Algorithmics and Complexity
Lecture 3/7 : Minimum Spanning Tree

CentraleSupélec – Gif

ST2 – Gif

Plan

Plan

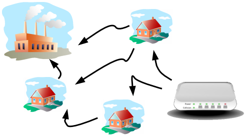
Internet access provider

Problem

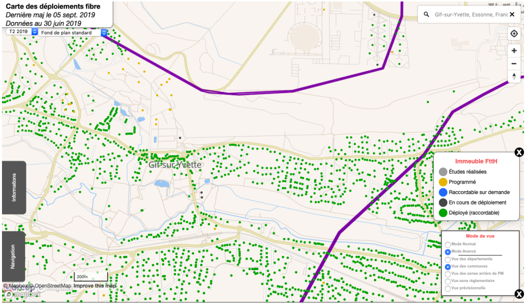
Connect n sites with optical fiber.

$\rightarrow$ We know the cost for threading a cable between two sites v_i and v_j, $i, j \in 1, 2, \ldots, n$

(it is not always possible to connect two sites directly).



Gif Optical Fiber Network - source : ARCEP



Internet access provider

Problem

Connect n sites with optical fiber.

$\rightarrow$ We know the cost for threading a cable between two sites v_i and v_j, $i, j \in 1, 2, \ldots, n$

(it is not always possible to connect two sites directly).

Goal

Install a new optical network infrastructure at the lowest cost.

Internet access provider

Problem

Connect n sites with optical fiber.

$\rightarrow$ We know the cost for threading a cable between two sites v_i and v_j, $i, j \in 1, 2, \ldots, n$

(it is not always possible to connect two sites directly).

Goal

Install a new optical network infrastructure at the lowest cost.

Type of the problem

This is an optimization problem.

Problem modelling

Network → graph

- Sites → vertices of an undirected graph
- Possible connection → edge between two vertices
- Threading cost → weight of the edge

Problem modelling

Network → graph

- Sites → vertices of an undirected graph
- Possible connection → edge between two vertices
- Threading cost → weight of the edge

Instance of the problem

An undirected graph G with n vertices which is connected and has positive weights.

We consider the weight function $\omega : E \rightarrow]0, +\infty[$

Problem modelling

Sub-graph

Let $G = (V, E)$ be a weighted graph. A sub-graph of G is a tuple (V', E') such that:

- $V' \subseteq V$ (no additional nodes)
- $E' \subseteq E$ (no additional edges)
- $E' \subseteq V' \times V'$ (edges between nodes from V')

Problem modelling

Sub-graph

Let $G = (V, E)$ be a weighted graph. A sub-graph of G is a tuple (V', E') such that:

- $V' \subseteq V$ (no additional nodes)
- $E' \subseteq E$ (no additional edges)
- $E' \subseteq V' \times V'$ (edges between nodes from V')

Objective?

Find a sub-graph $T = (V', E')$ of G :

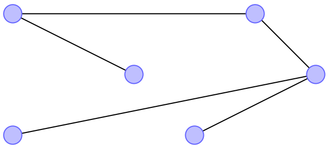
- It has all vertices of G (without all edges)
- It is connected
- The value of the sum of edge weights $\sum_{e \in E'} w(e)$ is minimal.

Some definitions

Tree (recall)

An undirected graph that is connected and acyclic is called a tree.

Example:



Definition: Forest

A forest is a finite set of trees.

Property

Theorem

T, the search sub-graph, is a tree.

Property

Theorem

T, the search sub-graph, is a tree.

proof

Reminder: a tree is a connected and acyclic graph.

T is a connected undirected graph by definition

Property

Theorem

T, the search sub-graph, is a tree.

proof

Reminder: a tree is a connected and acyclic graph.

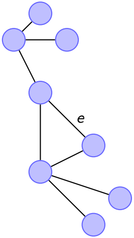
T is a connected undirected graph by definition

Acyclic: proof by contradiction

Assume that T has a cycle. Remove one edge e (anyone) from the cycle and consider T_1 the sub-graph of T that does not contain e.

$\sum_{a \in E_{T_1}} w(a) = \sum_{a \in E_T} w(a) - w(e)$

Thus T_1 has a lower cost than T (supposed to be the minimum).



