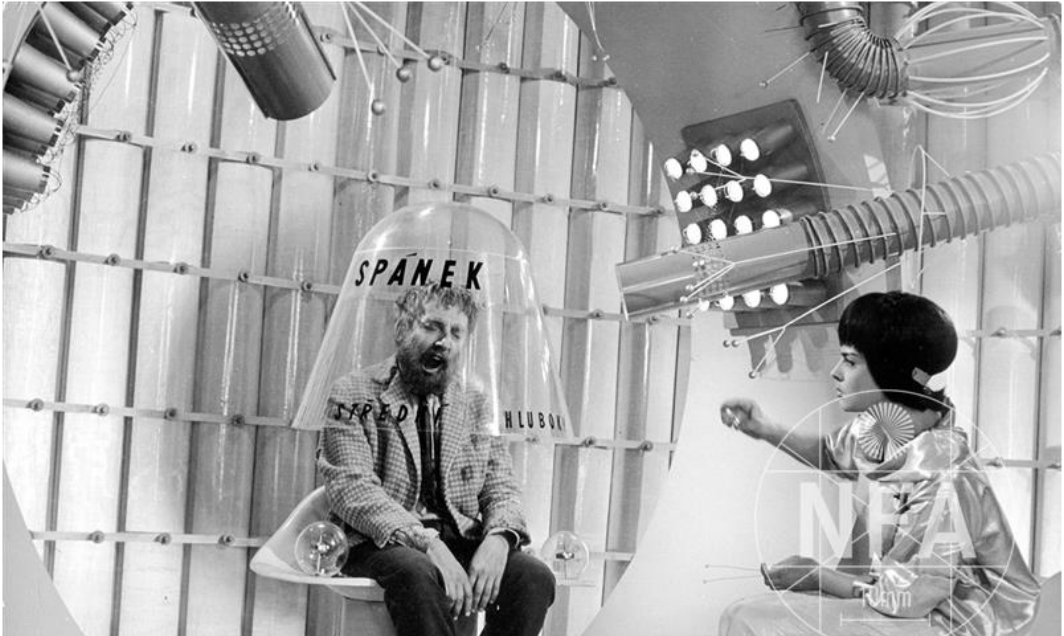
Algorithms and datastructures I
Lecture 6: shortest paths and minimum spaning trees

Jan Hubička

Department of Applied Mathematics
Charles University
Prague

March 24 2020


SPÁNEK

. . . and ask me questions during the lecture

Past lecture (self study)

Definition (Edge-valued graph)

We equip a given graph $G = (V, E)$ with a function $\ell : E \to \mathbb{R}$ defining the length (or label) of a given edge. This way we create an edge-valued graph (sometimes also called network).

Past lecture (self study)

Definition (Edge-valued graph)

We equip a given graph $G = (V, E)$ with a function $\ell : E \to \mathbb{R}$ defining the length (or label) of a given edge. This way we create an edge-valued graph (sometimes also called network).

Definition (Length of a walk)

Given a walk W in graph G, the length of W, written as $\ell(e)$ is defined as

$$\sum_{e \in E(W)} \ell(e).$$

Past lecture (self study)

Definition (Edge-valued graph)

We equip a given graph $G = (V, E)$ with a function $\ell : E \to \mathbb{R}$ defining the length (or label) of a given edge. This way we create an edge-valued graph (sometimes also called network).

Definition (Length of a walk)

Given a walk W in graph G, the length of W, written as $\ell(e)$ is defined as

$$\sum_{e \in E(W)} \ell(e).$$

Definition (Distance in edge-valued graph)

Given two vertices $u, v \in V(G)$ their distance, denoted by $d(u, v)$, is the minimum over lengths of all possible paths from u to v. Every path from u to v of length $d(u, v)$ is called a shortest a path from u to v.

Negative versus non-negative lengths

Lemma (About simplifying walks)

Let G be an labelled graph such that all lengths are non-negative. Then for every walk W from u to v there exists a path P from u to v such that $\ell(P) \leq \ell(W)$.

