 Sparse and 5 Dense graphs. For each graph, 5 source-destination vertex pairs are generated and passed to the three algorithms. The running time of the three algorithms for each type of graph is recorded and printed.

Implementation Details

1. Random Graph Generation

For random graph generation, a method is created that takes as input the type of graph to be generated and the number of vertices in the graph. As per the requirements described in the project, the average vertex degree for sparse graphs is taken as 6 and for generating dense graphs, average vertex degree is 1000 (each vertex is adjacent to about 20% of the other vertices, i.e. 20% of 5000 vertices = 1000 vertices) and no of vertices is kept as 5000.

To ensure the graph is connected, a cycle is created containing all the vertices of the graph in the beginning. Thereafter, for each vertex till the average vertex degree is not reached, we keep on adding edges between random vertices if an edge doesn't exist already and update the adjacency list of the graph which is an array of ArrayList of Edge class. The edges are assigned weights from 1 to 1000 at random using a Random class in java.

2. Heap Structure

We have implemented a Max-Heap structure that is used in the second Dijkstra algorithm to find Max-Bandwidth-Path in a network. It supports extractMax(), insert() and delete() operations. We have made use of three arrays, **int[] H** - array which stores the names or ids of the vertices - 0, 1, etc, **int[] D -** array which stores HeapNode objects. HeapNode has 2 variables, the name of the node and its weight. We have used a class here because multiple vertices can have the same weight, so we have kept the weight together with the node name in an array. **int[] P -** array to store the position of each node in the max-heap. The array is updated for each of the extractMax(), insert() and delete() operations. Keeping a position array for the heap elements allows faster deletion from the max-heap instead of searching for the node first and then deleting it.

3. Routing Algorithms

● Dijkstra Algorithm

We have implemented the Dijkstra algorithm as discussed in the class. The method takes as input the graph, source and destination vertex between which max-bandwidth-path needs to be calculated and returns the max bandwidth. We have kept three arrays **status[]** - which stores status of the nodes as UNSEEN, INTREE or FRINGE, **dad[]** - which stores the predecessor of each node and **bw[]** which stores the max bandwidth achievable till that node. We initialise the source vertex as INTREE and update all its neighboring nodes as FRINGE in the status array. We get the max fringe among the fringes and iterate through its neighbors. If the neighbor is UNSEEN, we update it to fringe and also update its bandwidth in bw[] array. Else if the neighbor is a FRINGE, we check if its bandwidth is less than the current bandwidth and update its weight. This process is repeated till the status of destination vertex becomes INTREE. In the end, we return the bw of the destination node.

Time complexity analysis:

As discussed in class, the TC of the above algorithm is O(V^2). To get the maximum fringe, we iterate over the status[] of max size V and this operation is done for at max V times. Therefore the TC is O(V^2).

- ## Dijkstra Algorithm with Heap

The above algorithm is the same as the Dijkstra algorithm mentioned above. The only difference is with regards to how we are storing the fringes here. We make use of the Max-heap data structure implemented above to store the fringes. The extractMax() operation to extract the max fringe takes time O(logV) because after extracting the max fringe, we fix the heap so that it remains a max-heap which runs in time O(logV) equal to the height of the heap. The extractMax() operation can run for maximum V times. We insert the unseen neighbors of the max fringe to the heap. This operation also takes time O(logV). If the neighbor is already a fringe and its bandwidth is less than the current bandwidth, we delete the node from the heap using **P[]** array and insert the node with updated weight. These insert() and delete() operations are performed for a maximum of #edges and therefore takes time O(E logV).

Time complexity analysis:

The worst case TC of the algorithm as discussed above is O((V+E) log V). If all the vertices are reachable from the source, then E <= $|V^2|$ and the TC can be expressed as O(E logV).

- **Kruskal Algorithm**

  In the Kruskal algorithm, we store all the edges of the graph in an array and sort it in non-increasing order of edge weights using heapsort(). Then we pick the edge with the largest weight and add it to another graph. We make use of Union-Find data structure to check if the edges do not form a cycle and if they don't we add it to a new graph which is a Maximum Spanning Tree. Then we do a BFS on the maximum spanning tree to find the maximum bandwidth path between source and vertex.

  Time complexity analysis:

  The TC of sorting the edges using heapsort takes time O(E log E). The Union-find data structure implemented uses **Union by rank** and **Path compression**. The TC of Find operation is O(log V) and Union operation is O(1). This is repeated no of edges E times. Therefore, the total TC of the algorithm is O(E logE + E log V). If all the vertices are reachable from the source, then E <= $|V^2|$ and the TC can be expressed as O(E logV).

