CSCE 629-601 Analysis of Algorithms

Fall 2022

Instructor: Dr. Jianer Chen
Office: PETR 428
Phone: (979) 845-4259
Email: `chen@cse.tamu.edu`
Office Hours: MWF 3:50pm–5:00pm

Teaching Assistant: Vaibhav Bajaj
Office: EABC 107B
Phone: (979) 739-2707
Email: `vaibhavbajaj@tamu.edu`
Office Hours: T; 2pm-3pm, TR: 4pm-5pm

Course Notes 5. The Maximum Bandwidth Path

1 Definitions and problem formulation

We work on weighted graphs. The graphs can be either directed or undirected.

We interpret the weight of an edge $[u, v]$ as the *bandwidth* of the edge, which gives the capacity of the edge that allows the amount of flow to go through from vertex u to vertex v.

Let $P = \{v_0, v_1, \ldots, v_m\}$ be a path in a weighted graph G, where for each i, $[v_i, v_{i+1}]$ is an edge in G with bandwidth $bw(v_i, v_{i+1})$. The *bandwidth of the path* P is defined as

$$bw(P) = \min_{0 \le i \le m-1} \{bw(v_i, v_{i+1})\}.$$

We will work on the following problem:

> Maximum Bandwidth Path (Max-BW).
>
> Given a weighted graph G and two vertices s and t in G, construct a path from s to t such that the bandwidth of the path is the largest over all paths from s to t.

The Max-BW problem has many applications in computer networks, such as network communication, network flow analysis, and network reliability.

2 Dijkstra's algorithm

Our first algorithm for solving the Max-BW problem is based on Dijkstra's algorithm. Dijkstra's algorithm has been well-known for solving the Shortest Path problem. We show in this section that, with some minor changes, Dijkstra's algorithm can be used to solve the Max-BW problem.

Dijkstra's algorithm works based on greedy methods. At each stage, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. Greedy methods do not always yield globally optimal solutions, but they do for certain problems, in particular for a number of path problems (e.g., shortest path and maximum bandwidth path). If you use a greedy algorithm to find globally optimal solution for a problem, you in general need to give a formal proof to show that the solution constructed by your algorithm is indeed globally optimal.

Let us start with a review on Dijkstra's algorithm for path problems. Suppose we want to find a path from vertex s to vertex t. We start from the source vertex s, and grow a tree T by repeatedly adding the next (locally) best fringer to the tree T. The process stops when the sink vertex t is included in the tree T. At this point, we claim that the path in the constructed tree T from s to t is the best path. To implement this, suppose that the vertex set of the graph G is the integer set $\{1, 2 \ldots, n\}$. We use three arrays: array `status`$[1..n]$ to record the status of the vertices in the graph G (in-tree, fringer, or unseen), array `b-width`$[1..n]$ to record the bandwidth of the tree path from s to each vertex when the vertex becomes in-tree or fringer, and array `dad`$[1, , n]$ to record the parent of each vertex in the tree T.

The algorithm is given as follows.

```
Dijkstra-BW(G, s, t)
\\ construct a maximum bandwidth path from s to t in the graph G
1.  for(v = 1; v ≤ n; i++)
      status[v] = unseen;  b-width[v] = 0;  dad[v] = 0;
2.  status[s] = in-tree;  b-width[s] = +∞;  dad[s] = -1;
3.  for (each edge [s,w])
      status[w] = fringer;  b-widtht[w] = bw(s,w);  dad[w] = s;
4.  while (there are fringers)
4.1    pick the fringer v with the largest b-width value;  status[v] = in-tree;
4.2    for (each edge [v,w])
4.2.1     if (status[w] == unseen)
               status[w] = fringr;  dad[w] = v;
               b-width[w] = min{b-width[v], bw(v,w)};
4.2.2     else if (status[w] == fringer) & (b-width[w] < min{(b-width[v], bw(v,w)})
               dad[w] = v;  b-width[w] = min{(b-width[v], bw(v,w)};
5.  return the arrays dad[1..n] and b-width[1..n].
```

The proof for the correctness of the algorithm is very similar to that for the original Dijkstra's algorithm for the shortest path problem, which is based on the following lemma.

Lemma 1 *Once a vertex v becomes "in-tree", the path in the tree T from s to v is a maximum bandwidth path from s to v, whose bandwidth is given by b-width$[v]$.*

Proof. We prove the lemma by induction on the number k of vertices in the tree T:

When $k = 1$, the tree T has a single vertex s, whose bandwidth is $+\infty$ (see step 2). Therefore, the above claim holds true.