Dijkstra's algorithm

Input: Graph $G = (V, E)$, labelling of edges ℓ by non-negative reals and initial vertex v_0

For every vertex $v \in V$:
  state$(v) \leftarrow$ unvisited
  $h(v) \leftarrow \infty$
  $P(v) \leftarrow$ undefined
state$(v_0) \leftarrow$ open
$h(v_0) \leftarrow 0$
While there are open vertices:
  Choose open vertex v with minimal $h(v)$
  For every w such that $(v, w) \in E$
    If $h(w) > h(v) + \ell(v, w)$ then:
      $h(w) \leftarrow h(v) + \ell(v, w)$
      state$(w) \leftarrow$ open
      $P(w) \leftarrow v$
  state$(v) \leftarrow$ closed

Output: Array of distances h, array of predecessors P

Dijkstra's algorithm

Input: Graph $G = (V, E)$, labelling of edges ℓ by non-negative reals and initial vertex v_0

- For every vertex $v \in V$:
  - state(v) $\leftarrow$ unvisited
  - $h(v) \leftarrow \infty$
  - $P(v) \leftarrow$ undefined
- state(v_0) $\leftarrow$ open
- $h(v_0) \leftarrow 0$
- While there are open vertices:
  - Choose open vertex v with minimal $h(v)$
  - For every w such that $(v, w) \in E$
    - If $h(w) > h(v) + \ell(v, w)$ then:
      - $h(w) \leftarrow h(v) + \ell(v, w)$
      - state(w) $\leftarrow$ open
      - $P(w) \leftarrow v$
  - state(v) $\leftarrow$ closed

Output: Array of distances h, array of predecessors P

Relaxation (meta)algorithm

Input: Graph $G = (V, E)$, labelling of edges ℓ by non-negative reals and initial vertex v_0

- For every vertex $v \in V$:
  - state(v) $\leftarrow$ unvisited
  - $h(v) \leftarrow \infty$
  - $P(v) \leftarrow$ undefined
- state(v_0) $\leftarrow$ open
- $h(v_0) \leftarrow 0$
- While there exists open vertex v:
  - For every w such that $(v, w) \in E$
    - If $h(w) > h(v) + \ell(v, w)$ then:
      - $h(w) \leftarrow h(v) + \ell(v, w)$
      - state(w) $\leftarrow$ open
      - $P(w) \leftarrow v$
  - state(v) $\leftarrow$ closed

Output: Array of distances h, array of predecessors P

Relaxation (meta)algorithm

Input: Graph $G = (V, E)$, labelling of edges ℓ by non-negative reals and initial vertex v_0

1. For every vertex $v \in V$:
2.   state$(v) \leftarrow$ unvisited
3. $h(v) \leftarrow \infty$
4. $P(v) \leftarrow$ undefined
5. state$(v_0) \leftarrow$ open
6. $h(v_0) \leftarrow 0$
7. While there exists open vertex v:
8.   For every w such that $(v, w) \in E$
9.   If $h(w) > h(v) + \ell(v, w)$ then:
10. $h(w) \leftarrow h(v) + \ell(v, w)$
11.   state$(w) \leftarrow$ open
12. $P(w) \leftarrow v$
13.   state$(v) \leftarrow$ closed

Output: Array of distances h, array of predecessors P

Relaxation (meta)algorithm

Input: Graph $G = (V, E)$, labelling of edges ℓ by non-negative reals and initial vertex v_0

For every vertex $v \in V$:
state$(v) \leftarrow$ unvisited
$h(v) \leftarrow \infty$
$P(v) \leftarrow$ undefined
state$(v_0) \leftarrow$ open
$h(v_0) \leftarrow 0$
While there exists open vertex v:
For every w such that $(v, w) \in E$
If $h(w) > h(v) + \ell(v, w)$ then:
$h(w) \leftarrow h(v) + \ell(v, w)$
state$(w) \leftarrow$ open
$P(w) \leftarrow v$
state$(v) \leftarrow$ closed

Output: Array of distances h, array of predecessors P

Invariant about values

$h(v)$ never increases and always corresponds to a length of some walk.