Minimum Spanning Tree (MST)

Data

- $G = (V, E)$ an undirected and connected graph with $|V| = n$
- $\omega : E \longrightarrow \mathbb{R}_+^*$ the weight function

Goal

Find a tree $T = (V, E_T), E_T \subseteq E$ such that T is a spanning tree and $\sum_{e \in E_T} \omega(e)$ is minimum.

Minimum Spanning Tree (MST)

Data

- $G = (V, E)$ an undirected and connected graph with $|V| = n$
- $\omega : E \longrightarrow \mathbb{R}_+^*$ the weight function

Goal

Find a tree $T = (V, E_T), E_T \subseteq E$ such that T is a spanning tree and $\sum_{e \in E_T} \omega(e)$ is minimum.

Brute-force approach : enumerate all possible T and compare their total weight

Complexity : $\mathcal{O}\binom{|E|}{|V|-1}$. *Undesirable*

Plan

Greedy Algorithms

Definition

A greedy algorithm is an algorithm that:

- Builds a solution one step at a time;

  e.g. Lego construction or line-by-line multiplication

- Makes a choice at each step to optimize a local criteria;

  i.e. evaluation of the current situation

- Never revokes a previous choice.

Greedy Algorithms

Definition

A greedy algorithm is an algorithm that:

- Builds a solution one step at a time;

  e.g. Lego construction or line-by-line multiplication

- Makes a choice at each step to optimize a local criteria;

  i.e. evaluation of the current situation

- Never revokes a previous choice.

- The best solution is chosen locally: no guarantee on finding the global optimum
- No going back: the algorithm goes directly toward a solution.
- Example ?

Greedy Algorithms

Definition

A greedy algorithm is an algorithm that:

- Builds a solution one step at a time;
  e.g. Lego construction or line-by-line multiplication
- Makes a choice at each step to optimize a local criteria;
  i.e. evaluation of the current situation
- Never revokes a previous choice.

- The best solution is chosen locally: no guarantee on finding the global optimum
- No going back: the algorithm goes directly toward a solution.
- Example : Shortest path
  - local criteria: node of the frontier with minimum distance

Minimum spanning tree : generic approach

Goal

Given $G = (V, E)$ and $w : E \longrightarrow \mathbb{R}_+^*$

➜ We must build, step by step, a subset of E.

General principle

- We start with $E_T = \emptyset$
- At each step, a new edge (u, v) is chosen such that $E_T \cup (u, v)$ is always a sub-set of a minimum spanning tree of G.

➜ We say that (u, v) is a safe edge for E_T

This greedy algorithm reaches a global optimum...

Minimum spanning tree : generic approach

General algorithm

```
def MST(V,E):
  E_T = []
  while !isSolution(E_T,V,E):
    e = safeEdge(E_T,V,E)
    E_T.append(e)
  return E_T
```

Minimum spanning tree : generic approach

General algorithm

```
def MST(V,E):
  E_T = []
  while !isSolution(E_T,V,E):
    e = safeEdge(E_T,V,E)
    E_T.append(e)
  return E_T
```

How can we find a safe edge?

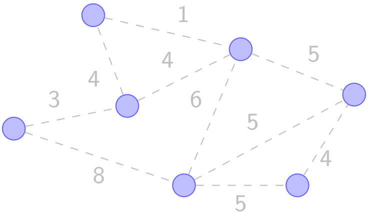
We shall discover it soon. . .

Let us first discover two algorithms that adopt this schema.

Kruskal method (1956)

Overview

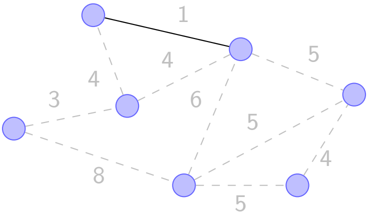
- **Initialization:** a graph T with all vertices of G but without any edges
- **Iteration:** add to T an edge with the minimum weight without creating a cycle
- **Stop:** after adding $n-1$ edges



Kruskal method (1956)