Now suppose that the claim holds true for $k \geq 1$, and we consider how the $(k+1)$-st vertex v is added to the tree T. By step 4.1, v is the fringer with the largest b-width value. After v becomes in-tree, the tree path P_{sv} from s to v consists of the tree path from s to the parent $u = \text{dad}[v]$ of v and the edge $[u, v]$. The bandwidth of the path P_{sv} is equal to b-width$[v]$.

We prove that the path P_{sv} is a maximum bandwidth path from s to v. Assuming the contrary that a maximum bandwidth path P'_{sv} from s to v has its bandwidth strictly larger than that of P_{sv}, i.e., $bw(P'_{sv}) > bw(P_{sv}) = \text{b-width}[v]$. Let

$$P'_{sv} = \{w_0, w_1, \ldots, w_h\},$$

where $w_0 = s$ and $w_h = v$. Let w_b be the first vertex in the path P'_{sv} that is not in the tree T (here we assume that the tree T contains k vertices while the vertex v is still a fringer to be added to the tree). Note that $b > 0$ so w_{b-1} is a vertex in the tree T so w_b is a fringer. By the inductive hypothesis, b-width$[w_{b-1}]$ is equal to the bandwidth of a maximum bandwidth path from s to w_{b-1}. Thus,

$$\text{b-width}[w_b] \geq \min\{\text{b-width}[w_{b-1}], bw(w_{b-1}, w_b)\} \geq bw(P'_{sv}) > bw(P_{sv}) = \text{b-width}[v].$$

The first inequality is because of steps 4.2.1-4.2.2 of the algorithm when the vertex w_{b-1} was been added to the tree T and the edge $[w_{b-1}, w_b]$ was examined, and the second inequality is because, by the inductive hypothesis, that b-width$[w_{b-1}]$ is not smaller than the bandwidth of the path $\{w_0, w_1, \ldots, w_{b-1}\}$ and that $\{w_0, w_1, \ldots, w_b\}$ is a partial path of P'_{sv}. However, this contradicts step 4.1 of the algorithm that should have picked the fringer with the largest b-width value – the vertex w_b is also a fringer and it has a b-width value larger than that of v.

This contradiction shows that the path P_{sv} must be a maximum bandwidth path from s to v. The inductive proof goes through, thus proving the lemma. □

By Lemma 1, for each vertex v (not only the sink t), once v becomes in-tree, by following the array dad$[1..n]$, we can get a maximum bandwidth path from s to v (in the reversed order).

We study the complexity of the algorithm `Dijkstra-BW`. Steps 1-3 take time $O(n)$. The `while`-loop of step 4 is executed at most $n-1$ times because each vertex can transition from fringer to in-tree at most once and once it becomes in-tree, it will never become a fringer again. Step 4.1 takes time $O(n)$ by linearly scanning the fringer list. Step 4.2 takes time $O(n)$ because each vertex has at most $n-1$ neighbors. In summary, the algorithm runs in time $O(n^2)$.

Step 4.2 of the algorithm can be analyzed more precisely. For each vertex v, step 4.2 takes time $O(deg(v))$, where $deg(v)$ is the degree of the vertex v (note that each execution of steps 4.2.1-4.2.2 takes time $O(1)$). Therefore, the total time spent on step 4.2 is actually bounded by

$$O\left(\sum_{v=1}^{n} deg(v)\right) = O(m),$$

where m is the number of edges in the graph G. Thus, indeed, the bottleneck is step 4.1, which takes time $O(n)$ for each fringer and can be executed $O(n)$ times.