Relaxation (meta)algorithm

Input: Graph $G = (V, E)$, labelling of edges ℓ by non-negative reals and initial vertex v_0

For every vertex $v \in V$:
    state$(v) \leftarrow$ unvisited
    $h(v) \leftarrow \infty$
    $P(v) \leftarrow$ undefined
state$(v_0) \leftarrow$ open
$h(v_0) \leftarrow 0$
While there exists open vertex v:
    For every w such that $(v, w) \in E$
        If $h(w) > h(v) + \ell(v, w)$ then:
            $h(w) \leftarrow h(v) + \ell(v, w)$
            state$(w) \leftarrow$ open
            $P(w) \leftarrow v$
    state$(v) \leftarrow$ closed

Output: Array of distances h, array of predecessors P

Invariant about values

$h(v)$ never increases and always corresponds to a length of some walk.

Lemma (on reachability)

At the end of computation $h(v)$ is finite $\iff$ v is reachable from v_0

Relaxation (meta)algorithm

Input: Graph $G = (V, E)$, labelling of edges ℓ by non-negative reals and initial vertex v_0

- For every vertex $v \in V$:
  - state$(v) \leftarrow$ unvisited
  - $h(v) \leftarrow \infty$
  - $P(v) \leftarrow$ undefined
- state$(v_0) \leftarrow$ open
- $h(v_0) \leftarrow 0$
- While there exists open vertex v:
  - For every w such that $(v, w) \in E$
    - If $h(w) > h(v) + \ell(v, w)$ then:
      - $h(w) \leftarrow h(v) + \ell(v, w)$
      - state$(w) \leftarrow$ open
      - $P(w) \leftarrow v$
  - state$(v) \leftarrow$ closed

Output: Array of distances h, array of predecessors P

Invariant about values

$h(v)$ never increases and always corresponds to a length of some walk.

Lemma (on reachability)

At the end of computation $h(v)$ is finite $\iff$ v is reachable from v_0

Lemma (on distance)

If there are no negative cycles in graph G, at the end of computation $h(v) = d(v_0, v)$

Bellman-Ford algorithm

This is a relaxation algorithm created from Dijkstra's algorithm by replacing heap by queue (thus it always closes the vertex which was open for longest time).

Definition (Stage of a computation)

Stage of computation is defined as follows:

- S_0 is the stage algorithm opens v_0
- S_{i+1} closes all vertices opened during stage S_i

Bellman-Ford algorithm

This is a relaxation algorithm created from Dijkstra's algorithm by replacing heap by queue (thus it always closes the vertex which was open for longest time).

Definition (Stage of a computation)

Stage of computation is defined as follows:

S_0 is the stage algorithm opens v_0

S_{i+1} closes all vertices opened during stage S_i

Invariant on stages

At the end of stage S_i the $h(v)$ is bounded by above by the length of shortest walk from v_0 to v with at most i edges.

Bellman-Ford algorithm

This is a relaxation algorithm created from Dijkstra's algorithm by replacing heap by queue (thus it always closes the vertex which was open for longest time).

Definition (Stage of a computation)

Stage of computation is defined as follows:

- S_0 is the stage algorithm opens v_0
- S_{i+1} closes all vertices opened during stage S_i

Invariant on stages

At the end of stage S_i the $h(v)$ is bounded by above by the length of shortest walk from v_0 to v with at most i edges.

Corollary

If G has no negative cycles, the algorithm will finish.

Bellman-Ford algorithm

This is a relaxation algorithm created from Dijkstra's algorithm by replacing heap by queue (thus it always closes the vertex which was open for longest time).

Definition (Stage of a computation)

Stage of computation is defined as follows:

S_0 is the stage algorithm opens v_0

S_{i+1} closes all vertices opened during stage S_i

Invariant on stages

At the end of stage S_i the $h(v)$ is bounded by above by the length of shortest walk from v_0 to v with at most i edges.

Corollary

If G has no negative cycles, the algorithm will finish.

Theorem

If G has no negative cycles, Bellman-Ford algorithm will find the shortest distances in time $O(nm)$.

Floyd-Washall algorithm

Let G be a graph with vertices $V = \{1, 2, \ldots n\}$.
Instead of distances from a given vertex v_0 we want to compute distance matrix D such that $D_{i,j} = d(i, j)$.