Performance Analysis

The results of the testing for each of the above algorithms on sparse and dense graphs are shown below with average running time in seconds:-

Sparse Graph Performance Analysis

		Source	Destination	Running time of Dijkstra algo (in sec)	Running time of Dijkstra algo with heap (in sec)	Running time of Kruskal algo (in sec)
Sparse Graph 1	Vertex Pair 1	4731	1268	0.038	0.03	0.061
	Vertex Pair 2	2669	371	0.006	0.006	0.032
	Vertex Pair 3	3874	4915	0.078	0.012	0.028
	Vertex Pair 4	1938	3976	0.005	0.001	0.033
	Vertex Pair 5	3758	1621	0.055	0.011	0.022
Sparse Graph 2	Vertex Pair 1	4350	1861	0.004	0.001	0.008
	Vertex Pair 2	2915	483	0.006	0.001	0.006
	Vertex Pair 3	3308	3994	0.023	0.002	0.006
	Vertex Pair 4	4140	992	0.001	0.001	0.008
	Vertex Pair 5	2664	3835	0.01	0.002	0.005
Sparse Graph 3	Vertex Pair 1	4086	432	0.008	0.001	0.006
	Vertex Pair 2	2850	2015	0.009	0.001	0.006
	Vertex Pair 3	782	2946	0.012	0.001	0.005
	Vertex Pair 4	3175	633	0.009	0.001	0.005
	Vertex Pair 5	4438	4561	0.019	0.002	0.004
Sparse Graph 4	Vertex Pair 1	1310	3476	0.015	0.002	0.009
	Vertex Pair 2	3054	724	0.023	0.003	0.007
	Vertex Pair 3	3953	880	0.021	0.002	0.006
	Vertex Pair 4	616	1521	0.009	0.001	0.006

	Vertex Pair 5	3056	2646	0.007	0.001	0.006
Sparse Graph 5	Vertex Pair 1	4944	4599	0.02	0.002	0.004
	Vertex Pair 2	2196	1936	0.001	0.001	0.004
	Vertex Pair 3	4262	1983	0.017	0.002	0.004
	Vertex Pair 4	2437	4470	0.018	0.001	0.005
	Vertex Pair 5	4891	3902	0.005	0.001	0.005
Avg Running Time (in sec)				**0.01676**	**0.00356**	**0.01164**

Dense Graph Performance Analysis

		Source	Destination	Running time of Dijkstra algo	Running time of Dijkstra algo with heap	Running time of Kruskal algo
Dense Graph 1	Vertex Pair 1	1958	435	0.107	0.308	6.071
	Vertex Pair 2	556	1067	0.168	0.061	5.111
	Vertex Pair 3	1621	549	0.02	0.023	3.903
	Vertex Pair 4	1566	3562	0.028	0.075	5.191
	Vertex Pair 5	1600	2459	0.051	0.021	4.881
Dense Graph 2	Vertex Pair 1	3709	1223	0.035	0.079	3.858
	Vertex Pair 2	4508	3973	0.047	0.045	4.162
	Vertex Pair 3	2709	1313	0.074	0.08	4.053
	Vertex Pair 4	2658	2207	0.045	0.089	4.459
	Vertex Pair 5	4515	2193	0.081	0.018	4.121
Dense Graph 3	Vertex Pair 1	43	2002	0.046	0.096	34.868

	Vertex Pair 2	1340	4523	0.147	0.09	10.909
	Vertex Pair 3	2163	846	0.017	0.108	11.295
	Vertex Pair 4	1798	1090	0.355	0.531	14.066
	Vertex Pair 5	4695	3617	0.111	0.032	4.647
Dense Graph 4	Vertex Pair 1	3572	4027	0.046	0.039	3.734
	Vertex Pair 2	4254	4644	0.053	0.033	3.682
	Vertex Pair 3	4881	4643	0.052	0.008	3.515
	Vertex Pair 4	116	1846	0.009	0.063	3.476
	Vertex Pair 5	4263	2665	0.025	0.016	3.438
Dense Graph 5	Vertex Pair 1	3540	4648	0.037	0.05	5.67
	Vertex Pair 2	4202	4836	0.049	0.029	6.183
	Vertex Pair 3	65	1445	0.048	0.055	6.16
	Vertex Pair 4	788	4787	0.086	0.099	4.382
	Vertex Pair 5	733	3174	0.044	0.062	4.118
Avg Running Time (in sec)				**0.07124**	**0.0844**	**6.63812**

As can be seen from the above table, in the case of **Sparse graphs**, performance of Dijkstra algorithm using heap is the best followed by Kruskal algorithm and Dijkstra algorithm without heap data structure.

Dijkstra algorithm using heap > Kruskal algorithm > Dijkstra algorithm without heap

Kruskal algorithm performs slower as compared to Dijkstra algorithm using heap, because in Kruskal we are performing heap sort on all the edges which takes time O(E log E). In Addition, after the Maximum spanning tree is constructed we are doing a BFS which takes time O(E+V) to find max bandwidth between source and destination vertex. Therefore the performance difference between Dijkstra algorithm using heap and Kruskal algorithm would be due to constants in the Big-O complexity. The worst performing algorithm in the case of Sparse graphs is the Dijkstra algorithm without heap structure which was expected since it takes time O(V^2) in the worst case as explained above in the Time complexity analysis section.