Overview

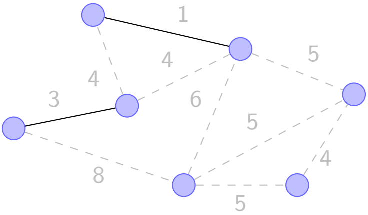
- **Initialization:** a graph T with all vertices of G but without any edges
- **Iteration:** add to T an edge with the minimum weight without creating a cycle
- **Stop:** after adding $n-1$ edges



Kruskal method (1956)

Overview

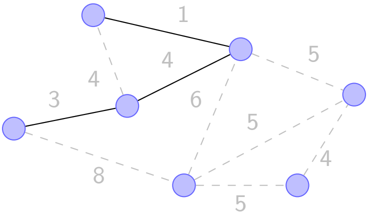
- **Initialization:** a graph T with all vertices of G but without any edges
- **Iteration:** add to T an edge with the minimum weight without creating a cycle
- **Stop:** after adding $n-1$ edges



Kruskal method (1956)

Overview

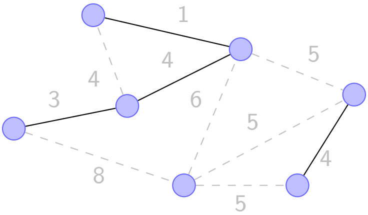
- **Initialization:** a graph T with all vertices of G but without any edges
- **Iteration:** add to T an edge with the minimum weight without creating a cycle
- **Stop:** after adding $n-1$ edges



Kruskal method (1956)

Overview

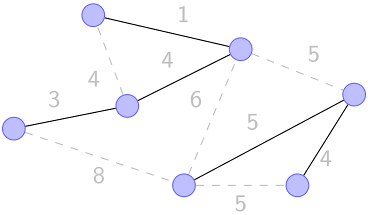
- **Initialization:** a graph T with all vertices of G but without any edges
- **Iteration:** add to T an edge with the minimum weight without creating a cycle
- **Stop:** after adding $n-1$ edges



Kruskal method (1956)

Overview

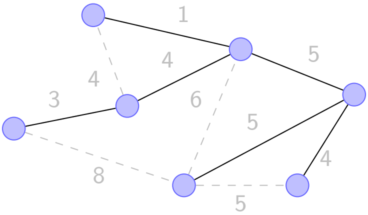
- **Initialization:** a graph T with all vertices of G but without any edges
- **Iteration:** add to T an edge with the minimum weight without creating a cycle
- **Stop:** after adding $n-1$ edges



Kruskal method (1956)

Overview

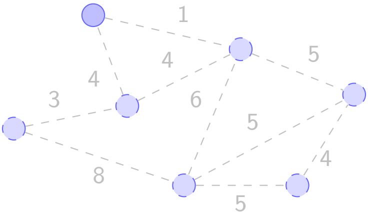
- **Initialization:** a graph T with all vertices of G but without any edges
- **Iteration:** add to T an edge with the minimum weight without creating a cycle
- **Stop:** after adding $n-1$ edges



Prim method (1957)

Overview

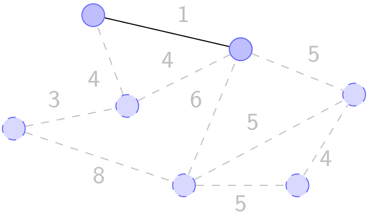
- **Initialization:** a tree T that contains one vertex from G (anyone)
- **Iteration:** add to T an edge with the minimum weight to connect a new node (*i.e.* not already in T)
- **Stop:** after adding $n-1$ edges



Prim method (1957)

Overview

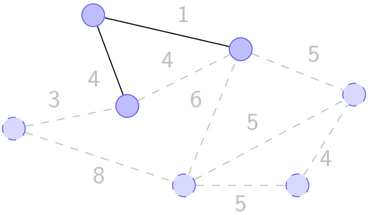
- **Initialization:** a tree T that contains one vertex from G (anyone)
- **Iteration:** add to T an edge with the minimum weight to connect a new node (*i.e.* not already in T)
- **Stop:** after adding $n - 1$ edges



Prim method (1957)

