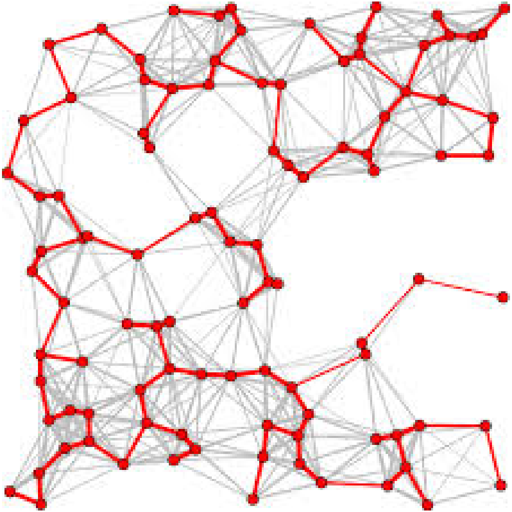
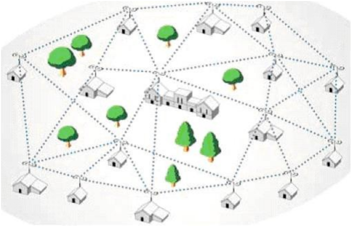
Minimum spanning trees



A network construction problem: Minimum Spanning Tree

CLRS 23, KT 4.5, DPV 5.1

- We have a set of locations.
- For some pairs of locations it is possible to build a link connecting the two locations, but it has a cost.

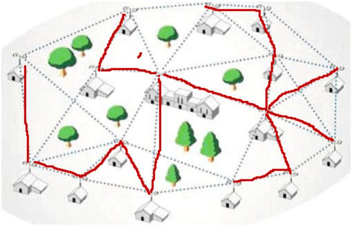


- We want to build a network (if possible), connecting all the locations, with total minimum cost.
- So, the resulting network must be a tree.

Network construction: Minimum Spanning Tree

- We have a set of locations. Build a link connecting the locations i and j has a cost $w(v_i, v_j)$.
- We want to build tree spanning all the locations with total minimum cost.

The MST



Properties of trees

- A tree on n nodes has $n-1$ edges.
- Any connected undirected graph with n vertices and $n-1$ edges is a tree.
- An undirected graph is a tree iff there is a unique path between any pair of nodes.

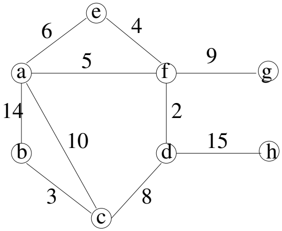
Let $G = (V, E)$ be a (undirected) graph.

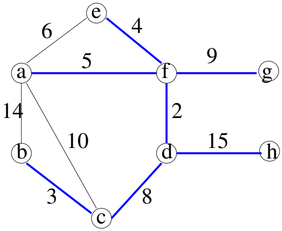
- $G' = (V', E')$ is a subgraph of G if $V' \subseteq V$ and $E' \subseteq E$.
- A subgraph $G' = (V', E')$ of G is spanning if $V' = V$.
- A spanning tree of G is a spanning subgraph that is a tree.

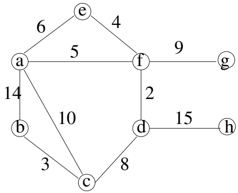
Any connected graph has a spanning tree

MINIMUM SPANNING TREE problem (MST)

Given as input an edge weighted graph $G = (V, E, w)$, where $w : E \to \mathbb{R}$. Find a tree $T = (V, E')$ with $E' \subseteq E$, such that it minimizes $w(T) = \sum_{e \in E(T)} w(e)$.







For a graph $G = (V, E)$:
A path is a sequence of consecutive edges.
A cycle is a path ending in an edge connecting to the initial vertex, with no other repeated vertex.
A cut is a partition of V into two sets S and $V - S$.
The cut-set of a cut is the set of edges with one end in S and the other in $V - S$. $cut(S, V - S) = \{e = (u, v) \in E \mid u \in S \; v \notin S\}$

MST: Properties

Given a weighted graph $G = (V, E, w)$, assume that all edge weights are different.

