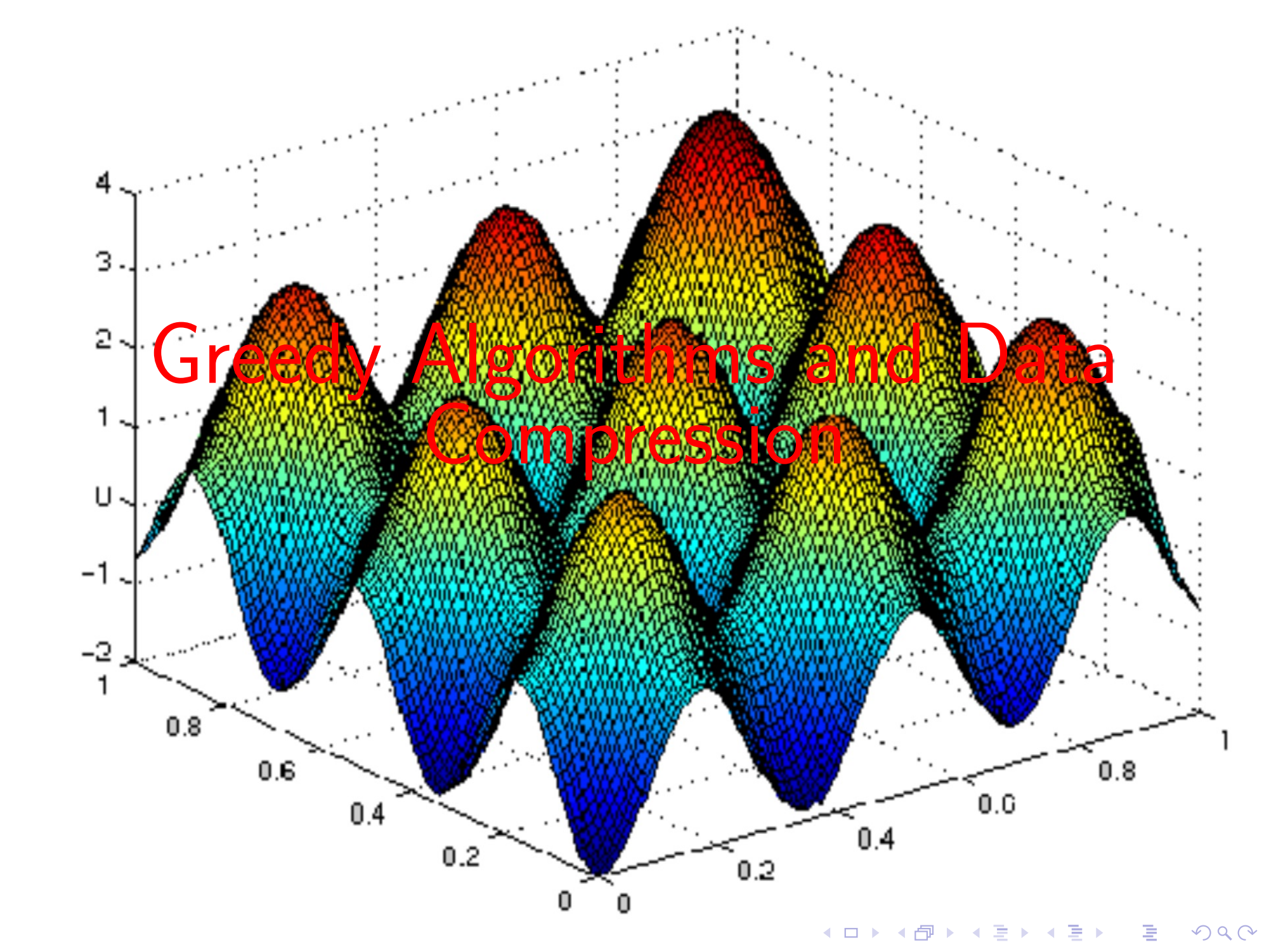

Greedy Algorithms and Data Compression
4
3
2
1
0
-1
-2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

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

The greedy algorithms for MST color edges of G with blue (in the MST) or red (not in MST) using the rules:

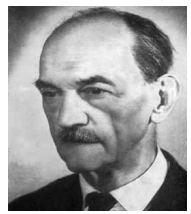
Blue rule: Given a cut-set between S and $V - S$ with no blue edges, select from the cut-set a non-colored edge with min weight and paint it blue

Red rule: Given a cycle C with no red edges, selected an non-colored edge in C with max weight and paint it red.

Greedy scheme:
Given G, $|V(G)| = n$, apply the red and blue rules until having $n - 1$ blue edges, those form the MST.

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's rule. Uses a priority queue (usually a heap) to store the edges to be added and retrieve the lighter one.
- Kruskal (Serial distributed) Considers every edge and grows a forest by using the blue and red rules to include or discard e. The insight of the algorithm is to consider the edges in order of increasing weight. This makes the complexity of Kruskal's to be dominated by $\Omega(m \lg m)$. At the end the fores becomes a tree. The efficient implementation of the algorithm uses the Union-find data structure.



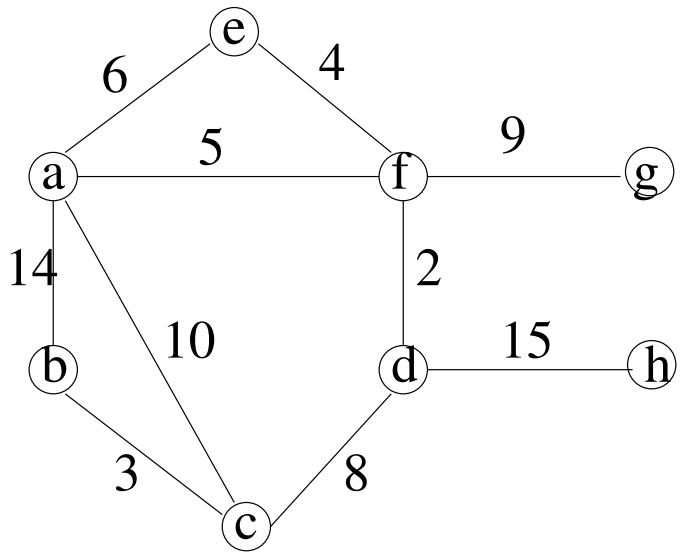
Jarník - Prim greedy algorithm.

