Max Bandwidth Path Algorithms

Vicente Balmaseda

December 2, 2022

1 Introduction

The maximum bandwidth path problem is a graph problem that, given a weighted graph, a source node and a target node, seeks to calculate the path from the source to the target such that the minimum weight of the edges in the path in the path is maximized. The weights are also known as bandwidth, and the minimum bandwidth of the edges in the path is the bandwidth for the path. This problem is also known as the widest path problem or maximum capacity problem.

This problem has multiple applications, such as finding the end-to-end bandwidth between two internet nodes or finding bottlenecks in a network.

In this work, we have implemented multiple algorithms for the Max-Bandwidth Path problem using Java[1]. More specifically, we have implemented Dijkstra's algorithm without a heap structure, Dijkstra's algorithm using a max-heap, and Kruskal's algorithm. It is important to consider that we have considered the nodes of a graph to be the numbers from 0 to n-1, with n being the number of nodes in the graph.

We will first give a formal definition of the max-bandwidth problem. We will then explain the implementation details concerning the data structures used. We will continue with the implementation of the three algorithms used, and finish presenting and discussing the results obtained.

2 Preliminaries

Def 1. *The bandwidth bw of a path* $p = \{e_1, \cdots, e_n\}$ *with capacities* $\{w_1, \cdots, w_n\}$ *is* $bw = \min_{i=1}^{n} w_i$.

Def 2. *Given a weighted graph* $G = (V, E)$*, the maximum bandwidth path between two nodes* u *and* v *is a path* p *from* u *to* v *such that there is no other path between* u *and* v *with greater bandwidth. In other words, if* P *is the set of all paths between* u *and* v*, then* p *is a maximum bandwidth path if for any* $p_i \in P$ $bw(p) \geq bw(p_i)$ *with* $bw(p) = \min_{e \in p} w_e$.

[1]The code can be found at `https://github.com/vibalcam/max-bandwidth-path`

3 Data Structures

In this section, we will go over the data structures implemented to help implement the Max-Bandwidth algorithms. The data structures used are LinkedList, MaxHeap, and UnionFind. To represent the graphs, we have used an array of LinkedLists.

3.1 Linked List

A LinkedList is a data structure that holds a list of elements. This data structure is especially helpful for our purpose since an array of LinkedLists can be used to represent the adjacency list of a graph.

Our implementation of LinkedList depends on another subclass called Node that represents each element in the list. The implementation of this subclass has been done to represent an edge in a graph. Therefore, a Node object has the following attributes:

- data (int): which represents the node v in the edge (u, v)
- weight (double): represents the weight of the edge
- next (Node): references the next Node object in the linked list

Once the subclass Node is defined, we have now implemented a LinkedList with the following attributes:

- head (Node): the first Node in the linked list
- size (int): holds the size of the linked list and provides constant-time access to the list size, as opposed to $O(n)$ time if we had to iterate over the list

The following operations have been implemented (their running time is shown in parenthesis, with n being the size of the list):

- Constructor ($O(1)$): Constructs a LinkedList by initializing its size to 0 and its head to null, since it has no elements.
- Add ($O(1)$): Adds a new element to the front of the linked list by creating a new node and setting it as the new head of the linked list
- Get index idx ($O(idx)$): Retrieves the data at the given index by iterating from the head to the next reference

- Delete index idx ($O(idx)$): Deletes the node at a certain index by deleting the node and referencing $idx + 1$ from $idx - 1$ (if they exist, a special case is done for $idx = 0$)
- Size ($O(1)$): Returns the size of the list by returning the size attribute, which is updated when an add or delete operation is done
- Get index ($O(n)$): Returns the index of the given element or -1 if it is not present, by iterating over the list and checking if any element equals the searched element

3.2 TestGraphs

Each graph is represented by its adjacency list, which is implemented using an array of LinkedList with size n (the number of nodes).

We have used a set of static variables for the graph creation:

- int NUM_VERTICES = 5000 -> the number of vertices of the graphs created
- double G1_AVERAGE_DEGREE = 6 -> the average vertex degree (test graph type 1)
- double G2_PERCENTAGE_NEIGHBORS = 0.2 -> the percentage of the other nodes to which each vertex is adjacent (test graph type 2)
- double MIN_WEIGHT = 1 -> the minimum weight of the edges
- double MAX_WEIGHT = 200 -> the maximum weight of the edges

Three methods have been implemented to create test graphs: get a random cycle, get a random test graph type 1, and get a random test graph type 2. We have also implemented a shuffle method to shuffle an array randomly.