Overview

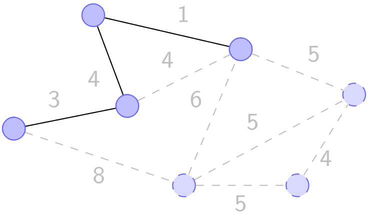
- **Initialization:** a tree T that contains one vertex from G (anyone)
- **Iteration:** add to T an edge with the minimum weight to connect a new node (*i.e.* not already in T)
- **Stop:** after adding $n-1$ edges



Prim method (1957)

Overview

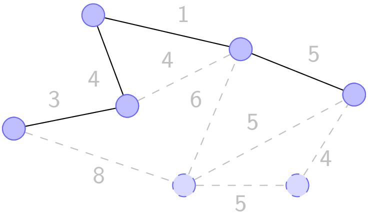
- **Initialization:** a tree T that contains one vertex from G (anyone)
- **Iteration:** add to T an edge with the minimum weight to connect a new node (*i.e.* not already in T)
- **Stop:** after adding $n-1$ edges



Prim method (1957)

Overview

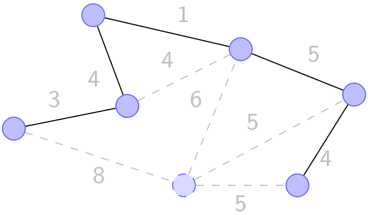
- **Initialization:** a tree T that contains one vertex from G (anyone)
- **Iteration:** add to T an edge with the minimum weight to connect a new node (*i.e.* not already in T)
- **Stop:** after adding $n-1$ edges



Prim method (1957)

Overview

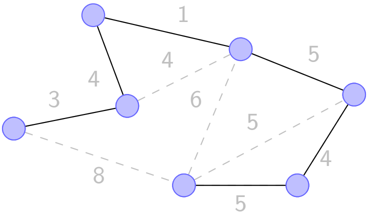
- **Initialization:** a tree T that contains one vertex from G (anyone)
- **Iteration:** add to T an edge with the minimum weight to connect a new node (*i.e.* not already in T)
- **Stop:** after adding $n-1$ edges



Prim method (1957)

Overview

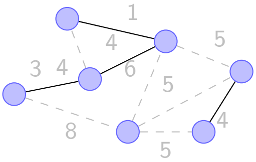
- **Initialization:** a tree T that contains one vertex from G (anyone)
- **Iteration:** add to T an edge with the minimum weight to connect a new node (*i.e.* not already in T)
- **Stop:** after adding $n-1$ edges

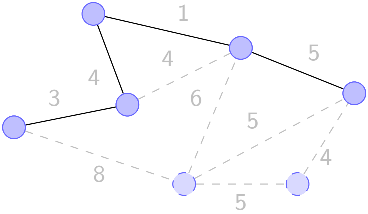


Remarks

Comparison of both approaches

- Kruskal maintains full coverage (forest) and builds up connexity
- Prim maintains T connected (sub-tree) and makes it a covering tree
- Both ensure the absence of cycle





Remarks

Comparison of both approaches

- Kruskal maintains full coverage (forest) and builds up connexity
- Prim maintains T connected (sub-tree) and makes it a covering tree
- Both ensure the absence of cycle
- Both stop $n-1$ edges. Why?

Remarks

Comparison of both approaches

- Kruskal maintains full coverage (forest) and builds up connexity
- Prim maintains T connected (sub-tree) and makes it a covering tree
- Both ensure the absence of cycle
- Both stop $n-1$ edges. Why?

Theorem

If T is a graph of n vertices, then the three statements are equivalent:

- T is a tree : acyclic and connected
- T is connected and has $n-1$ edges
- T is acyclic and has $n-1$ edges

Will you be able to prove this?

Prim algorithm details

Principle

Maintain a structure `nextnodes` containing the remaining nodes while updating their distance to the current tree:

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Prim algorithm details

Principle

Maintain a structure `nextnodes` containing the remaining nodes while updating their distance to the current tree:

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Iteration

1. Extract the vertex from `nextnodes` of minimum distance to the current tree T
2. Add it to the tree T with the corresponding minimal edge
3. Update distances of its neighbors

Prim algorithm details

Principle

Maintain a structure `nextnodes` containing the remaining nodes while updating their distance to the current tree:

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Iteration

1. Extract the vertex from `nextnodes` of minimum distance to the current tree T
2. Add it to the tree T with the corresponding minimal edge
3. Update distances of its neighbors

➜ It is very similar to SP (BFS)!