V. Jarník, 1936, R. Prim, 1957

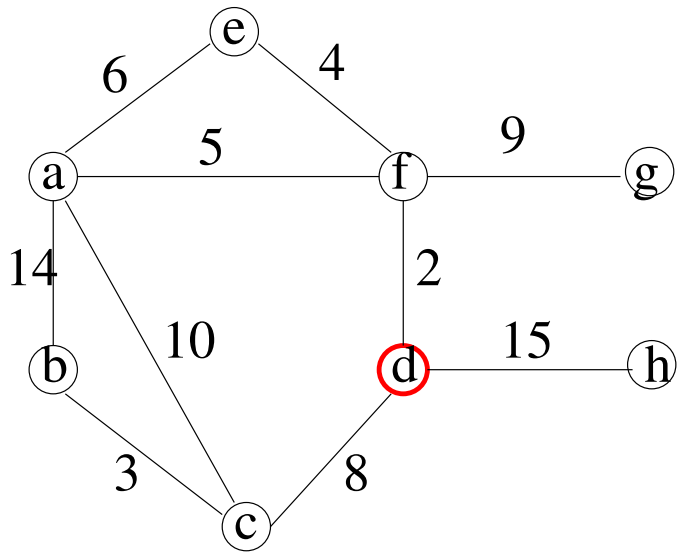
- The algorithmgs keeps a tree T and adds one edge (and one node) to T at each step.
- Initially the tree T has one arbitrary node r, and no edges.
- At each step T is enlarged adding a minimum weight edge in the $C(T) = cut - set(V(T), V - V(T))$.
- Note that an edge e is in the cut-set if e has one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

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

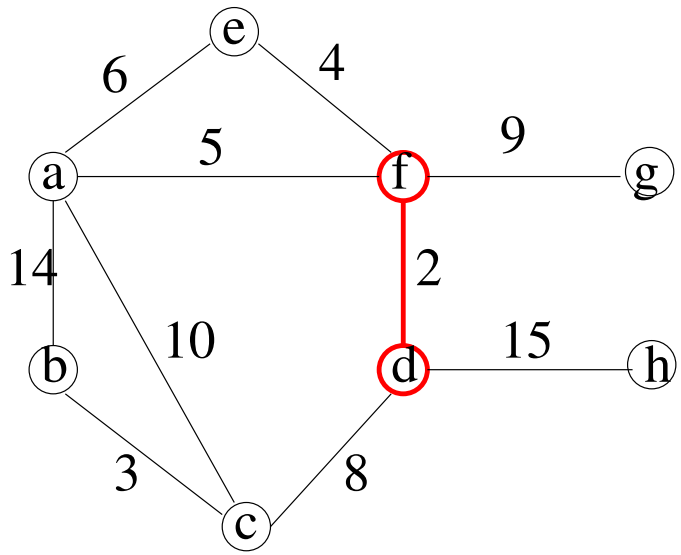
Example.



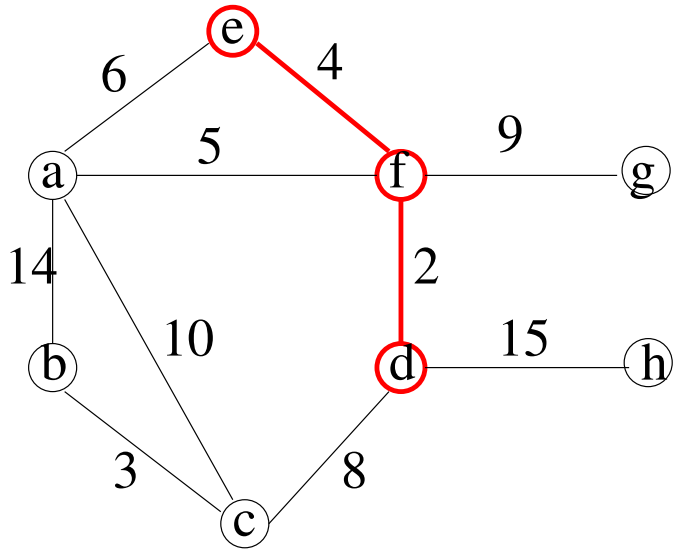
Example.



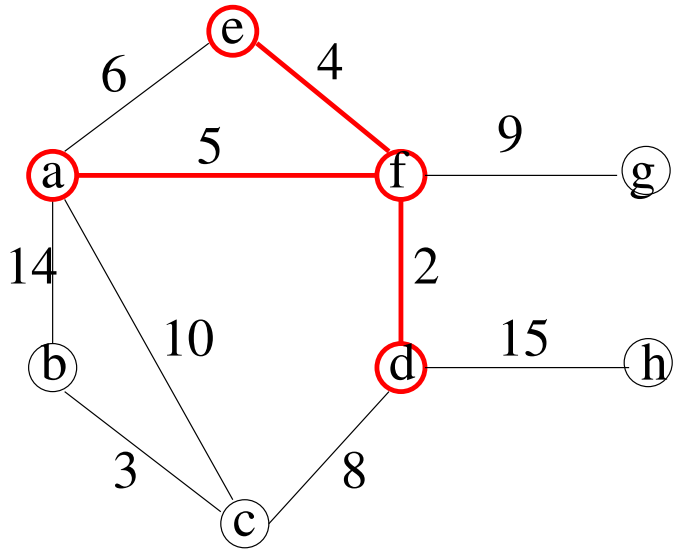
Example.



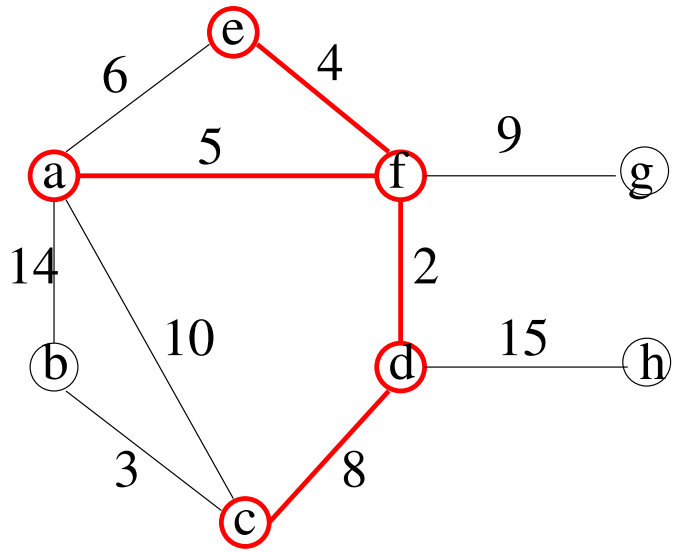
Example.