3.2.1 Shuffle Array

This method shuffles an array randomly and returns the first k elements. In other words, it returns k elements from an array in random order. It uses a seed to ensure reproducibility.

This method is an implementation of Fisher-Yates shuffle (Durstenfeld algorithm). The algorithm iteratively chooses an unsampled element by sampling uniformly an integer between 0 and the index with the last unsampled element. It then moves it to the end of the array by swapping it. This process is done until k elements have been chosen. The code for this method is presented in Listing 1. Therefore, the running time of the algorithm is $O(k)$, with $k \leq array.length$.

Even though Durstenfeld algorithm shuffles an array in place, we have chosen to include the sampled elements into a new array with length k, so the returned array has the length required.

This simplifies working with the returned array, which is important since this method is used as an intermediate step in more complex methods.

Listing 1: Java code for shuffling an array

```
private static int[] shuffleArray(int[] x, int k, long seed)
{
    // can also be done in place instead of using a return array,
       but this way we return only k elements
    int[] ret = new int[k];
    Random rnd = new Random(seed);
    for (int i = x.length - 1; i > x.length - k; i--) {
        int index = rnd.nextInt(i + 1);
        // Simple swap
        ret[x.length-i-1] = x[index];
        x[index] = x[i];
        x[i] = ret[x.length-i-1];
    }
    return ret;
}
```

3.2.2 Random Cycle

Generates a cycle with a given number of vertices and the edge weights chosen uniformly at random.

This is done by iterating over the nodes (integers from 0 to n-1) and adding the edge $(k, k+1)$ to the linked list of node k and the edge $(k+1, k)$ to the linked list of node $k+1$ (since the graph is undirected). An edge (u, v) is added to the linked list of u by adding node v to the list. Remember that each element of the linked list has attributes to represent node v (data) and edge weight (weight).

The running time of this algorithm is $O(n)$ since, for each node, we are randomly sampling an edge weight, creating a LinkedList, and making two additions to the linked lists, which are all operations done in constant time.

3.2.3 Random Graph 1

Creating a graph with random weights, a given number n of vertices and average vertex degree. The average vertex degree is ensured to be met to the extent possible. The graph generated is undirected and connected.

To ensure the graph is connected, we first generate a random cycle with n vertices. We then calculate the number of edges to generate randomly, which is $nEdges = AVERAGE\_DEGREE * nVertices/2 - nVertices$ edges (nVertices already included in the cycle).

The next step is to create an array *edges* that represents all the possible edges that can be generated. The total number of edges that can be generated is $nVertices * (nVertices - 3)$ since each vertex is already adjacent to two nodes. Each possible edge is assigned an integer from 0 to the total number of possible edges. Next, we randomly choose nEdges from the array of possible edges using our shuffling method.

Once we have the edges to be created, we iterate over all of them and add them. To do this, we have to obtain the edge to be created from the integer that represents it. Let x be the integer chosen:

- The weight of the edge is chosen uniformly at random
- $source = \left\lfloor \frac{x}{nVertices-3} \right\rfloor$ -> source node can be $[0, nVertices - 1]$ (each node has probability of 1/nVertices of being chosen)
- $target = x\%(nVertices - 3)$ -> target node can be $[0, (nVertices - 4)]$ (each node has probability of 1/(nVertices-3) of being chose)
- Source node is *source*, and target node is $(target + source + 2)\%nVertices$. This ensures that, given a source node, the target node is not itself nor one of its adjacent nodes.

By following the approach stated above, we have mapped the integers in $[0, nVertices(nVertices-3)]$ with a possible edge. We have also shown that the edges are chosen with uniform probability.

The running time of this algorithm is $O(n^2)$, since

- Generating random cycle takes $O(n)$ time
- Generating array with possible edges takes $O(n^2)$
- Shuffling the array of possible edges takes $O(n^2)$
- For each of the chosen edges, we add nodes to the corresponding LinkedLists, which takes constant time. This operation is done m times, with $m = G1\_AVERAGE_DEGREE * nVertices/2-nVertices$. Therefore this takes $O(G1\_AVERAGE_DEGREE/2*n) = O(3n) = O(n)$. In the worst case scenario this could be $O(n^2)$ (maximum vertex degree is n-1).

3.2.4 Random Graph 2

Creating a graph with random weights, a given number n of vertices, and a percentage of the nodes to which each vertex is adjacent. The percentage of neighbors is met to the extent possible. The

graph generated is undirected and connected.