A MST T in G has the following properties:

- Cut property
  $e \in T \Leftrightarrow e$ is the lightest edge across some cut in G.
- Cycle property
  $e \notin T \Leftrightarrow e$ is the heaviest edge on some cycle in G.

The MST algorithms use two rules for adding/discarding edges.

MST: Properties

The $\Leftarrow$ implication of the cut property yields the blue rule (include), which allow us to include safely in T a min weight edge from some identified cut.

The $\Rightarrow$ implication of the cycle property will yield the red rule (exclude) which allow us to exclude from T a max weight edge from some identified cycles.

The cut property

Let $G = (V, E, w)$, $w : E \to \mathbb{R}^+$, such that all weights are different. Let T be a MST of G.

Removing an edge $e = (u, v)$ from T yields two disjoint trees T_u and T_v, so that $V(T_u) = V - V(T_v)$, $u \in T_u$ and $v \in T_v$. Let us call $S_u = V(T_u)$ and $S_v = V(T_v)$.

Claim

$e \in E(T)$ is the min-weight edge among those in $cut(S_u, S_v)$.

Proof.

Otherwise, we can replace e by an edge in the cut with smaller weight. Thus, forming a new spanning tree with smaller weight. □

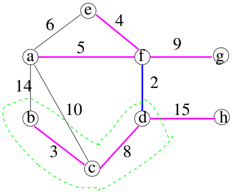
The cut property

Claim (The cut rule)

For $S \subseteq V$, let $e = (u, v)$ be the min-weight edge in $cut(S, V - S)$, then $e \in T$.

Proof.

Assume $e \notin T$ and that $u \in S$ and $v \notin S$. As T is a spanning tree there must be a path from u to v in T. As $u \in S$ and $v \notin S$, there is an edge $e' \in cut(S, V - S)$ in this path. Replacing e' with e produces another spanning tree. But then, as $w(e) > w(e')$, T was not optimal. □



The cycle property

For an edge $e \notin T$, adding it to T creates a graph $T + e$ having a unique cycle involving e. Lets call this cycle C_e.

Claim

For $e \notin E(T)$, e is the max-weight edge in C_e.

Proof.

Otherwise, removing any edge different from e in $T + e$ produces a spanning tree with smaller total weight. □

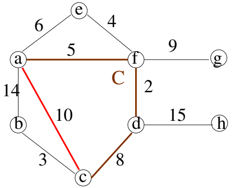
The cycle property

Claim (The cycle rule)

For a cycle C in G, the edge $e \in C$ with max-weight can not be part of T.

Proof.

Observe that, as G is connected, $G' = (V, E - \{e\})$ is connected. Furthermore, a MST for G' is a MST for G. □



C=cycle spanning {a,c,d,f}

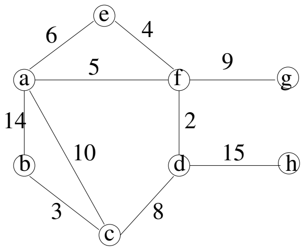
Generic greedy for MST: Apply blue and/or red rules

- The two rules show the optimal substructure of the MST. So, we can design a greedy algorithm.
- Blue rule: Given a cut-set between S and $V - S$ with no blue edges, select from the cut-set a non-colored edge with min weight and paint it blue
- Red rule: Given a cycle C with no red edges, selected a non-colored edge in C with max weight and paint it red.
- *Greedy scheme:*

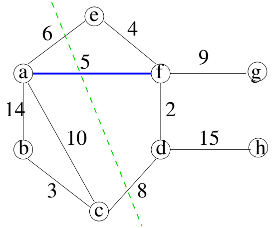
  Given G, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