Prim algorithm details

Principle

Maintain a structure `nextnodes` containing the remaining nodes while updating their distance to the current tree:

$$dist(x, T) = min\{\omega((x, u)) \mid u \in T\}$$

Iteration

1. Extract the vertex from `nextnodes` of minimum distance to the current tree T
2. Add it to the tree T with the corresponding minimal edge
3. Update distances of its neighbors

➜ It is very similar to SP (BFS)!

At the end

`nextnodes` is empty and T is a minimum spanning tree

Prim algorithm

```
def Prim_MST(graph,s):
    nextnodes = nodes(graph)
    parent = {}
    dist = {}
    dist[s] = 0
    For the first step, s will always be selected

    while len(nextnodes)>0:
        x = extract_min_dist(nextnodes,dist)

        update the neighbors:
        for y in neighbors(graph, x):
            new_dist = distance(graph,x,y)
            if (y in nextnodes) and \
                    (y not in dist or new_dist < dist[y]):
                dist[y] = new_dist
                parent[y] = x

    return parent
```

SP vs Prim

Prim

```
def Prim_MST(graph,s):
  nextnodes = nodes(graph)
  parent = {}
  dist = {}; dist[s] = 0
  while len(nextnodes)>0:
    x = extract_min_dist(nextnodes,dist)
    for y in neighbors(graph, x):


      new_dist = distance(graph,x,y)

      if (y in nextnodes) and \
           (y not in dist or \
           new_dist < dist[y]):
        dist[y] = new_dist
        parent[y] = x
  return parent
```

SP

```
def shortest_path(graph,s):
  frontier = [s]
  parent = {}; parent[s] = None
  dist = {}; dist[s] = 0
  while len(frontier)>0:
    x = extract_min_dist(frontier,dist)
    for y in neighbors(graph, x):
      if y not in parent:
        frontier.append(y)
      new_dist = dist[x] + \
                 distance(graph,x,y)

      if y not in dist or \
           dist[y] > new_dist:
        dist[y] = new_dist
        parent[y] = x
  return parent
```

SP vs Prim

Prim

```
def Prim_MST(graph,s):
  nextnodes = nodes(graph)
  parent = {}
  dist = {}; dist[s] = 0
  while len(nextnodes)>0:
    x = extract_min_dist(nextnodes,dist)
    for y in neighbors(graph, x):


      new_dist = distance(graph,x,y)

      if (y in nextnodes) and \
           (y not in dist or \
           new_dist < dist[y]):
        dist[y] = new_dist
        parent[y] = x
  return parent
```

SP

```
def shortest_path(graph,s):
  frontier = [s]
  parent = {}; parent[s] = None
  dist = {}; dist[s] = 0
  while len(frontier)>0:
    x = extract_min_dist(frontier,dist)
    for y in neighbors(graph, x):
      if y not in parent:
        frontier.append(y)
      new_dist = dist[x] + \
                 distance(graph,x,y)

      if y not in dist or \
           dist[y] > new_dist:
        dist[y] = new_dist
        parent[y] = x
  return parent
```

The difference comes from the storage in the queue

- BFS: elements ordered by number of edges from s
- SP: elements ordered by distance to s
- Prim: elements ordered by distance to the tree under construction

Kruskal algorithm details

At the beginning

The forest F (set of trees) is composed of n isolated vertices (n singleton trees).

Kruskal algorithm details

At the beginning

The forest F (set of trees) is composed of n isolated vertices (n singleton trees).

Iteration

Merge two neighbouring trees, connected by a new edge (of minimal weight) that is added to T.

Invariant : F is a forest (without cycle)

Kruskal algorithm details

At the beginning

The forest F (set of trees) is composed of n isolated vertices (n singleton trees).