To ensure the graph is connected, we first generate a random cycle with n vertices. We then build an adjacency matrix which makes it possible to check if an edge already exists in constant time $O(1)$ as opposed to linear time $O(n)$ if we used the implementation in LinkedList.

We first create an array of length n that we will use to choose the neighbors, and we calculate the number of neighbors for each vertex ($nNeighbors = \lfloor (nVertices - 1) * G2\_PERCENTAGE\_NEIGHBORS \rfloor$). We then iterate over each vertex u in the graph and do the following:

1. Get the size of the node's LinkedList to calculate how many neighbors have to be generated
2. Randomly choose the required number of neighbors by shuffling the helper array of length n
3. For each node v of the randomly chosen nodes
   - Check if the edge already exists
   - Check if the node v has already reached its limit of neighbors
   - Check if $v == u$

   If v is a valid neighbor, add the edge with a randomly sampled weight. Otherwise, try with the next node. If a cycle is completed, that means that there are no valid nodes. The only reason for this is that all nodes already have the required amount of neighbors, so we end the algorithm.

The running time is bounded by the following:

- Get random cycle takes $O(n)$
- Create adjacency matrix takes $O(n^2)$
- Create array with possible neighbors takes $O(n)$
- For loop runs n times for each node (steps as mentioned above):
  - Step 1. Takes $O(1)$ time
  - Step 2. Takes at most $O(n)$ time
  - Step 3. Iterates over at most $nNeighbors = \lfloor (nVertices - 1) * G2\_PERCENTAGE\_NEIGHBORS$ nodes and takes at most $O(n)$ time for each (if it completes a cycle). Thus, the total running time of the loop is $O(n^2)$ in the worst case.

Therefore, an upper bound for the running time complexity is $O(n^3)$. However, this is not a tight bound since obtaining a valid node does not take $O(n)$ time for every node. Moreover, when it completes a cycle, the algorithm finishes.

In this algorithm, two small changes have been used to improve its runtime:

- Having a size attribute in the LinkedList allows obtaining the size of the current neighborhood in constant time instead of linear time. This is useful since we call the size function between $n * nNeighbors$ and $n^2 * nNeighbors$ times
- Building the adjacency matrix takes $O(n^2)$ time but allows checking the existence of an edge in $O(1)$ time as opposed to $O(n)$ time. This is useful since this operation is also done between $n * nNeighbors$ and $n^2 * nNeighbors$ times

3.3 MaxHeap

A MaxHeap is a complete binary tree in which the value of each node is greater or equal than the value of the children of that node. Thus, each node in a MaxHeap is the root of a subtree that is also a MaxHeap.

We have implemented a MaxHeap using four attributes:

- heap (int array): array that represents the binary tree used for the max heap. Its elements are the nodes of the tree (thus, index 0 is the root)
- values (double array): stores the values of the elements in the heap. Thus, values[idx] is valid if the node idx is in the heap
- position (int array): helper array such that position[idx] returns the position in the heap of node idx. It allows retrieving the position in the heap for an element in $O(1)$ instead of $O(n)$, which allows deleting an element in $O(\log n)$
- last (int): records the last index being used in the heap

Given our implementation, setting a heap element requires modifying the three arrays. Moreover, to obtain the value of an element in the heap, we first obtain that element from the heap array and then obtain its value from the values array.

The following operations have been implemented (their running time is shown in parenthesis, with n being the size of the list):

- Constructor ($O(1)$): An empty MaxHeap is created in constant time by initializing the arrays and by setting last to -1. The only parameter needed is its size, since for our implementation, a MaxHeap of size n can hold nodes in the range $[0, n-1]$ (it has been implemented to hold fringer nodes for Dijkstra's algorithm)
- IsEmpty ($O(1)$): Returns whether the MaxHeap is empty by checking if the last valid index is -1

- Max ($O(1)$): Returns the maximum element in the MaxHeap, which is index 0
- Add ($O(\log n)$): Inserts a new element to the MaxHeap and fixes the heap. This implements the algorithm seen in class
- Delete index ($O(\log n)$): Deletes element at a certain index from the heap and fixes the heap. This implements the algorithm seen in class
- Delete element ($O(\log n)$): Deletes an element from the heap and fixes the heap. It uses the position array to obtain the index of the element and then runs delete index
- Pop ($O(\log n)$): Returns the maximum element in the MaxHeap and deletes it

3.3.1 Heapsort

Heapsort uses a MaxHeap to sort an array of doubles. It is important to mention that our heapsort implementation does not return the sorted array but the array obtained by an argsort operation, i.e., the array of indices in sorted order.