  Robert Tarjan: Data Structures and Network Algorithms, SIAM, 1984

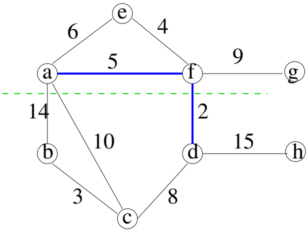
Application of red/blue rules



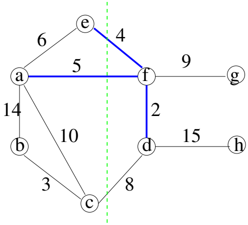
Application of red/blue rules



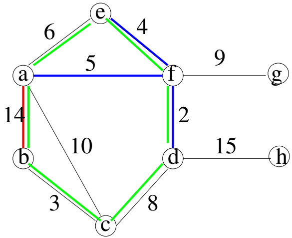
Application of red/blue rules



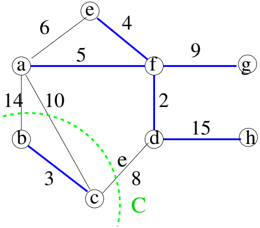
Application of red/blue rules



Application of red/blue rules



Application of red/blue rules



Greedy for MST : Correctness

Theorem

The greedy scheme finishes in at most m steps and at the end of the execution the blue edges form a MST

Sketch.

- As in each iteration an edge is added or discarded, the algorithm finishes after at most m applications of the rules.
- As the red edges cannot form part of any MST and the blue ones belong to some MST, the selections are correct.
- A set of $n - 1$ required edges form a spanning tree!

□

We need implementations for the algorithm!

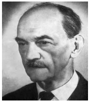
A short history of MST implementation

There has been extensive work to obtain the most efficient algorithm to find a MST in a given graph:

- O. Borůvka gave the first greedy algorithm for the MST in 1926. V. Jarnik gave a different greedy for MST in 1930, which was re-discovered by R. Prim in 1957. In 1956 J. Kruskal gave a different greedy algorithms for the MST. All those algorithms run in $O(m \lg n)$.
- Fredman and Tarjan (1984) gave a $O(m \log^* n)$ algorithm, introducing a new data structure for priority queues, the Fibbonacci heap. Recall $\log^* n$ is the number of times we have to apply iteratively the log operator to n to get a value ≤ 1, for ex. $\log^* 1000 = 2$.
- Gabow, Galil, Spencer and Tarjan (1986) improved Fredman-Tarjan to $O(m \log(\log^* n))$.
- Karger, Klein and Tarjan (1995) $O(m)$ randomized algorithm.
- In 1997 B. Chazelle gave an $O(m\alpha(n))$ algorithm, where $\alpha(n)$ is a very slowly growing function, the inverse of the Ackermann function.

Basic algorithms for MST

- Jarník-Prim (Serial centralized) Starting from a vertex v, grows T adding each time the lighter edge already connected to a vertex in T, using the blue rule.
  Uses a priority queue
- Kruskal (Serial distributed) Considers every edge, in order of increasing weight, to grow a forest by using the blue and red rules. The algorithm stops when the forest became a tree.
  Uses a union-find data structure.



Jarník - Prim greedy algorithm.

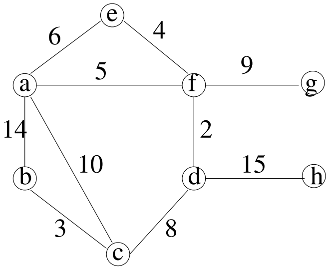
V. Jarník, 1936, R. Prim, 1957

- The algorithmgs keeps a tree T and adds one edge (and one node) to T at each step.
- Initially the tree T has one arbitrary node r, and no edges.
- At each step T is enlarged adding a minimum weight edge in the $C(T) = cut - set(V(T), V - V(T))$.
- Note that an edge e is in the cut-set if e has one end in $V(T)$ and the other outside.

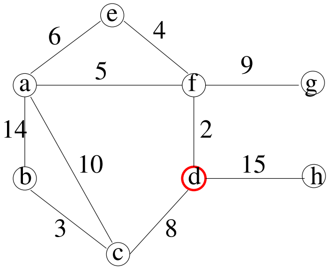
Jarník - Prim greedy algorithm.

MST (G, w, r)
$T = \{r\}$
for $i = 2$ **to** $|V|$ **do**
  Let e be a min weight edge in the $cut(V(T), V - V(T))$
  $T = T \cup \{e\}$
end for

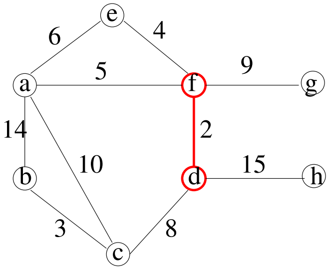
Example.



