CS-E3190 Principles of Algorithmic Techniques

04. Local search – Tutorial Exercise

1. **Borůvka's algorithm.** The goal of this greedy algorithm is to compute a minimum spanning tree for a weighted undirected graph $G = (V, E)$. Let $w : E \to \mathbb{R}$ be the weight function on the edges of G and suppose that the weights are unique.

   To be able to easily describe the algorithm, we define the concepts of *safe edge* and *useless edge*.

   Let G be the input graph and F be its subgraph. Let $e = uv$ be an edge of $G \setminus F$ and C be a connected component of F.

   - The edge e is *useless* if u and v are both in C.
   - The edge e is *safe* if it is the minimum weight edge with exactly one endpoint in C.

   Edges that are neither safe nor useless are undefined, and can become safe or useless once more edges are added to F. If F is connected, all edges are useless.

   The idea of Borůvka's algorithm is to add all the safe edges to F at each iteration, until F is connected. We initialize the tree as $F = (V, \emptyset)$, ie. there are no edges and each vertex forms its own connected component.

   The algorithm is presented as pseudocode in Algorithm 1. To detect the safe edges we use the subroutine in Algorithm 2, and Algorithm 4 is used to detect and add the safe edges to F.

Algorithm 1: Borůvka's algorithm

```
input : G = (V, E)
F = (V, ∅)
#CC ← CountAndLabel(F)
while #CC > 1 do
    AddAllSafeEdges(E, F, #CC)
    #CC ← CountAndLabel(F)
end
return F
```

Algorithm 2: CountAndLabel(G)

```
count ← 0
for v ∈ V do
    Unmark v.
end
for v ∈ V do
    if v is unmarked then
        // Ensure a different label for
           each component.
        count ← count + 1
        LabelOne(v, count)
    end
end

return count
```

Algorithm 3: LabelOne(v,$count$)

```
// Goal:  label nodes in the component
   of v with count.
C ← {v}
while C ≠ ∅ do
    Take u from C.
    if u is unmarked then
        Mark u.
        comp(u) ← count
        // Store component for later
        for all w ∈ N(u) ∩ F do
            C ← C ∪ {w}
        end
    end
end
```

Algorithm 4: AddAllSafeEdges

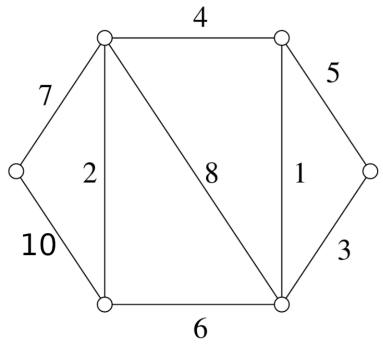
```
for i ∈ [count] do
    safe[i] ← NULL
end
// Find a safe edge for each component.
for every edge uv ∈ E do
    if comp(u) ≠ comp(v) then
        if safe[comp(u)] = NULL or w(uv) < w(safe[comp(u)]) then
            safe[comp(u)] ← uv
        end
        if safe[comp(v)] = NULL or w(uv) < w(safe[comp(v)]) then
            safe[comp(v)] ← uv
        end
    end
end
for i ∈ [count] do
    add safe[i] to F
end
```

To prove the correctness of the algorithm we need to prove the following lemma.

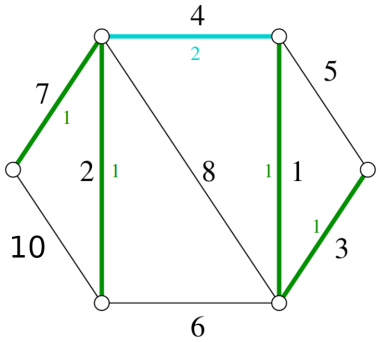
Lemma 1. The minimum spanning tree contains every safe edge.

Because the algorithm only adds safe edges to F, in the end of the algorithm, F must be equal to the minimum spanning tree of G.