The main idea is to construct a MaxHeap object with the given values and iteratively pop the maximum. Iteratively popping the maximum takes $O(n \log n)$, since we are adding n elements and each addition takes $O(\log n)$ time. This will also be the overall running time for heapsort.

However, in practice, we can speed up the algorithm by building the MaxHeap carefully. Building a MaxHeap from a given array can be done in two ways:

- Construct an empty MaxHeap and iteratively add the elements into it. The running time of this method is $O(n \log n)$ since we are adding n elements, and each addition takes $O(\log n)$ time.
- We initialize the three arrays of MaxHeap with the given array and then fix the MaxHeap using the MaxHeapify method to fix every subtree starting from the parent of the last leaf and going up (since the leaves are already valid MaxHeaps). MaxHeapify fixes the subtree rooted at a certain element to convert it into a MaxHeap. This method is presented on page 157 of the book [1, p. 150]. As mentioned in the book, it takes $O(n)$ time to build a MaxHeap through this method. Thus, we have used this method in our implementation.

3.4 UnionFind

UnionFind is a data structure that implements the union, find, and makeSet operations, and is used for Kruskal's algorithm. Each set in this data structure is saved as a tree and is represented by the tree's root.

The implementation has been done using two attributes:

- dad (int array): array that contains for each node its parent in the UnionFind tree
- rank (int array): array with the rank of each node

The UnionFind data structure implements the following operations as seen in class:

- Constructor ($O(1)$): Creates an empty UnionFind of a certain size
- MakeSet ($O(1)$): Makes an element the root of its tree
- Union ($O(1)$): Merges two trees into a single tree. This operation implements union by rank
- Find: Returns the tree's root containing a certain element. This operation implements find with path compression, meaning that all nodes on the path from x to the root become children of the root after the find operation

As seen in class, the time complexity for m Union-Find-MakeSet operations using a UnionFind data structure is $O(m \log^* n)$.

4 Path Algorithms

In this section, we will present the algorithms used to solve the Max Bandwidth problem: Dijkstra's and Kruskal's algorithms.

4.1 Dijkstra's Algorithm

Dijkstra's algorithm calculates the max-bandwidth path from a source node to all the other nodes in the graph. It does so by iteratively calculating the max-bandwidth path for the nodes adjacent to the in-tree nodes.

Our implementation calculates two arrays:

- dad: array with the parent node of each node in the max bandwidth path
- bw: index idx holds the bandwidth value of the max-bandwidth path from source to node idx

Our implementation takes an array of LinkedLists that represents a graph´s adjacency list and a source node. It then calculates the dad and bw arrays and returns a Dijkstra object which can be used to obtain the max-bandwidth path from the source node to any other target node. By doing so, if we wanted to calculate the max-bandwidth path from a single source to multiple targets, we do not need to rerun the algorithm.

4.1.1 Dijkstra's Algorithm Summary

We have already seen in class how Dijkstra's algorithm can be used to obtain max-bandwidth paths. Below, we present an overview of the algorithm without a heap structure:

1. Initialize status, dad and bw arrays
2. Set source node as in-tree and its bandwidth as positive infinity
3. Set values for nodes adjacent to the source node as fringer nodes and update its bw (capacity of the node) and dad values
4. It then iterates until there are no fringer nodes left. In each iteration, it does the following:
   (a) Get fringer v with maximum bw value. Since we are not using a heap, this is done by iterating over all nodes.
   (b) v is set as in-tree
   (c) For all $w \in \mathcal{N}(v)$, set w as fringer and update its bw (minimum between the capacity of w and the bandwidth of the path from s to v) and dad values. If w is already a fringer, then we only update it if doing so improves its bw

The running time for this algorithm using an array to hold fringers is:

- Step 1 does not need to be done explicitly since, in Java, arrays are automatically initialized to 0
- Step 2 takes constant time
- Step 3 takes constant time for each node. Thus, it takes $O(degree(s) = O(n)$
- Step 4 iterates n-1 times since the graphs are fully connected
  - Step 4.1 takes $O(n)$ time
  - Step 4.2 takes constant time
  - Step 4.3 takes $O(degree(v)$
  - Therefore, step 4 takes $O(n^2)$ in total

Therefore, its total running time is $O(n^2)$.