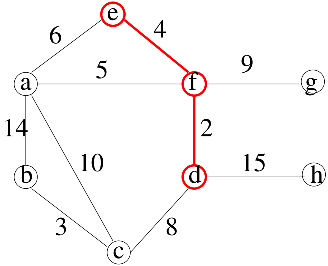
Example.



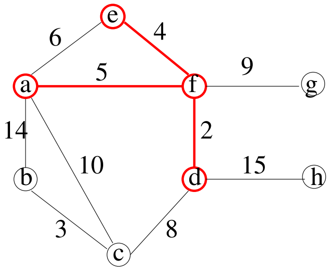
Example.



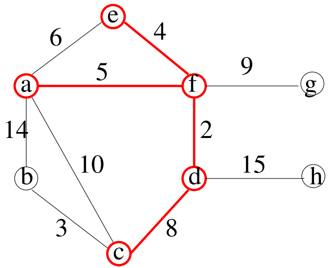
Example.



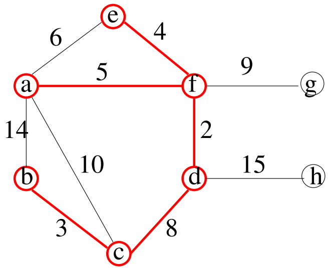
Example.



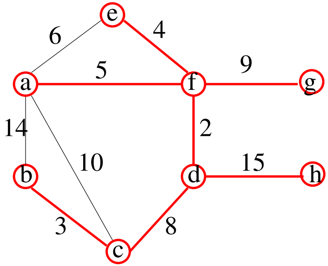
Example.



Example.



Example.



$w(T) = 52$

Jarník - Prim greedy algorithm.

Use a priority queue to choose min weight e in the cut set. In doing so we have to discard some edges

MST (G, w, r)
$T = (\{r\}, \emptyset)$; $Q = \emptyset$; $s = 0$
Insert in Q all edges $e = (r, v)$ with key $w(r, v)$
while $s < n - 1$ and Q is not empty **do**
  $(u, v, w) = Q.pop()$
  if $u \notin V(T)$ or $v \notin V(T)$ **then**
    Let u' be the vertex from (u, v) that is not in T
    Insert in Q all the edges $e = (u', v') \in E(G)$ for $v' \notin V(T)$ with key $w(e)$
    add e to T; $++s$
  end if
end while

Jarník - Prim greedy algorithm: Correctness

- The algorithm discards edge e:
  Such an edge $e = (u, v)$ has $u, v \in V(T)$, so it forms a cycle with the edges in T. But, e is the edge with highest weight in this cycle. This is an application of the red rule.
- The algorithm adds to T edge e:
  Then e has minimum weight among all edges in Q, as Q contains all edges in the cut-set$(V(T), V - V(T))$. This is the blue rule
- Therefore the algorithm computes a MST.

Jarník - Prim greedy algorithm: Cost

Time: depends on the implementation of the priority queue Q.
We have $\leq m$ insertions on the priority queue.

Q an unsorted array: $T(n) = O(|V|^2)$;
Q a heap: $T(n) = O(|E| \lg |V|)$.
Q a Fibonacci heap: $T(n) = O(|E| + |V| \lg |V|)$

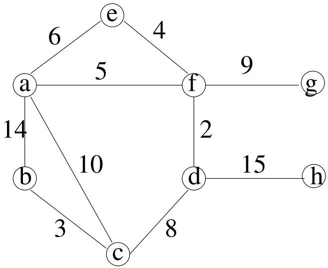
Kruskal's algorithm.

J. Kruskal, 1956

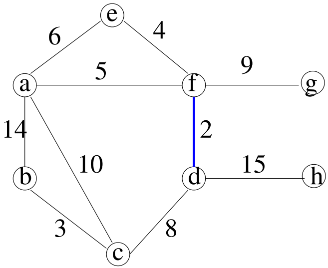
Similar to Jarník - Prim, but chooses minimum weight edges, in some cut. The selected edges form a forest until the last step.