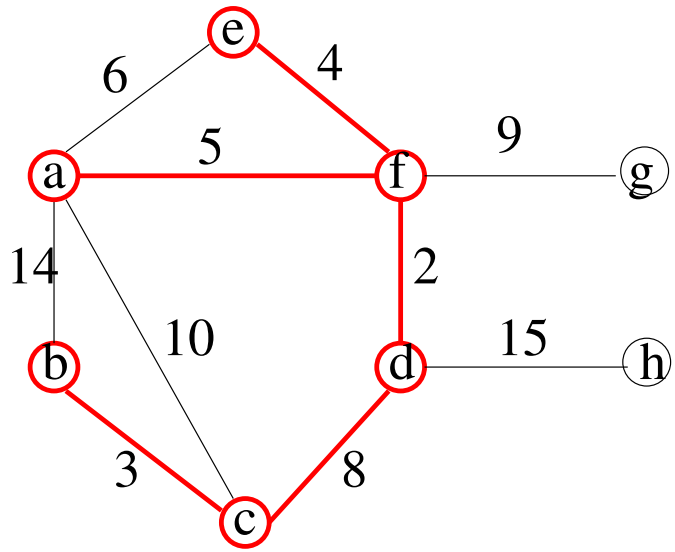
Example.



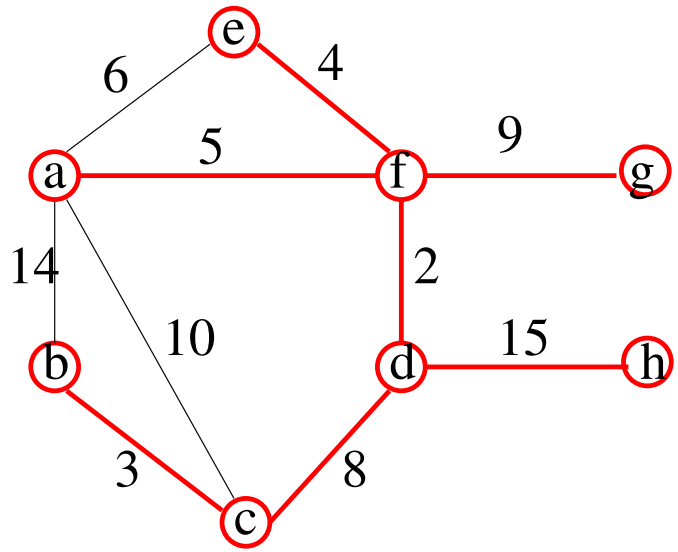
Example.



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)
$T = (\{r\}, \emptyset)$; $Q = \emptyset$; $s = 1$
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**
    add e to T; $++s$
    Insert in Q all edges e from the added vertex to a vertex not in T with key $w(e)$
  end if
end while

Jarník - Prim greedy algorithm: Correctness

- The algorithm discards edge e: Such edge $e = (u, v)$ has $u, v \in V(T)$, so it creates one cycle with the edges in T. Furthermore, e is the edge with highest weight in the cycle. This is 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 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: $T(n) = O(|E| + |V| \lg |V|)$

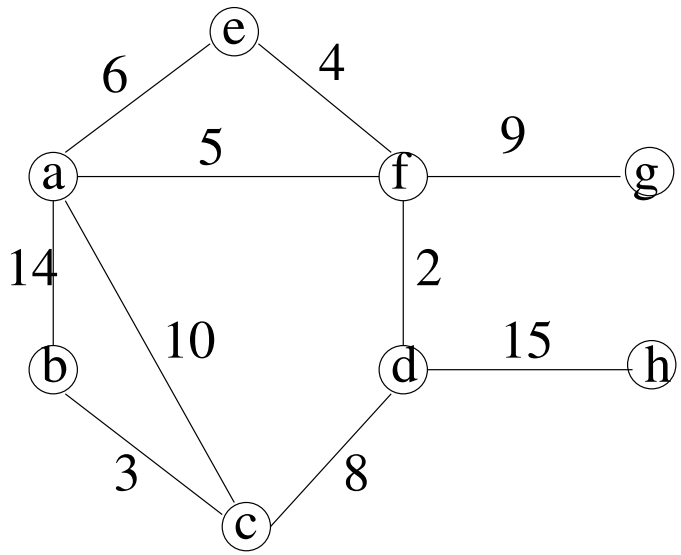
Kruskal's greedy algorithm.

J. Kruskal, 1956

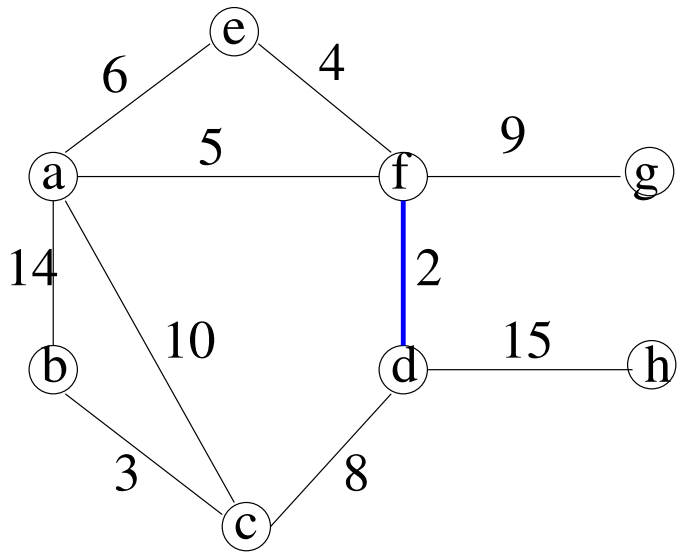
Similar to Jarník - Prim, but chooses minimum weight edges, in some cut, without keeping the graph connected.

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

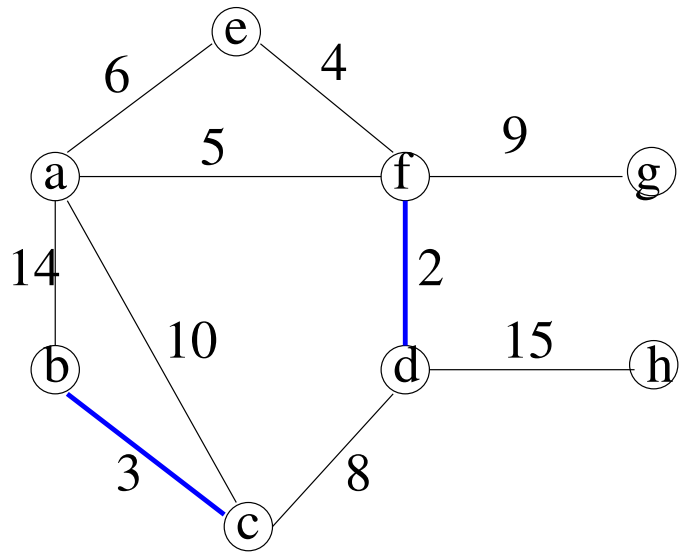
Example.



