Modern Algorithm Design (Monsoon 2024)
Homework 1

Deadline: 2nd September, 2024, 11:59 pm (IST)

Release Date: 19th August, 2024

1. (a) For a graph G, consider two edge-weight functions w_1 and w_2 such that

$$w_1(e) \leq w_1(e') \iff w_2(e) \leq w_2(e')$$

for all edges $e, e' \in E$. Show that T is an MST wrt w_1 iff it is an MST wrt w_2. (In other words, only the sorted order of the edges matters for the MST.)

(b) Suppose graph G has integer weights in the range $\{1, \cdots W\}$, where $W \geq 2$. Let G_i be the edges of weight at most i, and κ_i be the number of components in G_i . Then show that the MST in G has weight exactly $n - W + \sum_{i=1}^{W-1} \kappa_i$.

(c) Show that if the edge weights are all distinct, there is a unique MST.

Proof. (a) Recall the cut property of a graph regarding its MST. It says that for any arbitrary cut of the graph, The minimum weight edge crossing the cut has to be in the MST.

Using this property, we argue that if T is an MST with respect to w_1, then it is an MST with respect to w_2. The proof for the opposite direction holds the same argument. Take any arbitrary cut S, S' of the graph. Let $e \in T$ be the edge crossing the cut. Then e must be the minimum edge (with respect to w_1) crossing the cut. Observe that, for the same cut, e is one of the minimum edges (with respect to w_2) crossing the cut. This follows from the given relation between w_1 and w_2. This implies that for any cut, the crossing edge selected in T is minimum with respect to both w_1 and w_2. Thus, T is an MST for w_2.

(b) We have the following.

Fact 1. Let G be a graph having multiple components. If we add a new edge to G, then

- The number of components reduces if and only if e does not create a new cycle in G.
- If the number of components reduces, it reduces by exactly one.

The above fact implies that $\kappa_{i-1} \geq \kappa_i$; $\forall i \in \{1, \cdots, W\}$ (G_0 contains no edge, thus $\kappa_0 = n$). Moreover, $\kappa_{i-1} - \kappa_i$ reflects the number of edges of weight i, which does not introduce any new cycle in G_i compared to G_{i-1}. For all $i \in \{1, \cdots, W\}$, let $\varepsilon_i \in G_i \setminus G_{i-1}$ be the set of edges which does not introduce any cycle in G_i compared to G_{i-1}. More precisely, each edge $e \in \varepsilon_i$ is accountable for reducing exactly one component of G_{i-1}. Observe that, $\kappa_{i-1} - \kappa_i = |\varepsilon_i|$ and $\varepsilon_i \cap \varepsilon_j = \emptyset$ for all $1 \leq i < j \leq W$. Let $T = \cup_{i=1}^{W} \varepsilon_i$. By construction, T does not contain any cycle. Moreover, $|\cup_{i=1}^{W} \varepsilon_i| = \sum_{i=1}^{W} (\kappa_{i-1} - \kappa_i) = n - 1$. Thus, T is a spanning tree. We argue that T is a minimum spanning tree of G. For contradiction, let us assume that T is not an MST. Then, there must be edges in MST, say T^*, which are not in T. Let e be such an edge with the smallest possible weight i. By our construction, T does not contain e because e must introduce a cycle in G_i. Then e is the heaviest edge of that cycle. This contradicts the fact that e is part of T^*.

Now, as we established that T is MST, we compute the cost of T as

$$w(T) = \sum_{i=1}^{W} i(\kappa_{i-1} - \kappa_i) = (1.\kappa_0 + \cdots + W.\kappa_{W-1}) - (1.\kappa_1 + \cdots + W.\kappa_W)$$

From the fact that $\kappa_0 = n$ and $\kappa_W = 1$ along with the above equation, we get,

$$w(T) = n - W + \sum_{i=1}^{W-1} \{(i+1) - i\}\kappa_i = n - W + \sum_{i=1}^{W-1} \kappa_i$$

(c) For the sake of contradiction, let the graph have at least two different MSTs, say T and T'. Further, let T be the MST generated using Kruskal's algorithm while T' is an arbitrary spanning tree. Suppose $e = (xy)$ is the first edge in the sorted list of Kruskal such that e is added to T while it is not part of T' (note that there has to be such an edge since we have assumed T, T' to be different.) Clearly, x and y are part of disjoint connected components in the Kruskal forest built so far. Further $e \cup T'$ contains a cycle C which includes both x and y. Consider the heaviest edge $e^\star$ in C among the edges that are not part of the Kruskal forest till this point. But then this is also the heaviest edge in C because of the sorted order in which Kruskal considers edges. This contradicts the minimality of T' through violation of Cycle Rule.