Floyd-Washall algorithm

Let G be a graph with vertices $V = \{1, 2, \ldots n\}$.
Instead of distances from a given vertex v_0 we want to compute distance matrix D such that $D_{i,j} = d(i, j)$.

Definition

Let D^k be a matrix such that $D^k_{i,j}$ is the length of shortest path from i to j such that all internal vertices are in $\{1, 2, \ldots, k\}$.

Floyd-Washall algorithm

Let G be a graph with vertices $V = \{1, 2, \ldots n\}$.
Instead of distances from a given vertex v_0 we want to compute distance matrix D such that $D_{i,j} = d(i, j)$.

Definition

Let D^k be a matrix such that $D^k_{i,j}$ is the length of shortest path from i to j such that all internal vertices are in $\{1, 2, \ldots, k\}$.

Floyd-Washall Algorithm

Input: Matrix of length of edges D^0

1. For $k = 0, \ldots, n-1$
2. For $i = 1, \ldots, n$
3. For $j = 1, \ldots, n$
4. $D^{k+1}_{i,j} = \min(D^k_{i,j}, D^k_{i,k+1} + D^K_{k+1,j})$

Output: Matrix of distances D^n

Floyd-Washall algorithm

Let G be a graph with vertices $V = \{1, 2, \dots n\}$.
Instead of distances from a given vertex v_0 we want to compute distance matrix D such that $D_{i,j} = d(i,j)$.

Definition

Let D^k be a matrix such that $D^k_{i,j}$ is the length of shortest path from i to j such that all internal vertices are in $\{1, 2, \dots, k\}$.

Floyd-Washall Algorithm

Input: Matrix of length of edges D^0

1. For $k = 0, \dots, n-1$
2. For $i = 1, \dots, n$
3. For $j = 1, \dots, n$
4. $D^{k+1}_{i,j} = \min(D^k_{i,j}, D^k_{i,k+1} + D^K_{k+1,j})$

Output: Matrix of distances D^n

Time complexity $\Theta(n^3)$.

Floyd-Washall algorithm

Let G be a graph with vertices $V = \{1, 2, \dots n\}$.
Instead of distances from a given vertex v_0 we want to compute distance matrix D such that $D_{i,j} = d(i, j)$.

Definition

Let D^k be a matrix such that $D^k_{i,j}$ is the length of shortest path from i to j such that all internal vertices are in $\{1, 2, \dots, k\}$.

Floyd-Washall Algorithm

Input: Matrix of length of edges D^0

1. For $k = 0, \dots, n-1$
2. For $i = 1, \dots, n$
3. For $j = 1, \dots, n$
4. $D^{k+1}_{i,j} = \min(D^k_{i,j}, D^k_{i,k+1} + D^K_{k+1,j})$

Output: Matrix of distances D^n

Time complexity $\Theta(n^3)$.
Memory complexity can be reduced by $\Theta(n^2)$ by modifying matrix “in place”
(it holds that $D^{k+1}_{k+1,j} = D^k_{k+1,j}$ and $D^{k+1}_{i,k+1} = D^k_{i,k+1}$).

Minimum spanning tree

Definition

1. Let $G = (V, E)$ be connected unoriented graph $w : E \to \mathbb{R}$ an weight function.
2. Let H be a subgraph of G, then the weight of H is the sum of weights of all edges in H.
3. Spanning tree of G is a subgraph T of G which is a tree and contains all vertices of G
4. Spanning tree is minimum if there is no spanning three of smaller weight.

Minimum spanning tree

Definition

1. Let $G = (V, E)$ be connected unoriented graph $w : E \to \mathbb{R}$ an weight function.
2. Let H be a subgraph of G, then the weight of H is the sum of weights of all edges in H.
3. Spanning tree of G is a subgraph T of G which is a tree and contains all vertices of G
4. Spanning tree is minimum if there is no spanning three of smaller weight.

For simplicity: assume that for $e \neq e' \in E$ it holds that $w(e) \neq w(e')$ (weights are unique)

Jarník algorithm, 1930 (Prim, 1957, Dijkstra in 1959)

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $v_0 \leftarrow$ arbitrary vertx in V
2. $T \leftarrow (\{v_0\}, \emptyset)$
3. While there exists edge $\{u, v\} \in E$ such that $u \in V(T)$ and $v \notin V(T)$:
4. Add minimal such edge to T

Output: Minimum spanning tree T.