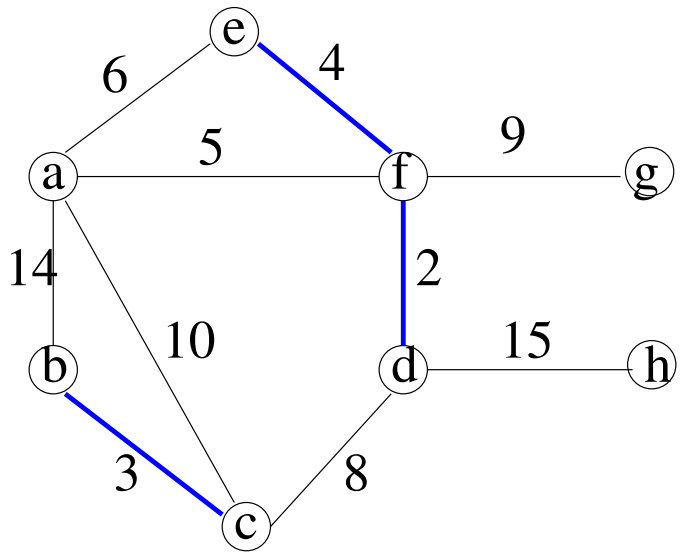
Example.



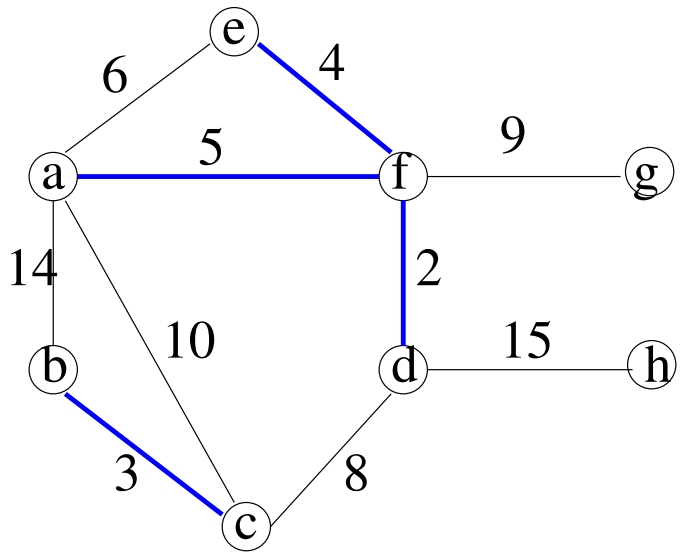
Example.



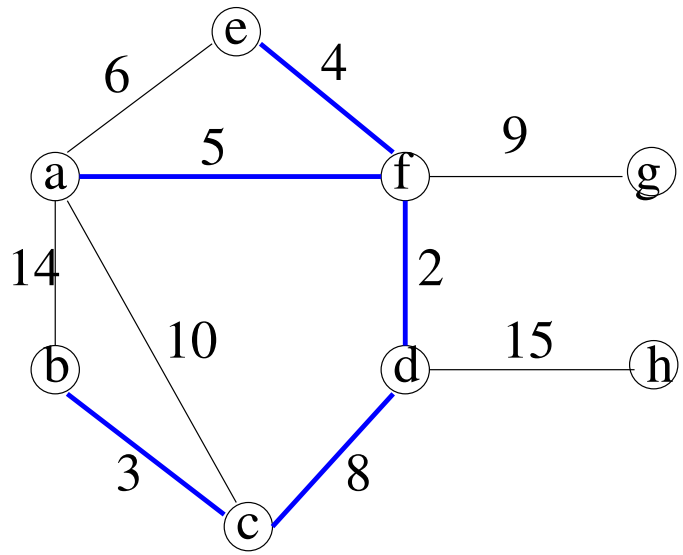
Example.



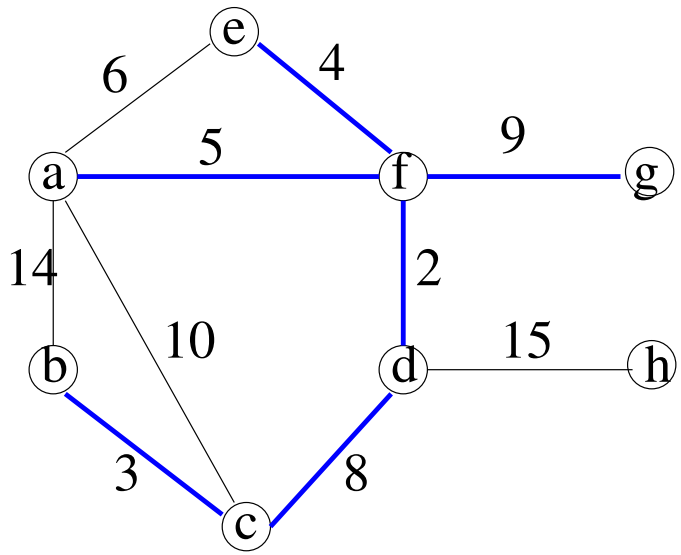
Example.



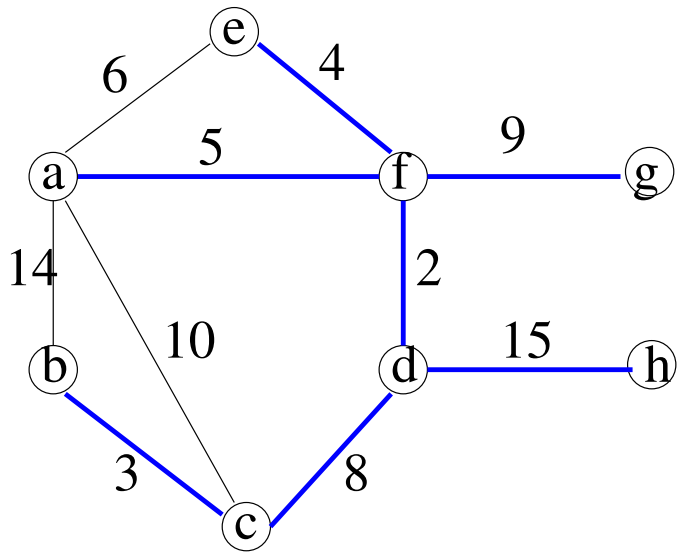
Example.



