# Project Report
# Analysis of Algorithms

Sai Ramana Reddy

## 1 Introduction

The Maximum Bandwidth Path problem is the problem of finding a path between two given vertices in a weighted graph wherein the weight of the minimum edge is maximised. This problem has many applications in varied areas. For example, in the context of Network routing algorithms, where there are multiple paths of different capacities between two routers, it is optimal to send the packet along the maximum bandwidth path to maximise throughput. Similarly in the context of Transportation planning, where goods and people need to be sent from one city to another, it is optimal to choose the maximum bandwidth path.

In this project, we aim to test and analyze the running times of various algorithms on the Maximum Bandwidth Path problem. Specifically, we will be testing the following algorithms on randomly generated graphs:

- Modification of Dijkstra's Algorithm without using any heap
- Modification of Dijkstra's Algorithm using a Max-heap
- Modification of Kruskal's Algorithm

We evaluate above algorithms on two different types of graphs:

- "Sparse" Graphs (call $G_1$), where the average vertex degree is 6
- "Dense" Graphs (call $G_2$), where each vertex is adjacent to around 20% of the other vertices.

## 2 Method

As described above, we evaluate two variations of Dijkstra's algorithm and the Kruskal's algorithm on the Maximum Bandwidth problem. In case of Dijkstra's, at each iteration in the algorithm, we need to choose the fringer with maximum bandwidth, and "relax" it's neighbours accordingly. We also maintain a parent array which is used to retrace the Maximum Bandwidth path between the two vertices.

In case of Kruskal's, at each iteration, we choose the edge with the maximum weight and unify the sets containing the vertices of that edge to form the Maximum Spanning Tree. Then, we choose the unique path between the vertices in the MST to construct the Maximum Bandwidth Path.

Note that it is possible to have more than one maximum bandwidth paths, however, the value of maximum bandwidth is unique between two vertices. We briefly describe the psuedocode for these algorithms below:

**Dijkstra-BW**$(G(V,E),s,t)$

```
1  // Without using a heap
2  status[v] ← unseen;   b_width[v] ← 0;   dad[v] ← 0   ∀ v from 1 to n
3  status[s] ← in_tree;   b_width[s] ← ∞;   dad[s] ← −1
4  for (s, w) ∈ E
5        status[w] ← fringer;   b_width[w] = bw(s, w);   dad[w] ← s
6  while ∃v ∋ status[v] == fringer
7        pick the fringer v with the largest b_width[v];   status[v] ← in_tree
8        for (v, w) ∈ E
9              if status[w] == unseen
10                   status[w] ← fringer;   dad[w] ← v
11             elseif status[w] == fringer and b_width[w] < min(b_width[v], bw(v, w))
12                   b_width[w] ← min(b_width[v], bw(v, w))
13                   dad[w] ← v
14 return b_width[t] and retrace path using dad[...]
```

**Dijkstra-BW**$(G(V,E),s,t)$

```
1  // Using a heap
2  status[v] ← unseen;   b_width[v] ← 0;   dad[v] ← 0   ∀ v from 1 to n
3  status[s] ← in_tree;   b_width[s] ← ∞;   dad[s] ← −1
4  H ← ∅
5  for (s, w) ∈ E
6        status[w] ← fringer;   b_width[w] = bw(s, w);   dad[w] ← s
7        Insert(H, w)
8  while ∃v ∋ status[v] == fringer
9        pick the fringer v with the largest b_width[v];   status[v] ← in_tree
10       Delete(H, v)
11       for (v, w) ∈ E
12             if status[w] == unseen
13                   status[w] ← fringer;   dad[w] ← v
14                   b_width[w] ← min(b_width[v], bw(v, w))
15                   Insert(H, w)
16             elseif status[w] == fringer and b_width[w] < min(b_width[v], bw(v, w))
17                   Delete(H, w)
18                   b_width[w] ← min(b_width[v], bw(v, w))
19                   dad[w] ← v
20                   Insert(H, w)
21 return b_width[t] and retrace path using dad[...]
```

**Kruskal-BW**$(G(V,E),s,t)$

```
1  Sort E in non-increasing order of edge weights
2  T.V ← V;   T.E ← ∅
3  S ← MakeSet(|V|)
4  for (u, v) ∈ E
5        if FindSet(S, u) ≠ FindSet(S, v)
6              T.E ← T.E ∪ {(u, v)}
7              Union(S, u, v)
8  return Path and BW from s to t in T using BFS
```

The complexities of the above algorithms on connected graphs ($m \geq n - 1$) are as follows: (where $n$ and $m$ represent the number of vertices and edges respectively)

| Algorithm | Time Complexity |
|---|---|
| Dijkstra's without heap | $O(n^2)$ |
| Dijkstra's using heap | $O(m \log n)$ |
| Kruskal's | $O(m \log n)$ |

Table 1: Complexities of the algorithms

For the sake of simplicity, we ensure that the random graphs generated, are simple connected graphs. Furthermore, the weights assigned to the edges in $G_1$ and $G_2$ are also randomly chosen. More details about the implementation of sampling and connectedness are described in the Implementation section. In case of $G_1$, given that the average vertex degree is 6, we have,

$$\begin{aligned} \frac{\sum_{v=1}^{n} deg(v)}{n} &= 6 \\ \implies \frac{2m}{n} &= 6 \\ \implies m &= 3n \end{aligned}$$

Thus, we choose $m = 3n$ edges for $G_1$, by randomly sampling pairs of vertices and connecting them.