□

2. (a) Show how to implement the "contract" subroutine in $O(m)$ time. This algorithm takes as input a graph $G = (V, E)$ with some edges colored blue, and outputs a new graph $G' = (V', E')$ in which vertices $v_C \in V'$ correspond to blue connected components C in G, there are no self-loops, and there is a (single) edge $e_C = (v_C, v_{C'})$ if there exists some edge between the corresponding components C, C' in G, and the weight of edge $(v_C, v_{C'})$ is given by $\min_{x,y \in E : x \in C, y \in C'} w_{xy}$.

(b) We proved in class that Boruvka reduces the number of nodes by a constant factor in each round, but what about the number of edges? Show an example of a graph with n nodes and m edges where the number of edges in G_i remains $\Omega(m)$ for $\Omega(\log n)$ rounds, even after cleaning up.

(c) Design an $O(m \log \log n)$-time algorithm to find MST using algorithms/data-structures you have seen in the lectures.

Proof. (a) Consider C_i be the set of connected components at iteration i of Bruvka. The "contract" subroutine does the following two things.

- Remove all the edges (u, v) form G such that both u and v belongs to same component.
- For each component $c \in C_i$, identify the minimum weighted edge (u, v) such that $u \in c$ and $v \notin c$.

Let MIN_{C_i} be an array of size $|C_i|$, which stores the minimum weighted edge for each component in C_i. For any component $c \in C_i$, $MIN_{C_i}(c)$ gives the edge with weight, and $w(MIN_{C_i}(c))$ denotes the weight of that edge. Initially, we set $MIN_{C_i}(c) = NIL$ for all $c \in C_i$ and weight of $w(NIL) = \infty$. We provide a $O(m)$ "contract" subroutine.

Algorithm 1: contract(C_i)

```
for the components in C_i do
    number the components 1, ··· , |C_i|
end
for each component c ∈ C_i do
    for each vertex v ∈ c do
        associate the component number of c with v
    end
end
for each vertex v ∈ V do
    c ← the component containing u
    for each edge (u, v) adjacent to v do
        if u has component number of c, that is u ∈ c then
            remove (u, v) from the edge list of v
        else
            if w(MIN_{C_i}(c)) > w_uv then
                MIN_{C_i}(c) ← uv
            end
        end
    end
end
```

(b) Our example graph is a complete graph with n vertices denoted by K_n. We provide the weight assignments on edges in a constructive manner as follows: on i-th iteration (where $1 \le i \le \log n$) of Boruvka;

- If n_i is the number of vertices, choose $\frac{n_i}{2}$ pairs of vertices arbitrarily. For any pair, (u, v), assign weight i to the edge uv and color uv blue.

- Contract the colored edges.
- For any two different components C and C', arbitrarily keep one edge between C and C' and remove all the other edges between them. Assign weight ∞ to all the deleted edges in this phase.

Note that in each phase of Boruvka, the contracted graph is a complete graph; that is, in phase i, the graph is K_{n_i}. Now, we show how this example is valid in the problem context. The number of edges at the beginning of Boruvka is $\Omega(n^2)$. However, after $\log(\sqrt[100]{n}) = \frac{\log n}{100}$ number of steps, the number of vertices is $\frac{n}{\sqrt[100]{n}}$, which is $n^{.99}$. Then the number of edges is $\Omega(n^{1.98}) \approx \Omega(n^2)$.

(c) Following is an $O(m \log\log n)$ algorithm for MST.

i. Run Boruvka's algorithm for $(\log\log n)$ iterations. After that, the number of vertices in the contracted graph is at most $\frac{n}{\log n}$. The runtime of this step is $O(m \log\log n)$.

ii. On the contracted graph, use Prim's/Kruskal's algorithm with fibonacci heaps, which has a runtime of $O(m + \frac{n}{\log n} \log n) = O(m + n)$.