Example.



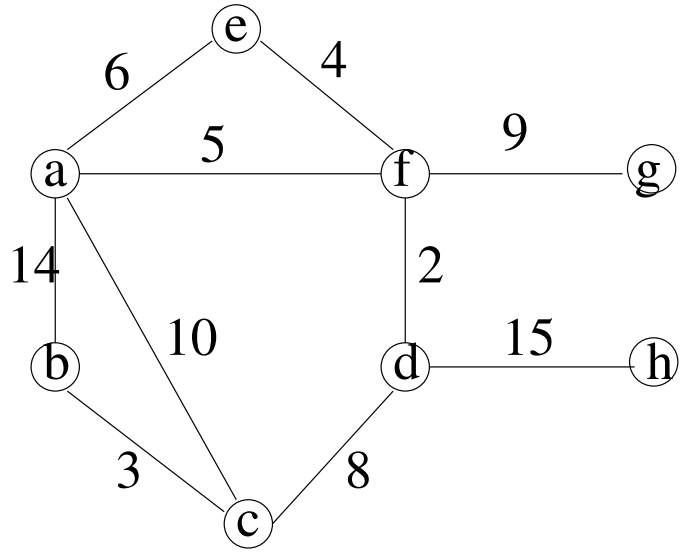
Example.



Kruskal's algorithm: Implementation

- We have an $O(m \lg m)$ from the sorting the edges.
  But as $m \leq n^2$ then $O(m \lg m) = O(m \lg n)$.
- 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.

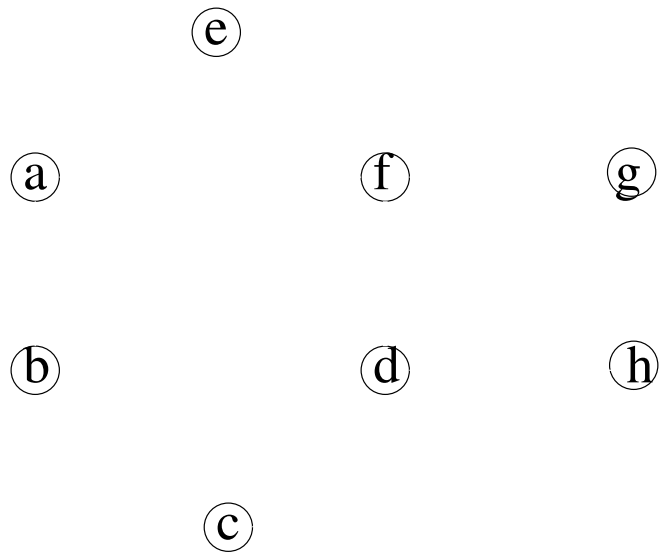
Another view of Kruskal's algorithm



edges sorted by weight

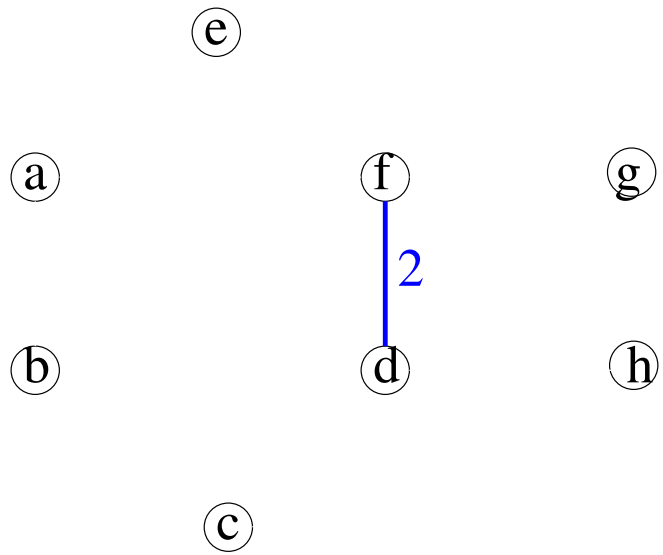
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



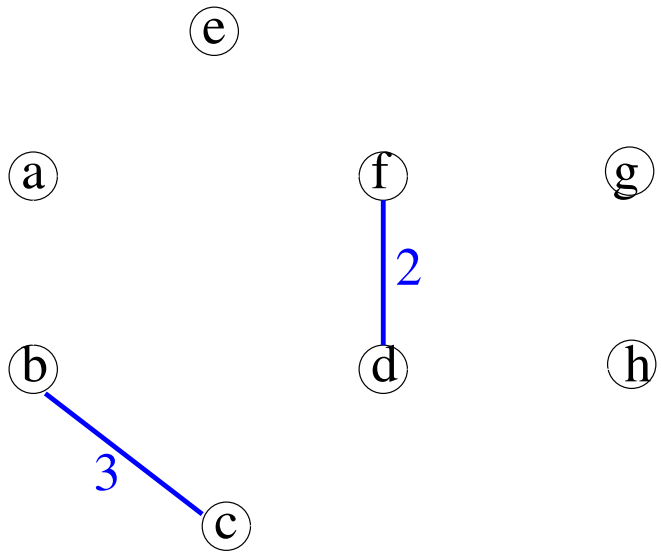
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



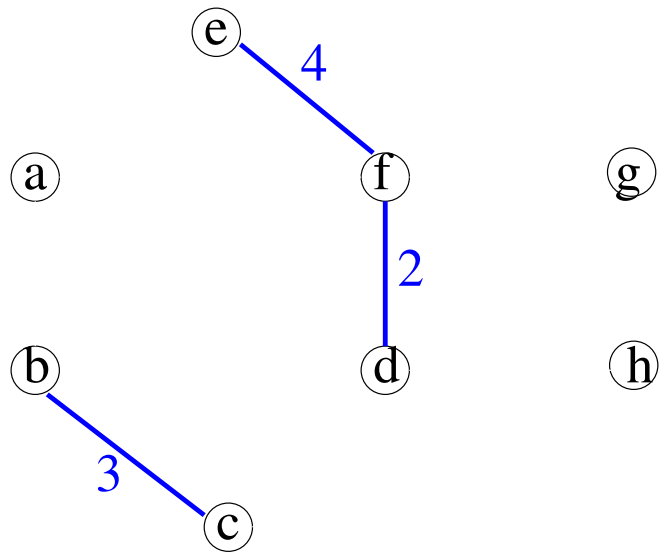
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