Step 4.1, calculating the maximum fringer, can be improved by keeping the fringer nodes in a Doubly-Linked List and deleting from it the fringer chosen. Doing so could be done in constant time, reducing the list size by one on each iteration, thus making future computations faster. This same process can also be accomplished using our LinkedList by keeping track of the previous element to the maximum, which would allow deleting the maximum in constant time by modifying the previous Node's next reference.

4.1.2 Dijkstra + MaxHeap

This algorithm can be easily improved by using a MaxHeap to hold fringer nodes. To do so, we have implemented Dijkstra's algorithm using the MaxHeap data structure we have already discussed. The main modifications that have to be done with respect to the algorithm already discussed are the following:

1. A MaxHeap of length n is constructed
2. No changes
3. We add each of the source node's neighbors to the MaxHeap structure
4. Iterate until the MaxHeap is empty
   (a) Pop from the MaxHeap (get maximum value node and delete it)
   (b) No change
   (c) If w is not a fringer, add it to the MaxHeap. If w is already a fringer and its value is being updated, we first delete it from the MaxHeap and then add it back

These modifications change the running time of the algorithm in the following way:

- Step 3 adds to the MaxHeap, which takes $O(\log n)$ for each addition. Thus, step 3 now takes $O(degree(s) \log n) = O(n \log n)$
- Step 4.1 now takes $O(\log n)$ since we are using the pop operation, which removes the maximum from the MaxHeap
- Step 4.3 takes $O(\log n)$ since we have to either add or delete and then add an element
- Overall, step 4 now takes $O(n \log n)$ time

Thus, after these modifications, the running time of Dijkstra's algorithm with MaxHeap is $O((n + m) \log n) = O(m \log n)$ (since the graph is connected), instead of $O(n^2)$ which we obtained using an array for fringer nodes.

Step 4.2 can be improved when updating the value of a fringer node by implementing an update operation in MaxHeap. This operation, instead of deleting, which takes $O(\log n)$, and then adding, which again takes $O(\log n)$, updates the value for an element and then fixes the MaxHeap starting on that element. By doing so, asymptotically, it still takes $O(\log n)$, but is doing half the work, which can make a difference in our use case.

4.1.3 Get source-target path

Once the algorithm has finished calculating the dad and bw arrays, we can easily obtain the max-bandwidth path from the source to any given target. To obtain the max-bandwidth value, we can get the value from the bw array in constant time.

Instead, if we want to obtain the path, we need to iterate over the dad array. In other words, we start at our target node, get its parent node from the dad array, and add it to the path. We keep doing this until the source node is reached. The addition of nodes to the path is done in reverse order. Therefore, we can get the path in $O(n)$ time since the max-bandwidth path is a simple path.

4.2 Kruskal's Algorithm

Kruskal's algorithm calculates the max-bandwidth path by building a maximum spanning tree. Therefore, it can be used to obtain the max-bandwidth path for all pairs of nodes. It does so by calculating a max spanning tree represented as an array of LinkedLists.

- maxSpanningTree: maximum spanning tree where the tree path between any two of its nodes is a maximum bandwidth path between those two nodes

Our implementation takes an array of LinkedLists representing a graph´s adjacency list. It then calculates the maxSpanningTree array and returns a Kruskal object which can be used to obtain the max-bandwidth path between any source and target nodes. By doing so, if we wanted to calculate the max-bandwidth path between multiple pairs in the same graph, we just need to run Kruskal's algorithm once and then run Depth First Search (DFS) once for each pair. This is useful since DFS takes less time to run than Kruskal's.

4.2.1 Kruskal's Algorithm Summary

We have already seen in class how Kruskal's algorithm can be used to obtain max-bandwidth paths. However, there is an implementation detail that has to be covered before going into the actual algorithm. In Kruskal's algorithm, we first need to sort the edges by their capacity. To do so, we first need to obtain a list of unique edges from the graph's adjacency list. This is done by initializing an array of size $\frac{n(n-1)}{2}$ (the maximum number of edges), iterating through each of the nodes, and for each LinkedList of each node iterating through the edges and adding the edge (i, j) only if $i > j$. By doing so, we make sure we are not adding duplicates. The information of each edge is added into two arrays: the edge weight is added into an edge weight array that will be sorted, and the node information (nodes i and j of edge (i, j) is added into a second array. Doing so allows using an array to sort the edges by weight and easily access their information for Kruskal's algorithm. It

is important to mention that our heapsort implementation does not return the sorted array but the array obtained by an argsort operation, i.e., the array of indices in sorted order.