MST (G, w, r)
Sort E by increasing weight
$T = \emptyset$
for $i = 1$ **to** $|V|$ **do**
  Let $e \in E$: with minimum weight among those that do not form a cycle with T
  $T = T \cup \{e\}$
end for

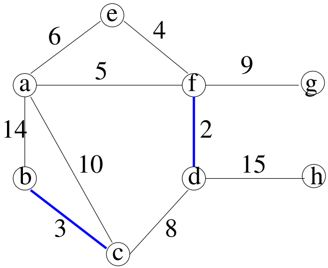
Example.



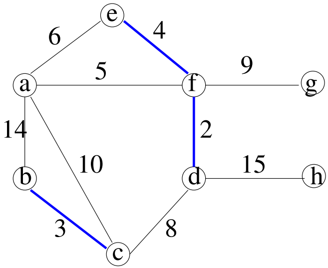
Example.



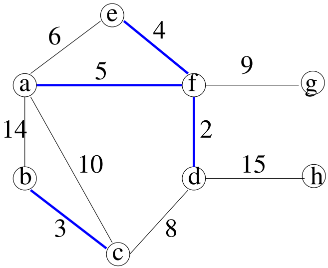
Example.



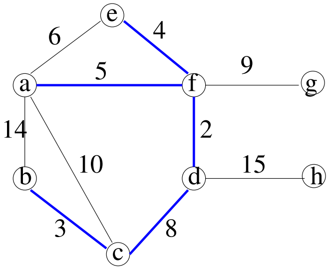
Example.



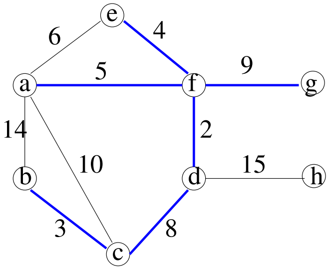
Example.



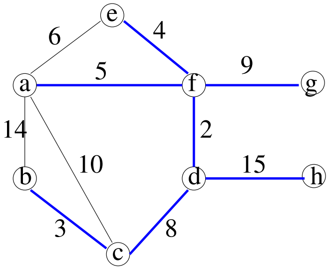
Example.



Example.



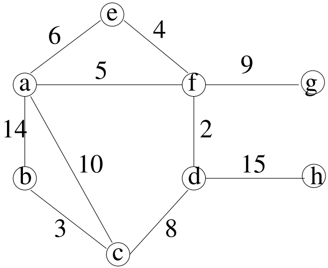
Example.



Kruskal's algorithm: Implementation

- We have a cost of $O(m \lg m)$ as we have to sort the edges. But as $m \leq n^2$, $O(m \lg m) = O(m \lg n)$.
- We need an efficient implementation of the algorithm.
- To find an adequate data structure lets look to some properties of the objects constructed along the execution of the algorithm.

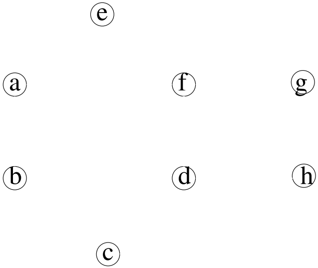
Another view of Kruskal's algorithm



edges sorted by weight

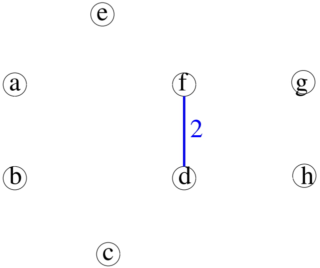
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



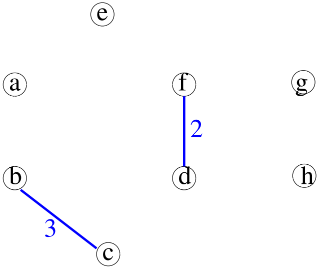
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



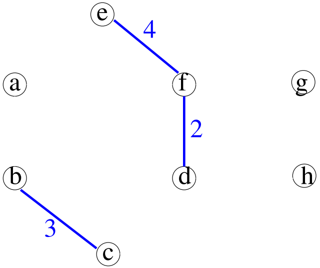
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