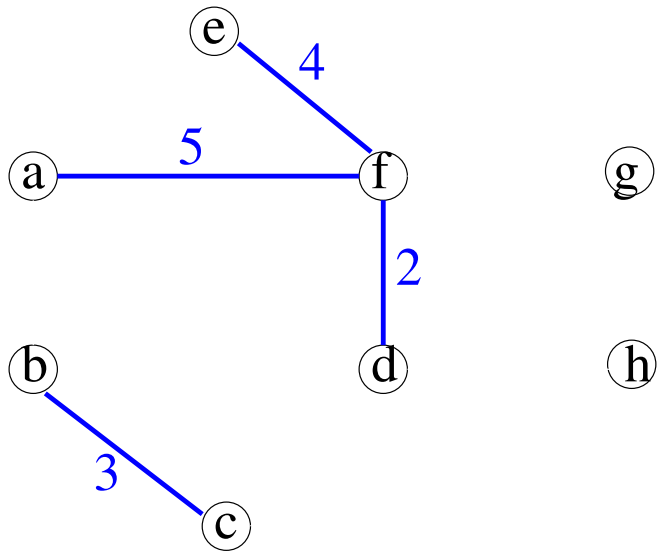
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



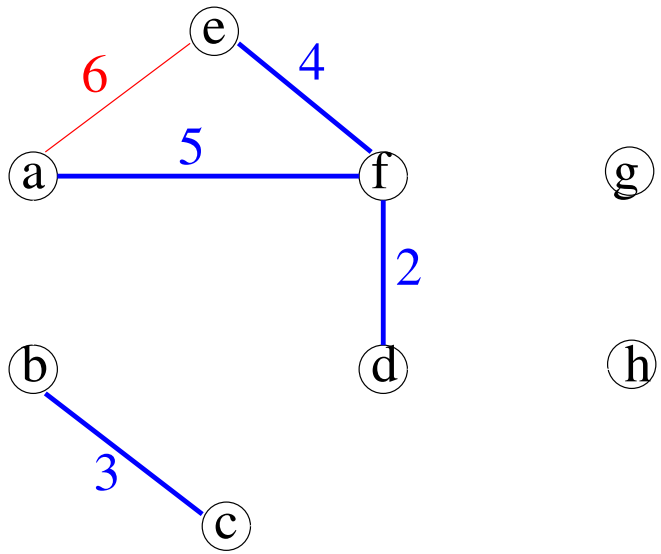
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



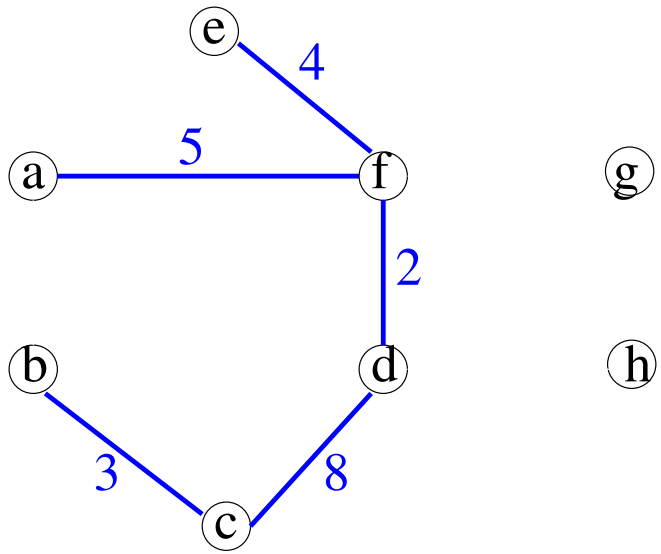
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



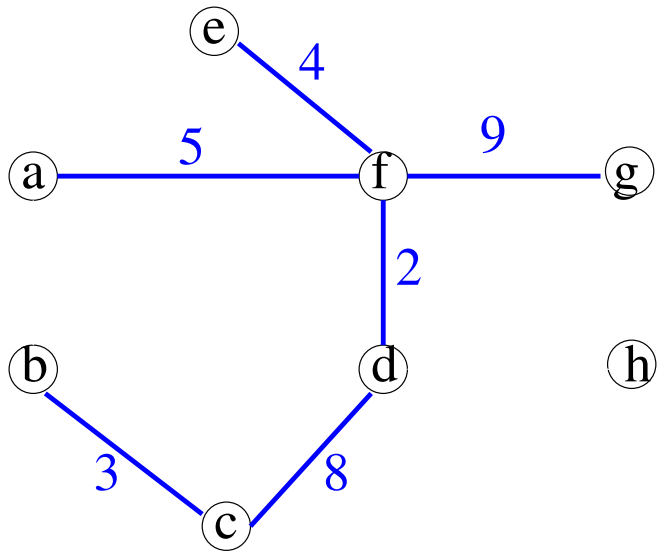
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



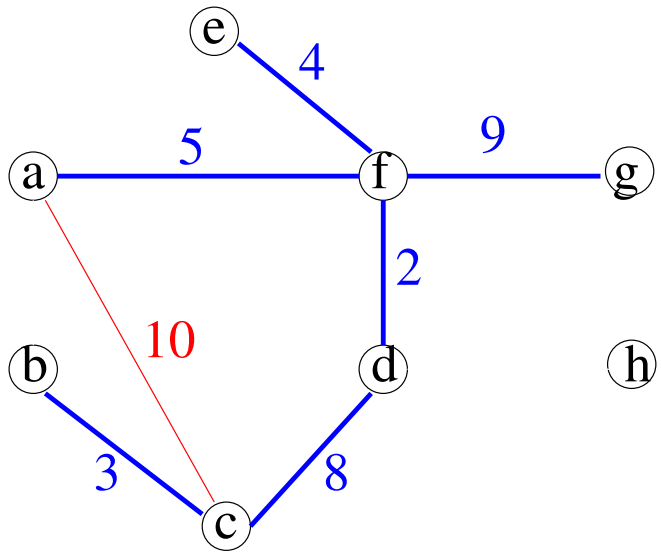
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