Below, we present an overview of the algorithm:

1. Initialize UnionFind data structure
2. Get the list of edges from the graph and perform makeSet operation on UnionFind for each node. Since getting the list of edges requires iterating over the nodes, we also do a makeSet operation, thus combining both steps into the same loop
3. Sort the list of edges by weight in non-increasing order using heapsort
4. Iterate over the sorted list of edges and do the following operations:
   (a) Check if both endpoints of the edge are in the same tree. This is done by doing a find operation on the UnionFind object for both edges and checking if the root of their trees is the same
   (b) If both endpoints are not in the same tree, add the edge to the spanning tree and do a union operation on the trees of the endpoints.

The running time for this algorithm is:

- Step 1 takes constant time
- Step 2 requires iterating over all nodes and edges. For each edge, it takes constant time to add its information into the edge weight and edge info arrays. For each node, the makeSet operation takes constant time. Therefore, it takes $O(n + m)$ time
- Step 3 takes $O(m \log n)$. Remember that our implementation builds the MaxHeap in linear time, which makes this step faster (although it does not affect its asymptotic running time)
- Step 4 iterates m times. This step does at most $3m$ union-find operations (for each edge, it does two finds and a union). Thus, it takes $O(m \log^* n)$ in total. All other operations (adding an edge to the spanning tree) take constant time. Thus, the total running time of step 4 is $O(m \log^* n)$

Therefore, its total running time is $O(m \log n)$.

4.2.2 Get source-target path

Once the algorithm has finished calculating the max-spanning tree, we can easily obtain the max-bandwidth path for any pair of edges. In contrast with Dijkstra's algorithm, obtaining the max-bandwidth value requires first obtaining the max-bandwidth path. Thus, the running time for both

operations will be the same. Obtaining the path can be done using Breadth First Search (BFS) or Depth First Search (DFS) on the max-spanning tree in linear time ($O(n)$ since the graph is a tree). For our implementation, we have used DFS. We have implemented DFS using the following three arrays:

- dad (int array): keeps track of the parent of each node in the path from the source
- visited (int array): keeps track of whether a certain vertex has already been visited
- bw (double array): keeps track of the max-bandwidth value of the path from the source to each edge

Once DFS finishes, and we have obtained the three arrays mentioned, we can obtain the max-bandwidth value and path in the same manner as in Dijkstra's algorithm. The bandwidth value can be obtained in constant time from the bw array in constant time. The path can be obtained by iterating over the dad array, building the path in reverse order, which takes at most $O(n)$ time.

5 Results

To test the three algorithms implemented, we have randomly generated 5 graphs of each type as explained in Section 3.2. For each of the graphs generated, we randomly selected 5 pairs of nodes and calculated the max-bandwidth path between them using each algorithm. We have set the random seed to ensure reproducibility to the extent possible (running time depends on other factors such as system performance).

Tables 1 and 2 present the results for Dijkstra's algorithm without using a MaxHeap. Tables 3 and 4 present the results for Dijkstra's algorithm using a MaxHeap. Kruskal's total time results are presented in tables 5 and 6, and the time results just to obtain the paths are presented in Table 7 and 8.

Table 1: Graph type 1: Dijkstra without heap (ms)

Duration(ms)	Pair 1	Pair 2	Pair 3	Pair 4	Pair 5	Mean ms
Graph 1	212,0	157,0	99,0	234,0	145,0	169,4
Graph 2	114,0	111,0	173,0	141,0	153,0	138,4
Graph 3	117,0	208,0	193,0	140,0	121,0	155,8
Graph 4	95,0	74,0	123,0	106,0	208,0	121,2
Graph 5	218,0	161,0	126,0	141,0	89,0	147,0
Mean ms	151,2	142,2	142,8	152,4	143,2	146,4

Before going into the results' discussion, it is important to remember that both graph times

Table 2: Graph type 2: Dijkstra without heap (ms)

Duration(ms)	Pair 1	Pair 2	Pair 3	Pair 4	Pair 5	Mean ms
Graph 1	310,0	231,0	1666,0	69,0	118,0	478,8
Graph 2	389,0	423,0	2904,0	534,0	133,0	876,6
Graph 3	356,0	299,0	307,0	337,0	397,0	339,2
Graph 4	449,0	226,0	189,0	182,0	154,0	240,0
Graph 5	789,0	163,0	170,0	171,0	112,0	281,0
Mean ms	458,6	268,4	1047,2	258,6	182,8	443,1