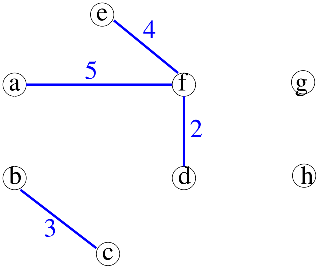
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



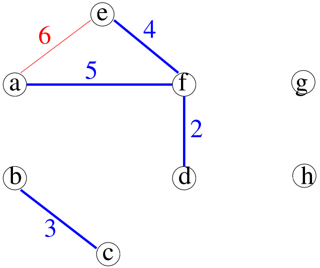
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



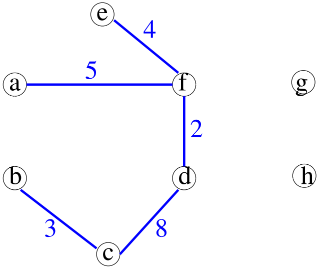
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8), (f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



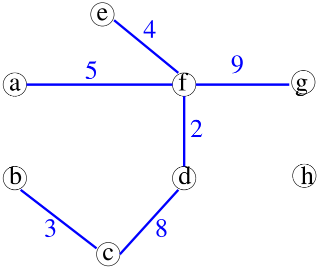
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



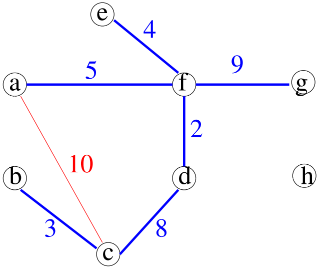
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



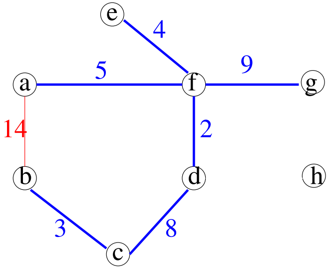
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



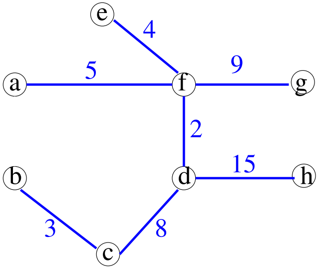
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Using Union-Find for Kruskal

- Kruskal evolves by building spanning forests, merging two trees (blue rule) or discarding an edge (red rule) so as to do not create a cycle.

- The connectivity relation is an equivalence relation: $u\mathcal{R}_F v$ iff there is a path between u and v.
- Kruskal, starts with a partition of V into n sets and ends with a partition of V into one set.
- $\mathcal{R}$ partition the elements of V in equivalence classes, which are the connected components of the forest

Disjoint Set Union-Find

B. Galler, M. Fisher: An improved equivalence algorithm. ACM Comm., 1964; R.Tarjan 1979-1985

- Union-Find is a data structure to maintain a dynamic partition of a set.
- Union-Find is one of the most elegant data structures in the algorithmic toolkit.
- Union-Find makes possible to design almost linear time algorithms for problems that otherwise would be unfeasible.
- Union-Find is a first introduction to an active research fields in algorithmic; Self organizing data structures and data stream computation.

Partition and equivalent relations

Remember a **partition** of an n element set S is collection $\{S_1, \ldots, S_k\}$ of subsets s.t.:

$$\forall S_i \subseteq S; \cup_{i=1}^{k} S_i = S; \forall S_i, S_j \text{ then } S_i \cap S_j = \emptyset$$

.

Recall also that a partition implies an equivalence relation:

$$\forall x, y \in S, x \equiv y \text{ iff } x \in S_i \& y \in S_i.$$

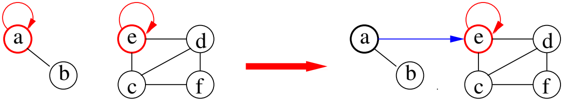
The collection $\{S_1, \ldots, S_k\}$ are the equivalence classes of the equivalence relation.

Union-Find

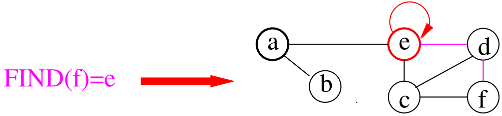
Union-Find supports three operations on partitions of a set:

MAKESET (x): creates a new set containing the single element x.