Minimum spanning tree

Definition

1. Let $G = (V, E)$ be connected unoriented graph $w : E \to \mathbb{R}$ an weight function.
2. Let H be a subgraph of G, then the weight of H is the sum of weights of all edges in H.
3. Spanning tree of G is a subgraph T of G which is a tree and contains all vertices of G
4. Spanning tree is minimum if there is no spanning three of smaller weight.

For simplicity: assume that for $e \neq e' \in E$ it holds that $w(e) \neq w(e')$ (weights are unique)

Jarník algorithm, 1930 (Prim, 1957, Dijkstra in 1959)

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $v_0 \leftarrow$ arbitrary vertx in V
2. $T \leftarrow (\{v_0\}, \emptyset)$
3. While there exists edge $\{u, v\} \in E$ such that $u \in V(T)$ and $v \notin V(T)$:
4. Add minimal such edge to T

Output: Minimum spanning tree T.

Lemma

Algorithm will finish in $\leq n$ steps and will return some spanning tree.

Minimum spanning trees and cuts

Definition (Elementary cut)

Given graph $G = (V, E)$, we call $C \subseteq E$ an elementary cut if there exists $A, B \subseteq V$ such that $A \cap B = 0$ $A \cup B = V$ and $C = \{\{a, b\} \in E | a \in A, b \in B\}$.

Minimum spanning trees and cuts

Definition (Elementary cut)

Given graph $G = (V, E)$, we call $C \subseteq E$ an elementary cut if there exists $A, B \subseteq V$ such that $A \cap B = 0$ $A \cup B = V$ and $C = \{\{a, b\} \in E | a \in A, b \in B\}$.

Lemma (Cut lemma)

Let G be a graph and w a weight function with unique weights, C an elementary cut in G and e minimum edge in C. Then e belongs to every minimum spanning tree in G.

Jarník algorithm

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $v_0 \leftarrow$ arbitrary vertx in V
2. $T \leftarrow (\{v_0\}, \emptyset)$
3. While there exists edge $\{u, v\} \in E$ such that $u \in V(T)$ and $v \notin V(T)$:
4. Add minimal such edge to T

Output: Minimum spanning tree T.

Minimum spanning trees and cuts

Definition (Elementary cut)

Given graph $G = (V, E)$, we call $C \subseteq E$ an elementary cut if there exists $A, B \subseteq V$ such that $A \cap B = 0$ $A \cup B = V$ and $C = \{\{a, b\} \in E | a \in A, b \in B\}$.

Lemma (Cut lemma)

Let G be a graph and w a weight function with unique weights, C an elementary cut in G and e minimum edge in C. Then e belongs to every minimum spanning tree in G.

Jarník algorithm

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $v_0 \leftarrow$ arbitrary vertx in V
2. $T \leftarrow (\{v_0\}, \emptyset)$
3. While there exists edge $\{u, v\} \in E$ such that $u \in V(T)$ and $v \notin V(T)$:
4. Add minimal such edge to T

Output: Minimum spanning tree T.

Consequences to Jarník algorithm:

1. Every edge chosen by the algorithm is minimum in some elementary cut.
2. $T \subseteq M$ for every minimum spanning tree M.
3. $T = M$ for every minimum spanning tree M.

Theorem

Let G be a connected graph and w a weight function with unique weight. Jarník algorithm will find its spanning tree in time $O(nm)$.

Minimum spanning trees and cuts

Definition (Elementary cut)

Given graph $G = (V, E)$, we call $V \subseteq E$ an elementary cut if there exists $A, B \subseteq V$ such that:
$A \cap B = \emptyset$ $A \cup B = C$ and $C = \{\{a, b\} \in E | a \in A, b \in B\}$.

Minimum spanning trees and cuts

Definition (Elementary cut)

Given graph $G = (V, E)$, we call $V \subseteq E$ an elementary cut if there exists $A, B \subseteq V$ such that: $A \cap B = \emptyset$ $A \cup B = C$ and $C = \{\{a, b\} \in E | a \in A, b \in B\}$.