We can improve the algorithm complexity by using a more efficient data structure to handle the fringers. For example, we can use a 2-3 tree to store the fringers. In this case, we can find the largest fringer in step 4.1 in time $O(\log n)$ instead of $O(n)$. Note that if we do this, then we also need the insertion operation when a new fringer is added and the deletion operation when a fringer is removed. We give this revision of the algorithm as follows, where F is the data structure (e.g., a 2-3 tree) for handling the fringers.

```
New-Dijkstra-BW(G, s, t)
\\ construct a maximum bandwidth path from s to t in the graph G
1.  for(v = 1; v ≤ n; i++)
       status[v] = unseen;  b-width[v] = 0;  dad[v] = 0;
2.  status[s] = in-tree;  b-width[s] = +∞;  dad[s] = -1;
    F = ∅;
3.  for (each edge [s,w])
       status[w] = fringer;  b-widtht[w] = bw(s,w);  dad[w] = s;
       Insert(F, w);
4.  while (there are fringers)
4.1     v = Max(F);  status[v] = in-tree;
        Delete(F, v);
4.2     for (each edge [v,w])
4.2.1     if (status[w] == unseen)
              status[w] = fringr;  dad[w] = v;
              b-width[w] = min{b-width[v], bw(v,w)};
              Insert(F, w);
4.2.2     else if (status[w] == fringer) & (b-width[w] < min{(b-width[v], bw(v,w)})
              Delete(F, w);
              dad[w] = v;  b-width[w] = min{(b-width[v], bw(v,w)};
              Insert(F, w);
5.  return the arrays dad[1..n] and b-width[1..n].
```

We analyze the new algorithm `New-Dijkstra-BW`. Steps 1-2 still take time $O(n)$, step 3, however, now takes time $O(n \log n)$ because of the insertion on `F`. Step 4.1 takes time $O(\log n)$

now because Max(F) and Delete(F, v) take time $O(\log n)$. As we discussed above, the total number of times steps 4.2.1-4.2.2 are executed in the entire execution of the algorithm is $O(m)$, while each execution of steps 4.2.1-4.2.2 takes time $O(\log n)$ because of the Insert(F, w) in step 4.2.1 and Delete(F, w) and Insert(F, w) in step 4.2.2. In conclusion, the total execution time of step 4 now becomes $O(n \log n + m \log n) = O((n + m) \log n)$ (where $n \log n$ is for the time of step 4.1), which is $O(m \log n)$ if we assume the graph G is connected (i.e., $m \geq n - 1$).

The algorithm `New-Dijkstra-BW` is not always better than the algorithm `Dijkstra-BW` because m can be as larger as $n(n-1)/2 \approx n^2/2$. On the other hand, we can "combine" the two algorithms to get one that guarantees to be not worse than both of them, i.e., an algorithm whose running time is $O(\min\{n^2, m \log n\})$.

The data structure F does not have to be a 2-3 tree. For example, we can use a max-heap that also supports Max, Insert, and Delete in $O(\log n)$ time per operation. The advantage of a max-heap over 2-3 trees is its simpler structure. Detailed implementation of a max-heap and its use in Dijkstra's algorithm for path problems are left to the students in their course project.

3 Kruskal's algorithm

Let G be a connected and weighted undirected graph. A *spanning tree* T of G is a subgraph of G that is a tree and contains all vertices of G. A *minimum spanning tree* (resp. *maximum spanning tree*) of G is a spanning tree of G whose weight is the smallest (resp. largest) over all spanning trees of the graph G.

Kruskal's algorithm has been famous for constructing a minimum spanning tree. With minor changes, it can be used to construct a maximum spanning tree. In this section, we study how the MAX-BW problem on undirected graphs can be solved based on a maximum spanning tree, then we use Kruskal's algorithm to construct a maximum spanning tree thus solve the MAX-BW problem.

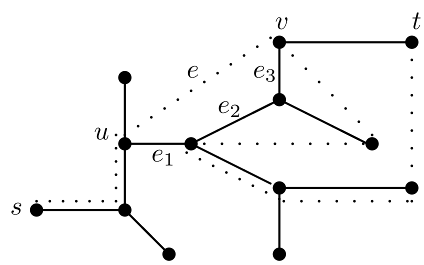
We start with the following lemma.

Lemma 2 *Let T be a maximum spanning tree of a graph G, and let s and t be any two vertices in G. Then the tree path from s to t in the tree T is a maximum bandwidth path from s to t.*

PROOF. Let $P_{\max}$ be a maximum bandwidth path from s to t in the graph G. We show that the bandwidth of the unique path P_{st} from s to t in the maximum spanning tree T is at least as large as that of $P_{\max}$.

If all edges in $P_{\max}$ are in T , then $P_{\max} = P_{st}$ and we have nothing to prove. Thus, assume that $e = [u, v]$ is the first edge on $P_{\max}$ that is not in the spanning tree T. Consider the unique path $P_{uv} = \{e_1, \ldots, e_r\}$ in the tree T from vertex u to vertex v (see Figure 1 for an illustration). Note that $P_{uv} \cup \{e\}$ forms a cycle.