Coming to the case of **Dense graphs,** we see the performance order of the algorithm is as follow:-

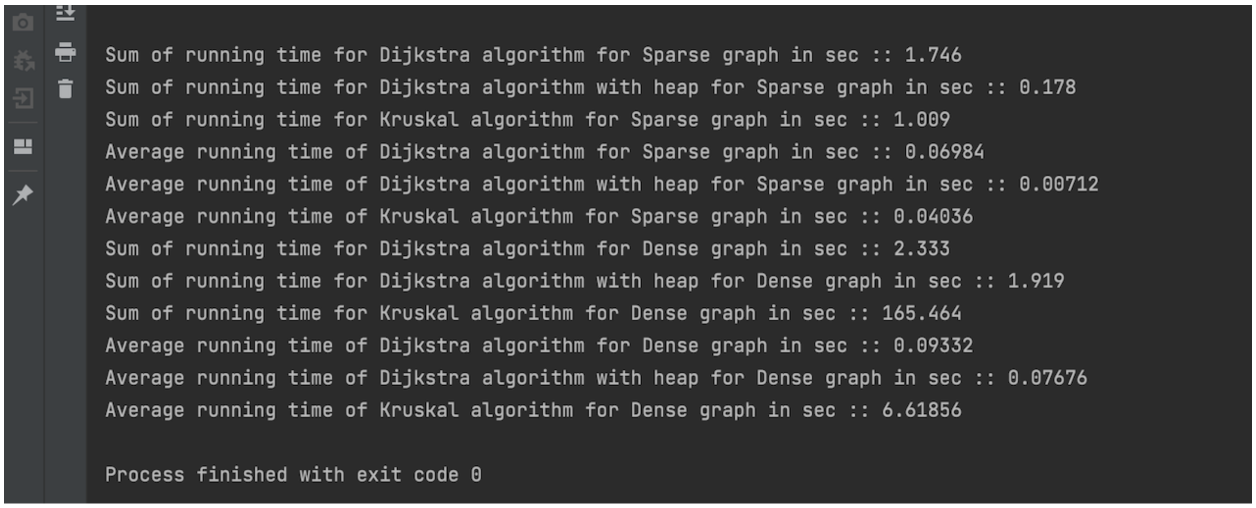
Dijkstra algorithm without heap > Dijkstra algorithm using heap >> Kruskal algorithm

Or

Dijkstra algorithm using heap > Dijkstra algorithm without heap >> Kruskal algorithm

Console output for a different test instance showing

Dijkstra algorithm using heap > Dijkstra algorithm without heap:-



```
Sum of running time for Dijkstra algorithm for Sparse graph in sec :: 1.746
Sum of running time for Dijkstra algorithm with heap for Sparse graph in sec :: 0.178
Sum of running time for Kruskal algorithm for Sparse graph in sec :: 1.009
Average running time of Dijkstra algorithm for Sparse graph in sec :: 0.06984
Average running time of Dijkstra algorithm with heap for Sparse graph in sec :: 0.00712
Average running time of Kruskal algorithm for Sparse graph in sec :: 0.04036
Sum of running time for Dijkstra algorithm for Dense graph in sec :: 2.333
Sum of running time for Dijkstra algorithm with heap for Dense graph in sec :: 1.919
Sum of running time for Kruskal algorithm for Dense graph in sec :: 165.464
Average running time of Dijkstra algorithm for Dense graph in sec :: 0.09332
Average running time of Dijkstra algorithm with heap for Dense graph in sec :: 0.07676
Average running time of Kruskal algorithm for Dense graph in sec :: 6.61856

Process finished with exit code 0
```

Dijkstra algorithm without heap performance is **comparable** in some cases to Dijkstra algorithm using heap data structure due to the density of the graph. As seen above, in this test instance, the average running time of Dijkstra algorithm with heap (0.07676s) is little better than Dijkstra algorithm without heap (0.09332s). **We know that in case of high graph densities, the number of edges, E, is comparable to V^2, giving the Dijkstra algorithm using binary heap implementation a time of $O(V^2 \log V)$.** However, for low graph densities the binary heap implementation is faster.

Reference : https://www.cosc.canterbury.ac.nz/research/reports/HonsReps/1999/hons_9907.pdf (Section 2.1)

The performance of Kruskal algorithm for dense graphs is worst due to time taken in heap sort. It sorts all the edges in the dense graph, and for dense graphs, average vertex degree is taken as 1000, i.e. the number of edges is very large compared to the vertices, it is of the order of millions and therefore heap sort performs very poorly. Thus, Kruskal algorithm using heap sort is not an optimal algorithm for finding Max Bandwidth Path in dense graphs.

Conclusion and Further Improvements

Thus we conclude that to find max bandwidth path in case of Sparse graphs, better performance can be achieved by using Dijkstra algorithm with max heap structure to store the fringes followed by Kruskal algorithm. However, in case of Dense graphs, use of Kruskal algorithm based on heap sort is not at all recommended. Instead, we can go for some other sorting algorithm such as **Counting sort** if the edge weights are limited by some constant K which is not too large. Also for dense graphs, the performance of Dijkstra algorithm with and without heap is almost comparable and either can be used to solve max bandwidth path problem.