□

3. We will design yet another $O(m \log\log n)$-time MST algorithm, but without using any fancy data-structures: *in Boruvka's algorithm we scan all the edges in the graph in each pass, and we should avoid this repetition.* Assume that G is a connected simple graph, and edge weights are distinct.

(a) Suppose for each vertex, the edges adjacent to that vertex are stored in increasing order of weights. Show a slight variant of Boruvka's algorithm with runtime $O(m + n \log n)$.

(b) (*k-partial sorting*) Given a parameter k and a list of N numbers, give an $O(N \log k)$-time algorithm that partitions this list into k groups $g_1, g_2, \cdots, g_k$ each of size at most $\lceil N/k \rceil$, so that all elements in g_i are smaller than those in g_{i+1}, for each i.

(c) Adapt your algorithm from part (a) to handle the case where the edges adjacent to each vertex are not completely sorted but only k-partially-sorted. Ideally, your run-time should be $O(m + \frac{m}{k} \log n + n \log n)$.

(d) Use the two parts above (setting $k = \log n$), preceded by some additional rounds of Boruvka, to give an $O(m \log\log n)$-time MST algorithm.

Proof. (a) Following is the modification over Boruvka's algorithm.

At i-th iteration of Boruvka, where $i \in \{1, \cdots, \log n\}$, do the following:

i. Let n_i denotes the vertices in i-th round. Recall that each vertex of n_i represents a connected component constructed during $(i-1)$-phases of Boruvka. Do the following:
For each component $c \in n_i$;

A. For each vertex $v \in c$, traverse the adjacent edges in sorted order and delete them until we get an edge e such that $e \notin c$. In other words, we keep removing edges in the order until we get an edge e, shared by two different components of n_i.

B. Observe that after step A, the first edge (say denoted by e_v^*) of any vertex $v \in c$ is shared by two different components of n_i. Now we take the edge $e = \arg\min_{e_v^*: v \in c} w(e_v^*)$ and color it blue.

ii. Contract the blue edges.

To analyse the algorithm's running time, one may observe that any edge either gets deleted or remains as e_v^* for some v for few consecutive Borukva phases. However, when we visit a vertex v, we repeatedly delete edges or see only the first edge, e_v^*. This repeated deletion of edges overall costs $O(m)$ through the execution of Borukva. Rest, in each phase, we spend $O(n)$ time to identify the edges to be colored blue and then contracting the edges. Thus, total running time becomes $O(m + n \log n)$.

(b) For sake of simplicity, we consider that k to be some power of 2. Consider Arr be the input array. Following is the algorithm which we initialize by k-partial sorting(Arr):

Algorithm 2: k-partial sorting(A)

```
if |A| > ⌈N/k⌉ then
    m ← median of A using the median-of-median procedure
    Run one pass of quick-sort, taking m as the pivot
    L_A ← left half of A (elements from start of A to m)
    R_A ← right half of A (elements just after m to last of A)
    k-partial sorting(L_A)
    k-partial sorting(R_A)
end
```

(c) For any vertex v, if the list of adjacent edges is k-partially sorted, then the minimum weight edge must contained in the first group. Since each group is of size $\frac{deg(v)}{k}$, we can find the minimum weight edge in $\frac{deg(v)}{k}$ time, where $deg(v)$ is the degree of v. Finding the minimum for each vertex takes $\frac{\sum_v deg(v)}{k}$, which is upper bounded by $\frac{2m}{k}$. So, we take the following steps in the i-th phase of Boruvka:

i. For each vertex v, we visit the first group in ordering edges, and in that group, we look for edges shared between different components. If no such edge exists, remove all edges from the current group, move to the next group and repeat the same.

ii. If such edge (lies between different components) exists in the current group, then mark the minimum weighted edge among all such edges as e_v^*.

iii. For any component c color the edge $e = \arg\min_{e_v^*: v \in c} w(e_v^*)$ with blue.

iv. Now, do contraction.

Note that The running time of each phase of Boruvka is bounded by $O(\frac{m}{k} + n)$. Again, since we delete an edge at most once, the overall cost of deletion of edges is bounded by $O(m)$. So, the total running time becomes $O(m + \frac{m}{k} \log n + n \log n)$.