UNION (x, y): Merge the sets containing x and y, by using their union.



FIND (x): Return the representative of the set containing x.



Warning about UNION operation

- Warning: For any $x, y \in S$, we might need to do UNION(x, y), for x, y that are not representatives. Depending on the implementation this might or might not be allowed.
- To determine the complexity under different implementations, we consider that

$$\text{UNION } (x, y) = \text{UNION } (\text{FIND}(x), \text{FIND}(y)).$$

Union-Find implementation for Kruskal

MST $(G(V,E),w,r)$, $|V|=n,|E|=m$
Sort E by increasing weight: $\{e_1,\ldots,e_m\}$
$T=\emptyset$
for all $v\in V$ **do**
  MAKESET(v)
end for
for $i=1$ **to** m **do**
  Assume that $e_i=(u,v)$
  if FIND(u) $\neq$ Find(v) **then**
    $T=T\cup\{e_i\}$
    UNION(u,v)
  end if
end for

- Sorting takes time $O(m\log n)$.
- The remaining part of the algorithm is a sequence of n MAKESET and $O(m)$ operations of type FIND/UNION

Amortized analysis

(See for ex. Sect. 17-1 to 17.3 in CLRS)

- An amortized analysis is any strategy for analyzing a sequence of operations on a Data Structure, to show that the "average" cost per operation is small, even though a single operation within the sequence might be expensive.
- An amortized analysis guarantees the average performance of each operation in the worst case.
- The easier way to think about amortized analysis is to consider total number of steps for a sequence of operations of a given size.

Union Find implementations: Cost

(4.6 KT)

For a set with n elements.

- Using an array holding the representative.
  - MAKESET and FIND takes $O(1)$
  - UNION takes $O(n)$.
- Using an array holding the representative, a list by set, and in a UNION keeping the representative of the larger set.
  - MAKESET and FIND takes $O(1)$
  - any sequence of k UNION takes $O(k \log k)$.

Complexity of Union Find implementations: Amortized cost

For a set with n elements.

- Using a rooted tree by set, in a UNION keeping the representative of the larger set.
  - MAKESET and UNION takes $O(1)$
  - FIND takes $O(\log n)$.
- Using a rooted tree by set, in a UNION keeping the representative of the larger set, and doing path compression during a FIND.
  - MAKESET takes $O(1)$
  - any intermixed sequence of k FIND and UNION takes $O(k\alpha(n))$.

$\alpha(n)$ is the inverse Ackerman's function which grows extremely slowly. For practical applications it behaves as a constant.

Union-Find implementation for Kruskal

MST $(G(V,E), w, r)$, $|V| = n, |E| = m$
Sort E by increasing weight: $\{e_1, \ldots, e_m\}$
$T = \emptyset$
for all $v \in V$ **do**
  MAKESET(v)
end for
for $i = 1$ **to** m **do**
  Assume that $e_i = (u, v)$
  if FIND$(u) \neq$ Find(v) **then**
    $T = T \cup \{e_i\}$
    UNION(u, v)
  end if
end for

- Sorting take time $O(m \log n)$.
- The remaining part of the algorithm has cost $n + O(m\alpha(n)) = O(n + m)$.

But due to the sorting instruction, Kuskal takes $O(n + m \lg n)$.
Unless we use a range of weights that allow us to use RADIX.

Some applications of Union-Find

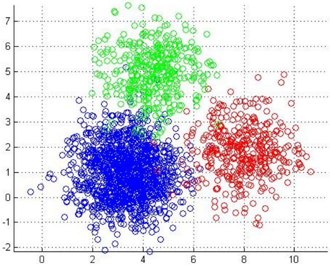
- Kruskal's algorithm for MST.
- Dynamic graph connectivity in very large networks.
- Cycle detection in undirected graphs.
- Random maze generation and exploration.
- Strategies for games: Hex and Go.
- Least common ancestor.
- Compiling equivalence statements.
- Equivalence of finite state automata.

