Combinatorial optimization for undergraduate students

Lecture note 6. Spanning trees-Prim and Kruskal's algorithms

Lecturer : O-joung Kwon
Spring, 2018

A *tree* is a connected graph having no cycles. A *forest* is the disjoint union of trees, that is, a graph having no cycles.

A tree or a forest in a graph G is called *spanning* if it contains all vertices of G.

We consider spanning trees in networks (G, w). For any subset T of the edge set of G, we define the weight of T by $w(T) := \sum_{e \in T} w(e)$. A spanning tree is called *minimum* if its weight is minimum among all the weights of spanning trees.

We assume that the given graph is connected, in the chapter.

If the weight function is constant, then BFS gives a tree which is a solution.

Lemma 1. *Let G be a graph. Then the following conditions are equivalent.*

(1) G is a tree.
(2) G does not contain any cycles, but adding any further edge yields a unique cycle.
(3) Any two vertices of G are connected by a unique path.
(4) G is connected and removing every edge disconnects the graph.

Consider a spanning tree T in a graph G. For an edge e not in T, there is a unique cycle formed by T and e. We denote this cycle by $C_T(e)$.

Theorem 1. *Let (G, w) be a network, where G is a connected graph. A spanning tree T of G is minimum if and only if the following condition holds for each edge e in $G - T$:*

- *$w(e) \geq w(f)$ for every edge f in $C_T(e)$.*

Another characterization of minimum spanning trees.

Let G be a connected graph with $V(G) = \{1, 2, \ldots, n\}$ and $w : E(G) \to \mathbb{R}$.

Algorithm 1 Prim's general method

1: **procedure MINTREE**$(G, w; T)$
2: for $i = 1$ to n do $V_i \leftarrow \{i\}$; $T_i \leftarrow \emptyset$;
3: for $k = 1$ to $n - 1$ do
4: choose V_i with $V_i \neq \emptyset$;
5: choose an edge $e = uv$ with $u \in V_i$, $v \notin V_i$, and $w(e) \leq w(e')$ for all edges $e' = u'v'$ with $u' \in V_i$ and $v' \notin V_i$;
6: determine the index j where $v \in V_j$;
7: $V_i \leftarrow V_i \cup V_j$; $V_j \leftarrow \emptyset$;
8: $T_i \leftarrow T_i \cup T_j \cup \{e\}$; $T_j \leftarrow \emptyset$;
9: if $k = n - 1$ then $T \leftarrow T_i$;
10: **end procedure**

We will give two algorithms depending on how to proceed lines 4 and 5.

Theorem 2. *Algorithm 1 determines a minimum spanning tree for the network* (G, w).

Algorithm 2 Prim's algorithm

```
1: procedure PRIM(G, w; T)
2:     g(1) ← 0, S ← ∅, T ← ∅;
3:     for i = 2 to n do g(i) ← ∞;
4:     while S ≠ V(G) do
5:         choose i ∈ V(G) \ S such that g(i) is minimum; S ← S ∪ {i};
6:         if i ≠ 1, then T ← T ∪ {e(i)},
7:         for j ∈ A_i ∩ (V(G) \ S) do
8:             if g(j) > w(ij), then g(j) ← w(ij); e(j) ← ij
9: end procedure
```

Theorem 3. *Prim's algorithm determines with complexity $\mathcal{O}(|V(G)|^2)$ a minimum spanning tree T for the network (G, w).*

We assume that the edges of G are ordered according to their weight, that is $E = \{e_1, \ldots, e_m\}$ with $w(e_1) \leq \cdots \leq w(e_m)$.

Algorithm 3 Kruskal's algorithm-preversion

```
1: procedure Kruskal(G, w; T)
2:     T ← ∅;
3:     for k = 1 to m do
4:         if e_k does not form a cycle together with some edges of T
5:         then append e_k to T;
6: end procedure
```

Note that Algorithm 3 is the special case of MINTREE, and Theorem 1 shows that it returns a minimum spanning tree.

In order to arrange the edges according to their weight and to remove the edge of smallest weight, we use the data structure *priority queue*, say Q. DELETEMIN(Q) delete an element which has minimum weight in Q, and it can be performed in $\mathcal{O}(|Q| \log |Q|)$ steps.

How do we determine whether the step 4 creates a cycle or not?

To be precise, we need the operation called MERGE(H, H') which takes disjoint union of two graphs.

Algorithm 4 Kruskal's algorithm

```
1: procedure Kruskal(G, w; T)
2:     T ← ∅;
3:     for i = 1 to n do V_i ← {i};
4:     put E(G) into a priority queue Q with priority function w;
5:     while Q ≠ ∅ do
6:         e := DELETEMIN(Q);
7:         find the end vertices u and v of e;
8:         find the components V_u and V_v containing u and v, respectively;
9:         if V_u ≠ V_v then MERGE(V_u, V_v); T ← T ∪ {e};
10: end procedure
```

Theorem 4. *Kruskal's algorithm determines with complexity* $\mathcal{O}(|E(G)|\log|E(G)|)$ *a minimum spanning tree of* (G, w).

Maximal spanning trees: for some problems, it is necessary to consider *maximal spanning trees*. Obviously, a spanning tree T for (G, w) is maximal iff it is minimum for $(G, -w)$, and we can use one of previous algorithms. Instead, we could also stay T and modify the algorithms: for instance, in Kruskal's algorithm, we order the edges according to decreasing weight.

(Example) For each edge ij, p_{ij} denotes the probability that information sent is overheard. So, $q_{ij} = 1 - p_{ij}$ is the probability that the information is sent without being overheard. In order to send information to all n persons, we look for a spanning tree for which the product of q_{ij}'s is minimum. So, we assign $w_{ij} = \log q_{ij}$.

(Example. Bottleneck problem) For a path W from v to w, $c(W) := \min\{w(e_i) : i = 1, \ldots, n\}$ is called the capacity of W. For each pair (v, w), we want to determine a path from u to v with maximal capacity.

Hu [The maximum capacity route problem. Oper Res. 1961] reduced this problem to finding a maximal spanning tree.

Theorem 5. *Let T be a maximal spanning tree for (G, w). Then for each pair (v, w) of vertices, the unique path from u to v in T is a path of maximal capacity in G.*