(d) Following is the algorithm:

i. Run vanilla Borukva for $(\log \log n)$ time.

ii. On the remaining graph of at most $\frac{n}{\log n}$ vertices, do k-partial sorting.

iii. Run the algorithm discussed in (c).

The running time of step (i) is $O(m \log \log n)$. The running time of step (ii) is $O(\frac{n}{\log n} \log \log n) \leq O(n \log \log n)$. The running time of step (iii) is $O(m + m \log \log n + n)$. Thus, total running time is $O(m \log \log n)$

□

4. Recall that Dijkstra's algorithm computes the single-source shortest-path (SSSP) correctly for directed graphs with non-negative edge-lengths. For graphs with negative-length edges, we use typically the Bellman-Ford or Floyd-Warshall algorithms. Let us explore what happens if we use Dijkstra's algorithm instead. Assume that the graph does not have negative-length cycles.

(a) Show an example of a graph with negative edge-lengths where Dijkstra's algorithm returns the wrong shortest-path distance from the source s. For your reference, we give Dijkstra's algorithm in algorithm 3.

Algorithm 3: Dijkstra's Algorithm

```
D(s) = 0, D(v) = ∞ for all v ≠ s
run Dijkstra-Iteration
```

Algorithm 4: Dijkstra-Iteration

```
unmark all nodes
while not all vertices marked do
    u ← unmarked vertex with least label D(u)
    mark u
    for all out-edges (u,v) of u do
        D(u) ← min{D(v), D(u) + l(u,v)}
    end
end
```

(b) Now suppose we iterate through Dijkstra's algorithm K times (shown formally as under). Consider any node v such that the shortest-path from s to node v contains at most $K - 1$ negative-length edges. Show that the final value of the label $D(v)$ equals the length of this shortest s-v path.

Algorithm 5: K-Fold Dijkstra's Algorithm

```
D(s) = 0, D(v) = ∞ for all v ≠ s
for i = 1, 2, ··· , K do
    run Dijkstra-Iteration
end
```

Proof. (a) For the example, we consider the following directed graph $G = (V, E)$ with three vertices.

- $V = \{A, B, C\}$
- $E = \{\overrightarrow{AB}, \overrightarrow{BC}, \overrightarrow{AC}\}$
- $w(\overrightarrow{AB}) = 2, w(\overrightarrow{BC}) = -2, w(\overrightarrow{AC}) = 1$

In the first iteration of Dijkstra, vertex C relaxes because the distance from A to B is 2 and the distance from A to C is 1. In the next iteration, vertex B relaxes. However, the shortest path from A to C is $A \to B \to C$, which costs 0.

(b) We prove it by induction on K. For each $i \in \{1, \cdots, K\}$, we claim that, after running Dijkstra i-many rounds, the K-Fold Dijkstra's algorithm correctly computes the shortest path from s to all those vertices, whose shortest path consists of at most $(i-1)$ negative weight edges.

Base case: Consider the case when $i = 1$. Running Dijkstra single time reports the shortest path to those vertices whose shortest path has no negative edge.

Induction hypothesis: Consider that the claim is true for all the vertices at sep $(i-1)$ whose shortest path consists of at most $(i-2)$ negative weight edges.

Induction Step: Now we argue about the i-th step. In the i-th step, let v be a vertex having at most $(i-1)$ negative edges in its shortest path. There are two cases. If v's shortest path consists of less than $(i-1)$ negative edges, then by induction hypothesis, we have already computed the shortest path to v, and thus, we can not relax the vertex further.

For the other case, v consists of exactly $(i-1)$ negative weight edges in its shortest path. Let $\overrightarrow{xy}$ be the last edge with negative weight in this shortest path. By the property of shortest path, the shortest path of y contains x. As the shortest path to x consists of exactly $(i-2)$ negative edges, we have already computed the shortest path to x by the induction hypothesis. Running Dijkstra in the i-th term would relax y with the negative edge weight of xy. Thus, Dijkstra computes the shortest path to y in the i-th phase. Furthermore, the shortest path from y to v has no negative edge. So, Dijkstra would also update the shortest path for all the vertices contained in the path from y to v. Hence, Dijkstra computes the shortest path from s to v.

□