In case of $G_2$, we allow for $\pm 5\%$ variation in the degree of each vertex, i.e., we randomly sample the number of neighbours from $[n/5 - 50, n/5 + 50]$ for each vertex. The edge weights are also randomly sampled from a uniform distribution. Please refer to the implementation section for the method of choosing the neighbours themselves.

# 3 Implementation

We use the `C++17` Programming language for implementing the above algorithms. `C++` is chosen because of its powerful syntax, performance and support for Object-Oriented Semantics. Please refer to the README.md file attached along with the code for more details on running the code.

## 3.1 Data Structures

We implement the following data structures for the purposes of this project. The headers for them are shown in brackets.

- Dynamic Array (`vector.hh`)
- Heap (`heap.hh`)
- Map (`map.hh`)
- Queue (`queue.hh`)
- Disjoint Forest or MakeSet-FindSet-Union (`graph.hh`)

Below, we briefly describe the function and utility of these data structures:

We implement a dynamic array called `vector` similar to the one provided by the standard library. This class is mainly useful for storing the adjacency list of the graph (i.e. `vector<vector<int>>`), where the number of neighbours are not known in advance, so a static array cannot be used. The dynamic memory allocation is based on `std::allocator`. Similar to the standard, we implement and use `push_back`. Below, we provide the complexities of the operations on the `vector` class:

We implement a Max-Heap data structure based on `vector` to store the fringer vertices in the second variant of the Dijkstra's Algorithm, and also for sorting the edges in HeapSort. As described in the problem statement, we maintain three `vectors`: `D, H, P` for various operations on the Heap.

We implement a Queue data structure based on Linked Lists. A Queue is useful for performing BFS and retracing the Maximum Bandwidth Path between two given vertices.

We implement a Map data structure (based on AVL-Trees) to store the edges and avoid duplicate edges. The utility of this data structure in our project is similar to set, however, we use a map for associating the edge weights with the edges. This is useful for the Kruskal's Algorithm, where we need to sort the edges in non-increasing order of their weights. Note that we could also have used an adjacency matrix for this purpose. However, this method would be space inefficient. Moreover, this data structure is not used after constructing the graph. Thus, the time complexity of using a map is immaterial for the analysis of the Maximum Bandwidth Path algorithms.

For Kruskal's algorithm, we need to maintain a Disjoint forest data structure, so that we can quickly find if two elements are in the same set and unify them. We implement the MakeSet-FindSet-Union operations using the Path Compression and Union by Rank heuristics to obtain the best performance.

Below, we present the complexities of some important methods on these data structures:

| Data Structure | Operation | Time Complexity |
|---|---|---|
| `vector` | `push_back` | $O(1)$ or $O(size)$ |
| `vector` | `constructor: vector(n)` | $O(n)$ |
| `heap` | `insert` | $O(\log(size))$ |
| `heap` | `extract_max` | $O(\log(size))$ |
| `heap` | `delete` | $O(\log(size))$ |
| `heap` | `constructor: vector(n)` | $O(n)$ |
| `queue` | `push` | $O(1)$ |
| `queue` | `pop` | $O(1)$ |
| `map` | `insert` | $O(\log(size))$ |
| `map` | `search` | $O(\log(size))$ |
| `dsu` | `make_set` | $O(n)$ |
| `dsu` | `find` | $O(1)$ |
| `dsu` | `union` | $O(1)$ |

Table 2: Complexities of the operations on the data structures

## 3.2 Sampling and Graph Construction

The `C++` Language Library provides a useful class called `std::uniform_int_distribution` which can be used to randomly and uniformly sample integers from an interval $[a, b] \ni a, b \in \mathbb{Z}$. The Standard Library also provides function for generating a random permutation of a given sequence, called `std::shuffle`. Using these utilities, we can achieve our goal of creating $G_1$ and $G_2$ as follows:

For forming a connected graph, we first generate a permutation of the sequence $[0, 1, ..., n-1]$ using `std::shuffle`. Let the resulting permutation be $[p_0, p_1, ..., p_{n-1}]$. We then connect consecutive vertices, i.e. $(p_i, p_{i+1})$, for $i = 0, 1, ..., n-2$ and also $(p_{n-1}, p_0)$ to create a cycle containing all vertices. Since we are using an adjacency list representation, we also add the edge $(u, v)$ whenever we add $(v, u)$ to make the graph undirected.