Table 3: Graph type 1: Dijkstra with heap (ms)

Duration(ms)	Pair 1	Pair 2	Pair 3	Pair 4	Pair 5	Mean ms
Graph 1	55,0	12,0	6,0	8,0	7,0	17,6
Graph 2	5,0	5,0	5,0	3,0	5,0	4,6
Graph 3	5,0	15,0	4,0	5,0	4,0	6,6
Graph 4	6,0	4,0	3,0	6,0	5,0	4,8
Graph 5	16,0	18,0	5,0	6,0	5,0	10,0
Mean ms	17,4	10,8	4,6	5,6	5,2	8,7

Table 4: Graph type 2: Dijkstra with heap (ms)

Duration(ms)	Pair 1	Pair 2	Pair 3	Pair 4	Pair 5	Mean ms
Graph 1	278,0	56,0	87,0	45,0	58,0	104,8
Graph 2	341,0	58,0	78,0	70,0	54,0	120,2
Graph 3	344,0	313,0	278,0	289,0	292,0	303,2
Graph 4	389,0	151,0	145,0	137,0	138,0	192,0
Graph 5	777,0	92,0	82,0	89,0	75,0	223,0
Mean ms	425,8	134,0	134,0	126,0	123,4	188,6

Table 5: Graph type 1: Kruskal build and get path (ms)

Graph 1	Graph 2	Graph 3	Graph 4	Graph 5	Mean ms
398,0	276,8	160,6	136,8	146,6	223,8

Table 6: Graph type 2: Kruskal build and get path (ms)

Graph 1	Graph 2	Graph 3	Graph 4	Graph 5	Mean ms
14417,4	19069,4	14334,4	12315,8	16493,2	15326,0

have the same number of nodes, but a different number of edges. Graph type 1 has

$$G1\_AVERAGE\_DEGREE * nVertices/2$$

Table 7: Graph type 1: Kruskal get path (ms)

Graph 1	Graph 2	Graph 3	Graph 4	Graph 5	Mean ms
1,4	0,4	1,4	0,8	0,6	0,9

Table 8: Graph type 2: Kruskal get path (ms)

Graph 1	Graph 2	Graph 3	Graph 4	Graph 5	Mean ms
0,8	1,8	0,6	2,0	0,8	1,2

edges, which in our case is $\frac{6*5000}{2} = 15,000$. Graph type 2 has

$$\frac{nVertices}{2} * \lfloor (nVertices - 1) * G2\_PERCENTAGE\_NEIGHBORS \rfloor$$

which in our case is $\frac{5000}{2} * \lfloor 4999 * 0.2 \rfloor = 2,499,500$. Therefore, graph type 2 has the same quantity of nodes but several magnitudes more number of edges.

A summary of the results can be found in Table 9. By looking at this table, we can observe that Dijkstra's algorithm using a heap solves the max-bandwidth problem in the minimum time for both graphs. However, the difference between using a heap and not using a heap becomes smaller as the number of edges increases. This is explained by their running time, since not using a heap takes $O(n^2)$ while using a heap takes $O(m \log n)$. In other words, the algorithm's running time not using a heap only depends on the number of nodes, while the algorithm's running time using a heap depends on the number of edges. This implies that using a heap the running time can be as bad as $O(n^2 \log n)$ (with $m = n^2$), in which case the heap algorithm would have a worse running time. This is also observed when looking at the data in the summary table. Even though both algorithms take more time in graph type 2 than 1 due to the increase in the number of edges, the increment in running time is a lot steeper for Dijkstra with heap (the no heap version's running time is approximately 3 times greater for graph type 2, while it is 22 times greater for the heap version). Therefore, we have shown that Dijkstra without heap is a better option for dense graphs, while Dijkstra with heap is a better option for sparse graphs.

Kruskal's algorithm is similar to Dijkstra's with heap in that it also performs better in sparse graphs. In our experiment, Kruskal's running time is 68 times greater for graph type 2 than graph type 1, which is an even greater multiplier than Dijkstra's with heap. Kruskal's asymptotic running time is the same as Dijkstra's with heap. However, its running time has a much higher constant factor, which is observed by the fact that its running time is much greater than that of Dijkstra's with heap (26 times greater for graph type 1 and 81 times for graph type 2). However, the running times shown for Kruskal's algorithm combines both building the max spanning tree and finding a path between a pair of nodes in the graph. This has been done in order to compare Kruskal's algorithm

with Dijkstra's in solving the same problem. However, Kruskal's algorithm has an advantage with respect to Dijkstra's. Once a maximum spanning tree is built for a given graph, obtaining the maximum bandwidth path between any pair of edges is done very fast, and it does not depend on the number of edges of the original graph (since the path is obtained running DFS on a maximum spanning tree, which has n-1 edges). Therefore, according to our experiment, depending on the number of pairs to be searched on the same graph, using Dijkstra or Kruskal is faster:

- A single pair: Dijkstra is faster (Dijkstra has faster build time)
- A single source and multiple targets: Dijkstra is faster. Dijkstra has faster build time, and obtaining paths from the same source takes the same time for both (Dijkstra has to iterate over the dad array, while Kruskal runs DFS once and then reuses the dad array in the same way as Dijkstra)
- Multiple sources and targets: Kruskal is faster (depends on the number of pairs). Kruskal needs to run DFS for each pair, while Dijkstra has to run the algorithm for each distinct source (slower build time is compensated by faster calculation of the path)

Moreover, let us assume we want to get the max-bandwidth path for k pairs of nodes on the same graph (with all nodes being different, so we cannot reuse Dijkstra's calculation). In that case, according to our experiment, using Kruskal becomes a faster option when k is greater than 82 (164 nodes) for graph type 2 (denser graph) and 30 (60 nodes) for graph type 1 (sparser graph). These numbers represent 3.28% and 1.2% of the nodes in the graph respectively. In other words, as the graph becomes denser, the number of pairs for Kruskal to be faster than Dijkstra with heap becomes higher, but it does not grow as much as the number of edges (graph type 2 has 167 times more edges than graph type 1, while the percentage of nodes to break-even is only 2.7 times greater). This implies that, if we want to obtain the max-bandwidth paths for multiple pairs of distinct nodes, even if they represent a small percentage of the total nodes, using Kruskal is the way to go. If otherwise we want to obtain the max-bandwidth path for a very small set of pairs or all pairs share a small number of nodes, then Dijkstra with heap is the way to go.

Table 9: Summary duration (ms)

	Dijkstra	Dijkstra-heap	Kruskal build	Kruskal path
Graph Type 1	146,4	8,7	223,8	0,9
Graph Type 2	443,1	188,6	15326,0	1,2
Running Time	$O(n^2)$	$O(m \log n)$	$O(m \log n)$	$O(n)$

6 Conclusions and Future Work

Our implementation can be improved in several ways. Software-wise, our implementation can improve by including more checks to ensure that the operations on the data structures make sense. For example, currently trying to search in a max heap, an element that has not been added returns a value (which is incorrect). This can be improved by keeping track of which nodes have been added to the heap. Other improvements refer to the creation of random test graphs. For example, graph type 1 could implement the configuration method instead of the current implementation, which allows setting different degrees for each node. Lastly, our algorithms' speed and computational efficiency can also be improved. An example of such improvement is the sorting for Kruskal's algorithm. Instead of using heapsort, a faster sorting algorithm can be used. We could even look at implementing a linear sorting algorithm. Since we want to sort double values within a fixed range (we have set the range for the weight values), we can convert them to integers by specifying a fixed number k of decimals to consider, multiplying them by 10^k, and taking the floor of the number.

In this work, we have implemented three algorithms for solving the maximum bandwidth path problem: Dijkstra without a heap structure, Dijkstra with a heap structure, and Kruskal's algorithm. We have tested these algorithms in several random graphs, which fall into two types: a denser graph with 5,000 nodes and approximately 2,499,500 edges and a sparser graph with 5,000 nodes and 15,000 edges. Our experiments confirm that using Dijkstra without a heap is slower than using Dijkstra with a heap for both graphs and that, as the number of edges increases, this difference becomes smaller. Moreover, the asymptotic running time analysis of both algorithms tells us that, as the graph becomes complete, the algorithm without a heap becomes faster than the version with a heap. Our experiments also show that using Dijkstra's algorithm with a heap is faster than using Kruskal. However, we have discussed that using Kruskal's algorithm can be a better option if we want to obtain the max-bandwidth paths for multiple pairs of distinct nodes, even if they represent a small percentage of the total nodes. If otherwise we want to obtain the max-bandwidth path for a very small set of pairs or all pairs share a small number of nodes, then Dijkstra with the heap is faster.

References

[1] Thomas H. Cormen, ed. *Introduction to Algorithms*. 3rd ed. Cambridge, Mass: MIT Press, 2009. 1292 pp. (cit. on p. 8).