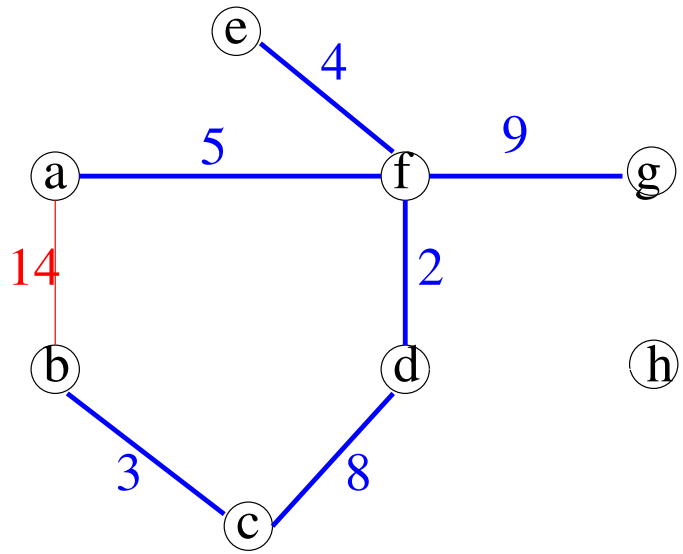
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

Example.



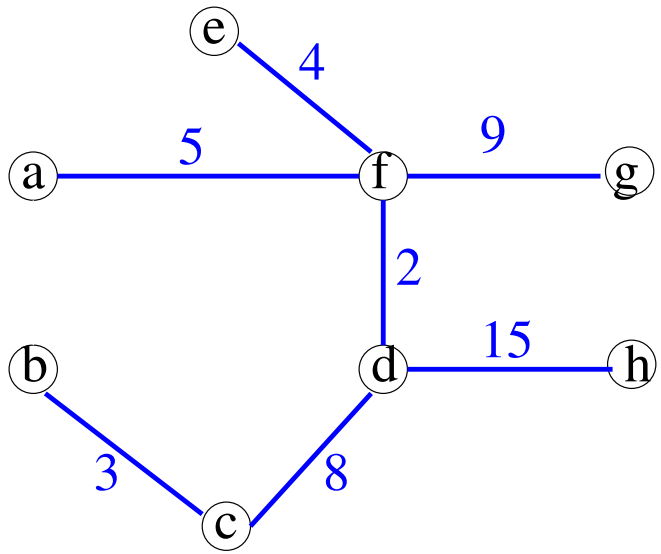
$(f, d, 2), (c, b, 3), (e, f, 4), (a, f, 5), (a, e, 6), (c, d, 8),$
$(f, g, 9), (a, c, 10), (a, b, 14), (d, h, 15)$

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

Notice that Kruskal evolves by building clumps of trees and merging the clumps into larger clumps, taking care (using the red rule) to do not create a cycle.

In forests, the connectivity relation is an equivalence relation, two nodes are connected if there is a path between them (in fact a unique path).

Given an undirected $G = (V, E)$, a forest F on $G = (V, E)$, induces an equivalence relation between the vertices of V: $u \mathcal{R}_F v$ iff there a path between u and v in F:

$\mathcal{R}$ partition the elements of V in equivalence classes, which are connected components without cycles

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 any collection of 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 field in algorithmic; Self organizing data structures.

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.

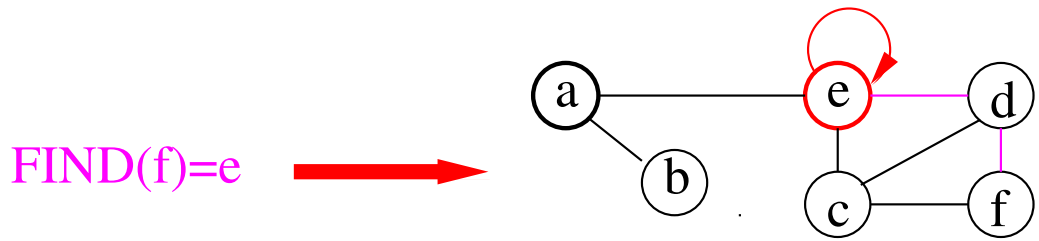
Union-Find

Union-Find is a data structure that 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 can need to do UNION(x, y), for x, y that might not be representatives.
Depending on the implementation this might or might not be allowed.
To study the complexity under different implementations, we consider that

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

Union-Find Data Structure: The basic working

Given a set S of size n, construct a data structure that maintains a collection $\{S_1, \ldots, S_k\}$ of disjoint dynamic sets, each set identified by a *representative*,.

We have n initial elements a set S, we start by applying n times MAKESET to have n single element sets.

After, we want to implement a sequence of m UNION and FIND operations on the initial sets, using the minimum number of steps.

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 $O(n)$.

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, in a UNION keeping the representative of the larger set.
  - MAKESET and FIND takes $O(1)$
  - any sequence of k UNION takes $O(k \log k)$.

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.

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

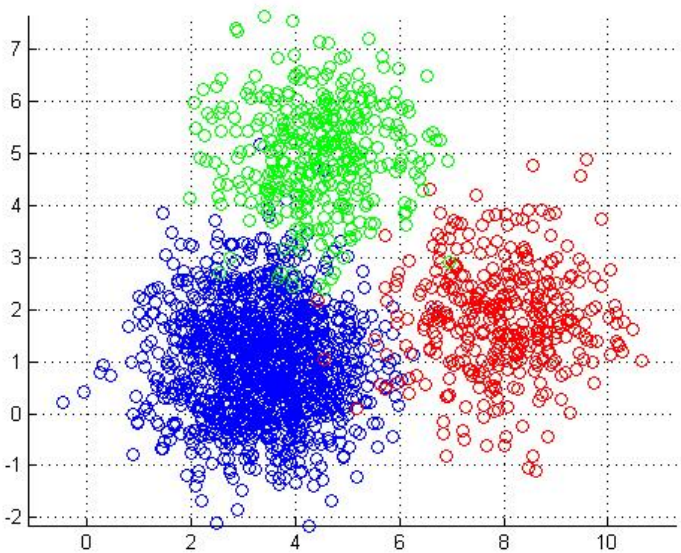
- Kruskal's algorithm for MST.
- Dynamic graph connectivity in very large networks.
- Cycle detection in undirected graphs.
- Random maze generation and exploration.
- Strategies for games: Hex and Go.
- Least common ancestor.
- Compiling equivalence statements.
- Equivalence of finite state automata.