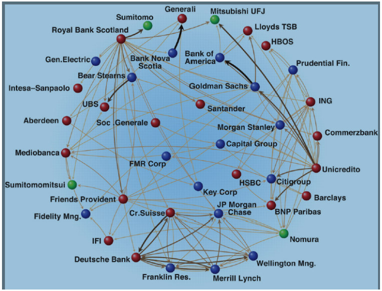
Clustering

- Clustering: process of finding interesting structure in a set of data.
- Given a collection of objects, organize them into similar coherent groups with respect to some (distance function $d(\cdot, \cdot)$).
- The distance function not necessarily has to be the physical (Euclidean) distance. The interpretation of $d(\cdot, \cdot)$ is that for any two objects x, y, the larger that $d(x, y)$ is, the less similar that x and y are.
- There are many problems in clustering, but for most of them, $d(\cdot, \cdot)$ must have be a metric: $d(x, x) = 0$ and $d(x, y) > 0$, for $x \neq y$; $d(x, y) = d(y, x)$; $d(x, y) + d(y, z) \leq d(x, z)$.
- If x, y are two species, we can define $d(x, y)$ as the years that they diverged in the course of evolution.

Generic clustering setting

Given a set of data points $\mathcal{U} = \{x_1, x_2, \ldots, x_n\}$ together with a distance function d on X, and given a $k > 0$, a *k-clustering* is a partition of X into k disjoint subsets.





The single-link clustering problem

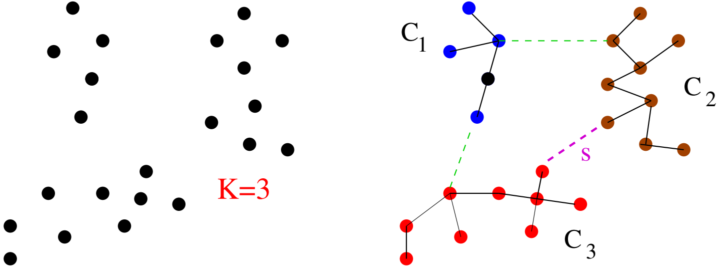
Let $\mathcal{U}$ be a set of n data points, assume $\{C_1, \ldots, C_k\}$ is a k-clustering for $\mathcal{U}$.
Define the spacing s in the k-clustering as the minimum distance between any pair of points in different clusters.

The single-link clustering problem: Given $\mathcal{U} = \{x_1, x_2, \ldots, x_n\}$, a distance function d, and $k > 0$, find a k-clustering of $\mathcal{U}$ maximizing the spacing s.

Notice there are exponentially many different k-clustering of $\mathcal{U}$.

TrKruskal: An algorithm for the single-link clustering problem

- Represent $\mathcal{U}$ as vertices of an undirected graph where the edge (x, y) has weight $d(x, y)$.
- Apply Kruskal's algorithm until the forest has k trees.



Complexity and correctness

Theorem

TrKruskal solves the single-link clustering problem in $O(n^2 \lg n)$

Proof.

We have to create a complete graph and sort the n^2 edges. This has cost $O(n^2 \lg n)$

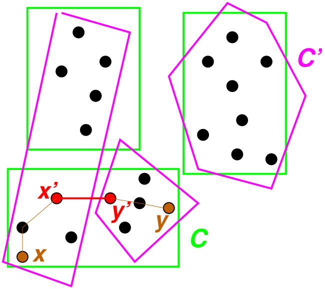
Correctness

Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be the k-clustering produced by TrKruskal, and let s be its spacing.

Assume there is another k-clustering $\mathcal{C}' = \{C'_1, \ldots, C'_k\}$ with spacing s' and s.t. $\mathcal{C} \neq \mathcal{C}'$. We must show that $s' \leq s$.

Complexity and correctness

If $\mathcal{C} \neq \mathcal{C}'$, then $\exists C_r \in \mathcal{C}$ s.t. $\forall C'_t \in \mathcal{C}', C_r \not\subseteq C'_t$.
That means $\exists x, y \in C_r$ s.t. $x, y \in C_r$ s.t. $x \in C'_t$ and $y \in C'_q$.
$\exists$ a path $x \rightsquigarrow y$ in $C_r \Rightarrow \exists (x', y') \in E(\text{MST})$ with $x' \in C'_t$ and $y' \in C'_q$ and s.t. $s' \leq d(x', y') \leq s$.



End Proof