Lemma (Cut lemma)

Let G be a graph and w a weight function with unique weights, C an elementary cut in G and e minimum edge in C. Then e belongs to every minimum spanning tree in G.

Proof.

Assume, to the contrary that there is elementary cut C, minimum edge $e \in C$ and minimum spanning tree T such that $e \notin T$.

Minimum spanning trees and cuts

Definition (Elementary cut)

Given graph $G = (V, E)$, we call $V \subseteq E$ an elementary cut if there exists $A, B \subseteq V$ such that: $A \cap B = \emptyset$ $A \cup B = C$ and $C = \{\{a, b\} \in E | a \in A, b \in B\}$.

Lemma (Cut lemma)

Let G be a graph and w a weight function with unique weights, C an elementary cut in G and e minimum edge in C. Then e belongs to every minimum spanning tree in G.

Proof.

Assume, to the contrary that there is elementary cut C, minimum edge $e \in C$ and minimum spanning tree T such that $e \notin T$. Then we can produce spanning tree T' of even smaller weight! □

Borůvka algorithm, 1926

Borůvka algorithm, 1926

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $T \leftarrow (V, \emptyset)$
2. While T is not connected:
3. Decompose T to (connected) components $T_1, \ldots T_k$.
4. For every tree T_i find minimum edge e_i between T_i and rest of a graph.
5. Add to T edges $\{e_1, \ldots, e_k\}$

Output: Minimum spanning tree T.

Borůvka algorithm, 1926

Borůvka algorithm, 1926

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $T \leftarrow (V, \emptyset)$
2. While T is not connected:
3. Decompose T to (connected) components $T_1, \ldots T_k$.
4. For every tree T_i find minimum edge e_i between T_i and rest of a graph.
5. Add to T edges $\{e_1, \ldots, e_k\}$

Output: Minimum spanning tree T.

Theorem

Algorithm will terminate in $\lfloor \log_2(n) \rfloor$ iterations and will return minimum spanning tree

Borůvka algorithm, 1926

Borůvka algorithm, 1926

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $T \leftarrow (V, \emptyset)$
2. While T is not connected:
3. Decompose T to (connected) components $T_1, \ldots T_k$.
4. For every tree T_i find minimum edge e_i between T_i and rest of a graph.
5. Add to T edges $\{e_1, \ldots, e_k\}$

Output: Minimum spanning tree T.

Theorem

Algorithm will terminate in $\lfloor \log_2(n) \rfloor$ iterations and will return minimum spanning tree

Proof.

After k iteration every connected component has at least 2^k vertices.

Borůvka algorithm, 1926

Borůvka algorithm, 1926

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $T \leftarrow (V, \emptyset)$
2. While T is not connected:
3. Decompose T to (connected) components $T_1, \dots T_k$.
4. For every tree T_i find minimum edge e_i between T_i and rest of a graph.
5. Add to T edges $\{e_1, \dots, e_k\}$

Output: Minimum spanning tree T.

Theorem

Algorithm will terminate in $\lfloor \log_2(n) \rfloor$ *iterations and will return minimum spanning tree*

Proof.

After k iteration every connected component has at least 2^k vertices.
Each edge e_i chosen is minimum in an elementary cut consisting of all edges out of T_i. □

Borůvka algorithm, 1926

Borůvka algorithm, 1926

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. $T \leftarrow (V, \emptyset)$
2. While T is not connected:
3. Decompose T to (connected) components $T_1, \ldots T_k$.
4. For every tree T_i find minimum edge e_i between T_i and rest of a graph.
5. Add to T edges $\{e_1, \ldots, e_k\}$

Output: Minimum spanning tree T.

Theorem

Algorithm will terminate in $\lfloor \log_2(n) \rfloor$ *iterations and will return minimum spanning tree*

Proof.

After k iteration every connected component has at least 2^k vertices.
Each edge e_i chosen is minimum in an elementary cut consisting of all edges out of T_i. □

Remark: Algorithm was later rediscovered by Florek, Łukasiewicz, Perkal, Steinhaus a Zubrzycki in 1951 and in 1960's by Sollin. It is useful in parallel computation and known as Sollin's algorithm.