We claim $bw(e) \leq \min\{bw(e_i) \mid 1 \leq i \leq r\}$. In fact, if $bw(e) > bw(e_i)$ for an edge e_i in the path P_{uv}, then $T' = T - \{e_i\} \cup \{e\}$ would form a spanning tree such that the sum of edge bandwidths of T' is larger than that of T, contradicting the assumption that T is a maximum spanning tree. Therefore, if we replace the edge e in $P_{\max}$ by the path P_{uv} in T, we get a path P' whose bandwidth is not smaller than that of $P_{\max}$. Moreover, the number of edges in P' that are not in T is 1 fewer than that in $P_{\max}$. Note that the resulting path P' may not be "simple", i.e., some nodes may repeat on the path P', but we can easily remove the segments between two appearances of the same vertex, without decreasing the bandwidth of the path. In any case, we will get a simple path from s to t, in which the number of edges not in T is 1 fewer than that in $P_{\max}$, and whose bandwidth is not smaller than that of $P_{\max}$. Repeating the above process will eventually give us a simple path entirely in T from s and t, whose bandwidth is not smaller than that of $P_{\max}$. Since there is a unique such path P'' in the tree T and since the bandwidth of the path P'' is not smaller than that of the maximum bandwidth path $P_{\max}$, this path P'' from s to t in the maximum spanning tree T must be a maximum bandwidth path from s to t in the graph G. The lemma is proved □



(1) $P_{\max}$: the dashed lines, (2) T: the solid lines.

Figure 1: The maximum spanning tree and the maximum bandwidth path

By Lemma 2, to construct a maximum bandwidth path from s to t, we can first construct a maximum spanning tree T, then find the unique tree path in T from s to t. Note that finding the unique tree path from s to t in the tree T can be done in time $O(n)$, using either DFS or BFS. Compared to Dijkstra's algorithm for the MAX-BW problem, the advantage of this approach is that once the maximum spanning tree T is constructed, you can find the maximum bandwidth path from *any* source vertex s to *any* sink vertex t in time $O(n)$, using DFS or BSF. On the other hand, Dijkstra's algorithm `Dijkstra-BW`, as given in the previous section, can only find maximum bandwidth paths from a fixed source vertex s (to all other vertices).

What that is left is to construct a maximum spanning tree. We use the famous Kruskal's algorithm, as given as follows, where we assume that the input graph G is connected.

```
Kruskal-MST(G)
\\ construct a maximum spanning tree for the graph G
1.  sort the edges of G in non-increasing order by their edge weights bw(e_i):  e_1, e_2, ..., e_m;
2.  T = the vertices of G (without any edges);
3.  for (i = 1; i ≤ m; i++)
3.1    let e_i = [u_i, v_i];
3.2    if (u_i and v_i are in different pieces of T)
           add e_i to T (that connects the two pieces);
4.  return (T).
```

We prove the correctness of Kruskal's algorithm, starting with the following lemma.

Lemma 3 *At any time in the execution of the algorithm* `Kruskal-MST`(G), *all the edges in the set T are entirely contained in a maximum spanning tree of the graph G.*

PROOF. We prove the lemma by induction on the edge index i for the edge e_i in step 3.1.

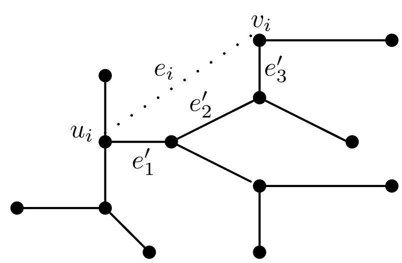
For $i = 0$, the set T contains no edges (see step 2), so the lemma holds true.

Now suppose that the lemma holds true for all $0 \leq h < i$, and we consider the i-th edge e_i processed by step 3.1. By induction, after processing the first $i-1$ edges e_1, e_2, $\ldots$, e_{i-1}, all the edges in the set T are entirely contained in a maximum spanning tree $T_{\max}$.

If the two endpoints u_i and v_i of the edge e_i are in the same piece in T, then by step 3.2, the edge e_i is not added to T so T is unchanged. Thus, after processing the edge e_i, we still have all edges in T contained in the maximum spanning tree $T_{\max}$.