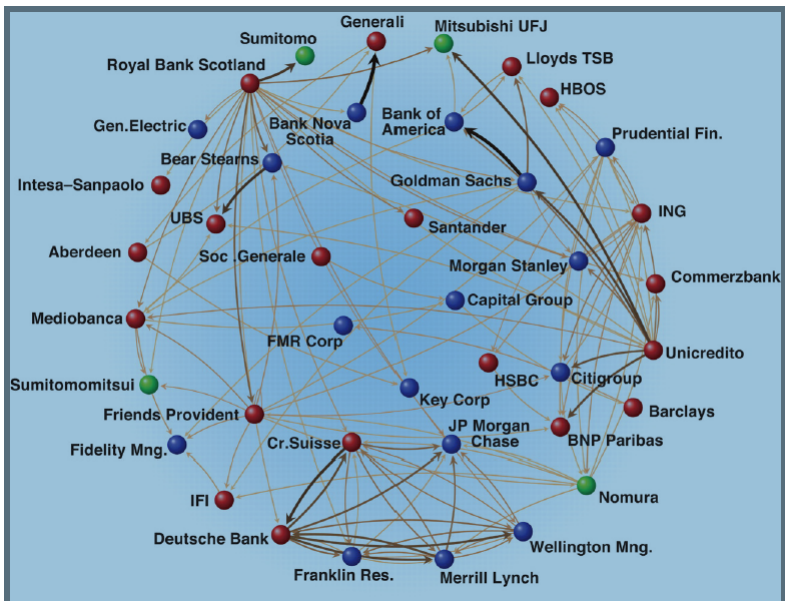
Clustering

- 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 $k > 0$, want to partition X into k disjoint subsets, a k-clusters, such as to optimize some function (depending on d).



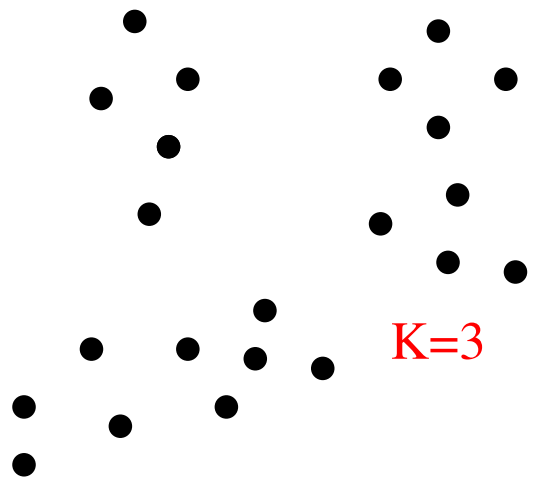


The single-link clustering problem

Let $\mathcal{U}$ be a set of n, for any constant $k > 0$, assume $\{C_i\}$ 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.

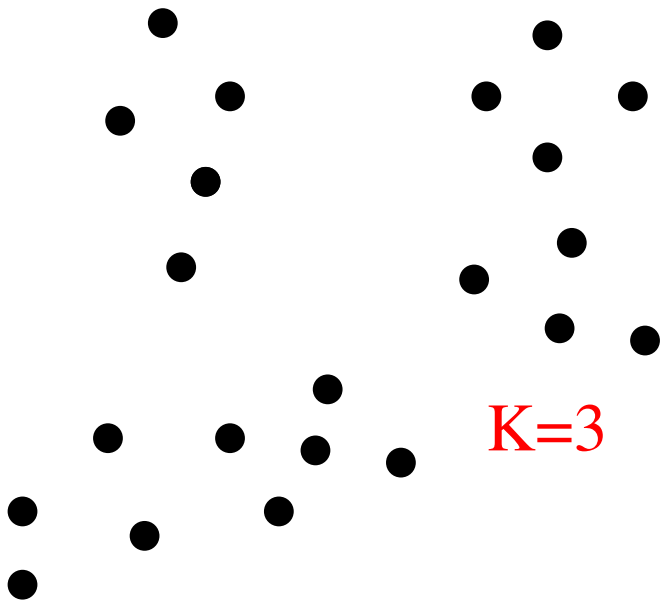
The single-link $k$clustering problem: Given $\mathcal{U} = \{x_1, x_2, \ldots, x_n\}$, a distance function d, and a constant $k > 0$, we want to find the k-clustering of $\mathcal{U}$, which maximizes the spacing s.

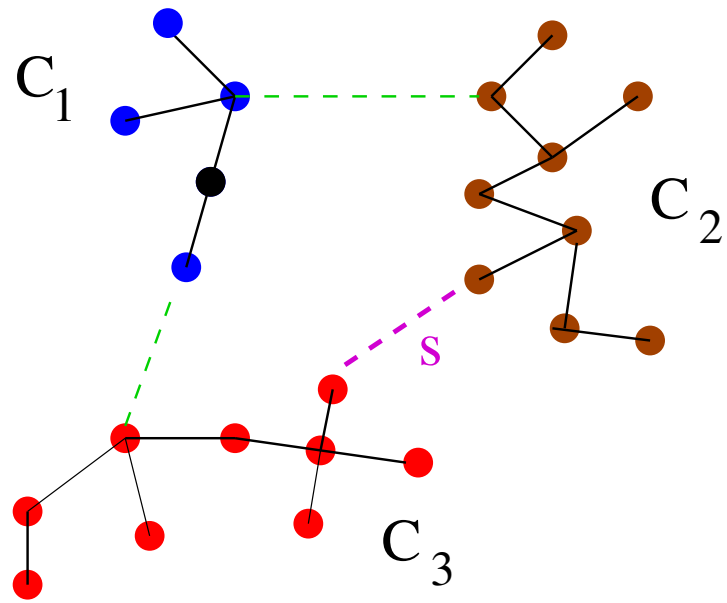
Notice there are exponentially many different k-clustering of $\mathcal{U}$.



Polynomial time algorithm for the single-link k-clustering problem

- Represent $\mathcal{U}$ as vertices of an undirected graph (a click) where the edge (x, y) has weight $d(x, y)$.
- Apply Kruskal's algorithm to find k disconnected MST.
- We get a a clustering $C_1, \ldots, C_k$ and the first non used edge is the spacing s.





Complexity and correctness

Complexity: $O(n^2 \lg n)$

Correctness Let $\mathcal{C} = \{C_1, \ldots, C_k\}$ be the k-cluster produced by the algorithm, and let s be its spacing.
Assume there is another k-cluster $\mathcal{C}' = \{C'_1, \ldots, C'_k\}$ with spacing s' and s.t. $\mathcal{C} \neq \mathcal{C}'$. We must show that $s' \leq s$.
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. $s' \leq d(x', y') \leq s$. $\square$