Kruskal algorithm, 1956

Kruskal algorithm, 1956

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. Sort edges by weights; $w(e_1) \leq \cdots \leq w(e_m)$
2. $T \leftarrow (V, \emptyset)$
3. For $i = 1, \ldots m$:
4. $\quad u, v \leftarrow$ vertices in edge e_i
5. $\quad$ If u and v are in different components of T:
6. $\quad\quad T \leftarrow T + e_i$.

Output: Minimum spanning tree T.

Kruskal algorithm, 1956

Kruskal algorithm, 1956

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. Sort edges by weights; $w(e_1) \leq \cdots \leq w(e_m)$
2. $T \leftarrow (V, \emptyset)$
3. For $i = 1, \ldots m$:
4. $\quad u, v \leftarrow$ vertices in edge e_i
5. $\quad$ If u and v are in different components of T:
6. $\quad\quad T \leftarrow T + e_i$.

Output: Minimum spanning tree T.

Lemma

Kruskal algorithm will terminate and return minimum spanning tree.

Kruskal algorithm, 1956

Kruskal algorithm, 1956

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. Sort edges by weights; $w(e_1) \leq \cdots \leq w(e_m)$
2. $T \leftarrow (V, \emptyset)$
3. For $i = 1, \ldots m$:
4. $\quad u, v \leftarrow$ vertices in edge e_i
5. $\quad$ If u and v are in different components of T:
6. $\qquad T \leftarrow T + e_i$.

Output: Minimum spanning tree T.

Time complexity: $O(nm)$.

Kruskal algorithm, 1956

Kruskal algorithm, 1956

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. Sort edges by weights; $w(e_1) \leq \cdots \leq w(e_m)$
2. $T \leftarrow (V, \emptyset)$
3. For $i = 1, \ldots m$:
4. $\quad u, v \leftarrow$ vertices in edge e_i
5. $\quad$ If u and v are in different components of T:
6. $\qquad T \leftarrow T + e_i$.

Output: Minimum spanning tree T.

Definition (Union-find data-structure)

Data-structure Union-find represents connected components and supports operations

FIND(u, v) return true iff u and v are in same components

UNION(u, v) adds edge $\{u, v\}$ that unions the two components to a single component.

Kruskal algorithm, 1956

Kruskal algorithm, 1956

Input: Connected graph $G = (V, E)$ and weight function w with unique weights

1. Sort edges by weights; $w(e_1) \leq \cdots \leq w(e_m)$
2. $T \leftarrow (V, \emptyset)$
3. For $i = 1, \ldots m$:
4. $\quad u, v \leftarrow$ vertices in edge e_i
5. $\quad$ If u and v are in different components of T:
6. $\quad\quad T \leftarrow T + e_i$.

Output: Minimum spanning tree T.

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

We will show a data-structure which implements both FIND and UNION in $O(\log n)$.
From his we obtain $O(m \log n)$ running time.

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.
FIND(u, v): $O(1)$ (compare if $c(u) = c(v)$)
UNION(u, v): $O(n)$ (search array v and change all occurences of $c(u)$ to $c(v)$)

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.
FIND(u, v): $O(1)$ (compare if $c(u) = c(v)$)
UNION(u, v): $O(n)$ (search array v and change all occurences of $c(u)$ to $c(v)$)
Runtime of complete algorithm: $O(m \log n + m + n^2) = O(m \log n + n^2)$

Union-find using arrays

Theorem

Kruskal algorithm finds minimal spanning tree in time $O(m \log n + mT_f(n) + nT_u(n))$ where T_f is time complexity of FIND *and T_u is a time complexity of* UNION *on graph with n vertices.*

Idea: use array c. For a given vertex v put $c(v)$ to ID of a component it belongs to.
FIND(u, v): $O(1)$ (compare if $c(u) = c(v)$)
UNION(u, v): $O(n)$ (search array v and change all occurences of $c(u)$ to $c(v)$
Runtime of complete algorithm: $O(m \log n + m + n^2) = O(m \log n + n^2)$

Homework: Try to analyze variant where you always rename the smaller component in time $O(s)$ where s is the size of the component. (it does improve time complexity).