(a) Apply the algorithm to the following graph:

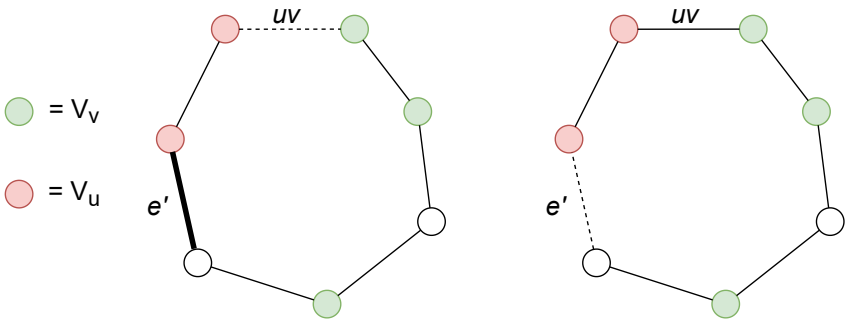


Solution. The coloured numbers are the phase numbers the edges were picked in. In this example, there are only $2 = \lfloor \log_2 6 \rfloor$ phases.



(b) Prove lemma 1. **Solution.** Suppose that in some iteration, a safe edge uv is added to F but uv does not belong to the MST. Let V_u denote the vertices in the same component with u in that iteration, and suppose uv was the minimum edge for that component.

Because the MST is connected, some other path P connects the vertices u and v in it. Not all the vertices in the path can be in V_u, as otherwise u and v would have belonged to the same connected component when uv was added. Hence there is at least one edge $e' \in P$ with only one endpoint in V_u. We have $w(e') > w(uv)$, because uv was the minimum edge of the component V_u. Adding uv to the MST creates a cycle in the MST, and removing any edge from that cycle gives us a spanning tree for G. Hence removing e' from the MST and adding uv in it would create a new spanning tree with smaller total weight, which contradicts the minimality of the MST.



(c) Using Lemma 1, conclude that the algorithm is correct. **Solution.** We showed that all the edges that we add, belong to the MST, ie. $F \subseteq$ MST throughout the algorithm. Because we continue the algorithm until F is connected, it must be equal to the MST, as trees are *minimally connected* (removing any edge would make the graph disconnected).

We also need to argue why the algorithm always terminates: as long as F is disconnected, there exists at least one edge between some two components (as the input graph is connected). Hence there must also be a minimum one, and in each iteration we end up adding at least one edge. Hence the graph will eventually become connected.

(d) Prove that this algorithm runs in $O(m \cdot \log n)$ time.

Solution. Notice, that adding an edge to F reduces the number of connected components by one. In each iteration, the number of connected components at least halves: in the worst case, each added edge is the safe edge for two of the components. There are initially n components so we need $O(\log n)$ iterations until there is only one component left. In other words, the while-loop runs $O(\log n)$ times.

Inside the while-loop, we call two functions: AddAllSafeEdges and CountAndLabel. Let's analyse their time complexity. We assume that the component of each vertex is stored somewhere so that it can be accessed in $O(1)$ time.

AddAllSafeEdges

1. Initializes an array for the safe edges. This is done for each connected component, and hence takes at most $O(n)$ time.
2. Iterates over all edges. Inside the for-loop, we check three if conditions, each in $O(1)$ time. The for-loop takes $O(m)$ time.
3. We add one edge for each connected component. This takes $O(n)$ time.

CountAndLabel

1. We initialize all vertices to unmarked in $O(n)$ time.
2. We iterate over the vertices. Inside the for-loop we call the function LabelOne for all unmarked nodes. The function LableOne marks all the nodes in the same component with v, and hence it is called once for each connected component. Inside LabelOne we end up iterating over all the edges and vertices in the component. As we do this for each component, CountAndLabel takes $O(n+m)$ time. As we call CountAndLabel for the forest F, we have $m < n$, and the runtime simplifies to $O(n)$.

This means, that the runtime of the insides of the while-loop is dominated by the for-loop iterating over the edges, and it runs in $O(m)$ time. The runtime of the whole algorithm is hence $O(m \log n)$.

(e) Show that Borůvka's algorithm does not necessarily work if the weights are not unique. **Solution.** Let the input graph G be the triangle graph K_3 and all edges have weight one. In the first iteration, it is possible that all the three components choose different edge, and adding the three edges creates a cycle in F.