We maintain an object for uniformly sampling vertices, called `vertex_gen` which samples from the interval $[0, n-1]$. Similarly, for the edge weights, we create a `std::uniform_int_distribution`

object to sample from $[0, 2^{31} - 1]$.

In case of $G_1$, we need to have $3n$ edges in the graph. We have already added $n$ edges when we created a cycle containing all vertices. So we need to additionally add $2n$ edges to $G_1$. To do this, we sample two vertices using `vertex_gen` and add the edge $(u, v)$ to the graph if $u \neq v$ and it does not already exist. We repeat this process until we have $3n$ total edges in the graph. To check if an edge already exists in the graph, we use the map data structure described above, which does not allow duplicates.

In case of $G_2$, we follow the same method for creating a cycle as for $G_1$. We know that each vertex should be adjacent to around $n/5$ other vertices. Since $n/5 = 1000 \gg 2$, we ignore the already created two edges in the cycle. Thus, for each vertex, we first randomly sample the (tentative) degree from the interval $[n/5 - 50, n/5 + 50]$ using another `std::uniform_int_distribution` object. After this, for choosing the neighbours, let $N[1, 2, ...n]$ be the sampled degrees, we then use the following procedure to randomly sample from the set of "valid" vertices. (Note that the below procedure may increase the degrees for some vertices, however, even after this procedure, the degree of all vertices will be in $[n/5 - 50, n/5 + 50]$)

```
1  while ∃v ∋ deg(v) < N[v]
2      P ← RandomPermutation({1, 2, ..., n})
3      for i ← 1 to n
4          if P_i ≠ v and |N(P_i)| < n/5 + ε and (v, P_i) ∉ E
5              E ← E ∪ {(v, P_i)}
6          if deg(v) = N[v]
7              break
8      if deg(v) < N(v)
9          Discard G and try again
```

Note that, there is a possibility that no vertex is available to be connected with $v$. In this case, we discard the graph and try again. Since the random number generator's state is different, we have a chance of generating $G_2$ this time. However, this is a very rare case and was never observed in our experiments.

# 4 Results and Analysis

We use the `std::chrono` library for recording the running time of the algorithms. For each graph, we consider 5 different pairs of vertices and compute the running time of the algorithms for each pair. After running the program, we obtain the following results:

| **Graph** | **Algorithm** | **Running Times for 5 pairs (in $\mu s$)** | **Average Running Time (in $\mu s$)** |
|---|---|---|---|
| $G_1$ | Dijkstra's without Heap | 172486, 159878, 142704, 156475, 175314 | 161371.4 |
| $G_1$ | Dijkstra's using Heap | 5699, 5326, 4593, 5212, 5773 | 5320.6 |
| $G_1$ | Kruskal's | 13767, 14346, 14137, 14472, 14308 | 14206.0 |
| $G_2$ | Dijkstra's without Heap | 497463, 241440, 248683, 274556, 244420 | 301312.4 |
| $G_2$ | Dijkstra's using Heap | 88357, 89825, 91781, 95725, 91545 | 91446.6 |
| $G_2$ | Kruskal's | 5518049, 6021645, 6252663, 5755151, 5986980 | 5906897.6 |

From the average running times in the above table, we observe that the algorithms take much more time on $G_2$ than on $G_1$. The running times follow the trend that Dijkstra's algorithm using heap is the fastest, followed by Dijkstra's algorithm without heap and Kruskal's algorithm is the slowest. The difference between the running times of Dijkstra's algorithm using heap and Dijkstra's algorithm without heap can be explained using the time complexity table in the Methods section (i.e. $O(n^2)$ vs. $O(m \log n)$ where $m < \frac{n}{\log n}$). However, the difference between Dijkstra's using Heap and Kruskal's can only be attributed to constant factors and the BFS overhead.

# 5 Further Improvements

In practice, the `C++` standard library uses very smart memory allocations and placing strategies to avoid calling `realloc`, which can be costly. Our implementation only makes a best effort at avoiding these problems. In particular, we try to pre-reserve $n/5$ slots for each vertex in case of $G_2$. and try to pre-allocate `vector` sizes where it can be pre-determined. Hence, improving memory allocation would definitely improve our running time.

In the case of Kruskal's algorithm, finding the path from the source to the destination uses a BFS which internally uses a Queue. In our implementation, we use linked lists for the Queue. However, in practice, we can use a modified version of `vector` which again reduces the amount of OS calls needed for allocating memory. This would greatly improve the running time of Kruskal's algorithm.