If the two endpoints u_i and v_i of the edge e_i are in different pieces in T but the edge e_i is in the maximum spanning tree $T_{\max}$, then the algorithm adds the edge e_i to T. In this case, we still have all edges in T contained in the maximum spanning tree $T_{\max}$.

The remaining case is that the two endpoints u_i and v_i of the edge e_i are in different pieces in T but the edge e_i is not in the maximum spanning tree $T_{\max}$. Then the graph $T_{\max} \cup \{e_i\}$ contains a cycle $C = \{e'_1, \ldots, e'_r, e_i\}$, where $e'_1, \ldots, e'_r$ are edges in $T_{\max}$. See Figure 2.



The maximum spanning tree $T_{\max}$: the solid lines.

Figure 2: The correctness of Kruskal's algorithm

Since u_i and v_i are not in the same piece in T, there must be an edge e'_h in $\{e'_1, \ldots, e'_r\}$ such that the two endpoints of e'_h are in two different pieces in T. We can easily verify:

(1) the edge e'_h has not been processed by step 3, yet: otherwise, it would have been added to T in step 3.2 and the two endpoints of e'_h would have been in the same piece in T. As a result, by step 1, we have $\text{bw}(e'_h) \leq \text{bw}(e_i)$.

(2) the weight $\text{bw}(e'_h)$ of the edge e'_h cannot be smaller than that $\text{bw}(e_i)$ of the edge e_i: otherwise, the tree $T' = T_{\max} - \{e'_h\} \cup \{e_i\}$ would be a spanning tree of G whose weight is larger than the maximum spanning tree $T_{\max}$.

Thus, we must have $\text{bw}(e'_h) = \text{bw}(e_i)$, so $T' = T_{\max} - \{e'_h\} \cup \{e_i\}$ is also a maximum spanning tree of G that contains all edges in T (note that the edge e'_h is not in T).

This completes the proof of the lemma. □

To prove that the output T of the algorithm `Kruskal-MST`(G) is a maximum spanning tree of the graph G, we still need to verify that T is a connected graph (note that by step 2, the set T contains all vertices of G). But this is easy: suppose that T is not connected. Since the graph G is connected, there must be an edge e_i in G that connects two different pieces in T, but this is impossible: when the edge e_i is processed in step 3.2, the algorithm would have added the edge e_i to T so that the two endpoints of e_i cannot be in two different pieces in the final set T. This contradiction shows that the output T of the algorithm `Kruskal-MST`(G) must be a connected graph. This, plus the fact that T contains all vertices of the graph G and that all edges in T are contained in a maximum spanning tree $T_{\max}$, concludes that T by itself is the maximum spanning tree $T_{\max}$ of the graph G.

We study the complexity of the algorithm `Kruskal-MST`(G). Step 1 of the algorithm takes time $O(m \log m) = O(m \log n)$ by any optimal sorting algorithm such as `MergeSort`.

To handle the dynamic changes of the set T, we use three functions: `MakeSet`(w), `Find`(w), and `Union`(p_1, p_2), where `MakeSet`(w) creates a set consisting of a single vertex w (thus, used in step 2 of the algorithm), `Find`(w) finds the piece of T that contains the vertex w (thus, used in step 3.2 to check if the two endpoints u_i and v_i of the edge e_i are in the same piece), and `Union`(p_1, p_2) merges two pieces p_1 and p_2 into a single piece (thus used in step 3.2 when we add the edge e_i to T to connect two pieces in T).

There has been extensive study on the complexity of the functions `MakeSet`, `Find`, and `Union`. We will study this in details in our class. For their use for the algorithm `Kruskal-MST`, it suffices to know that each of these operations takes time $O(\log n)$. Bringing this fact into the algorithm, we can easily conclude that the algorithm `Kruskal-MST` runs in time $O(m \log n)$ (note that since G is connected, $m \geq n - 1$).

We close this section by the following algorithm that solves the Max-BW problem for weighted undirected graphs using Kruskal's algorithm. By the above discussions, the algorithm `Kruskal-BW`(G, s, t) runs in time $O(m \log n)$ and correctly constructs a maximum bandwidth path from s to t in the graph G.

```
Kruskal-BW(G, s, t)
\\ construct a maximum bandwidth path from s to t in the graph G
1.  T = Kruskal-MST(G);
2.  Use DFS or BFS to construct the path P in T from s to t;
3.  return(P).
```