Iteration

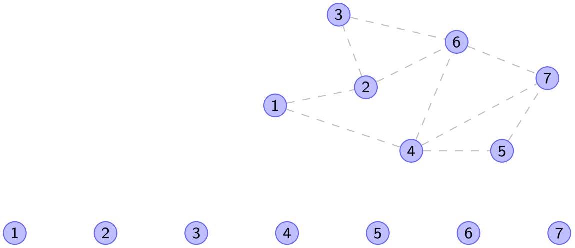
Merge two neighbouring trees, connected by a new edge (of minimal weight) that is added to T.

Invariant : F is a forest (without cycle)

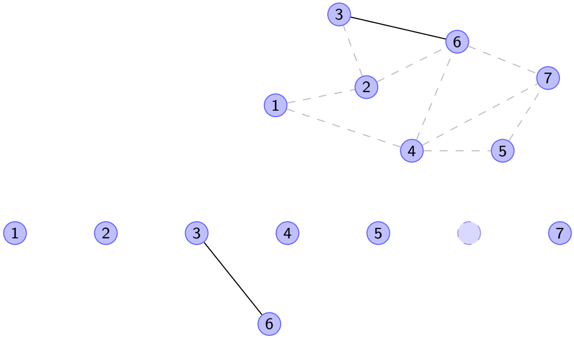
At the end

There is only one remaining tree $T = (V, E_T)$ in F and it is a minimum spanning tree.

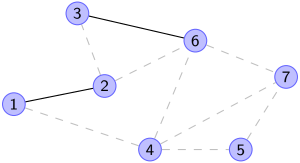
Forest $\Longrightarrow$ Tree

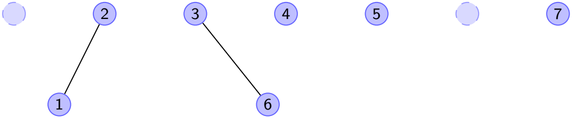


Forest $\Longrightarrow$ Tree

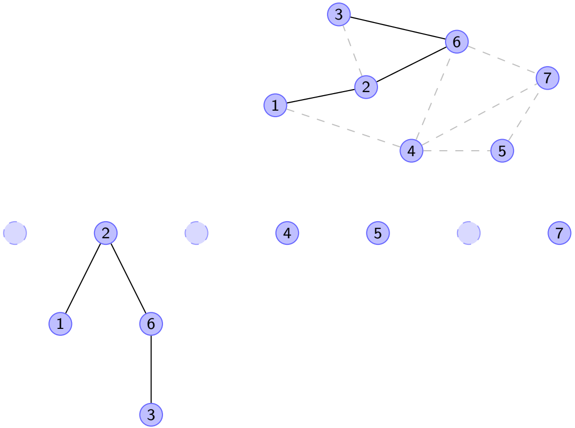


Forest $\Longrightarrow$ Tree

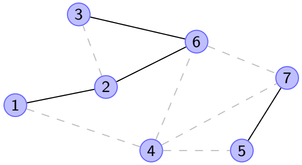


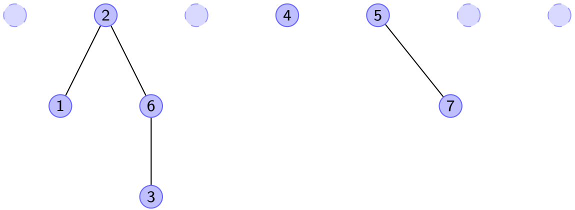


Forest $\Longrightarrow$ Tree



Forest $\Longrightarrow$ Tree





Kruskal algorithm

```
def Kruskal_MST(graph):
    max = len(nodes(graph))-1
    MST = []                    # list of selected edges
    forest = {}                 # set of trees
    for v in nodes(graph):
        tree = create_tree(v)
        forest.add(tree)

    cpt = 0
    for (u, v) in sort_by_weight(edges(graph)):
        if find_tree(u, forest) != find_tree(v, forest):
            MST.append((u, v))
            merge_trees(u, v, forest)

            cpt = cpt + 1
            if cpt == max:
                break

    return MST
```

Complexity

Kruskal Algorithm (more details later)

The Kruskal algorithm is of complexity $\mathcal{O}(|E| \log |E|)$

➜ with $|E| \approx |V|^2$ we obtain a complexity in $\mathcal{O}(|V|^2 \log |V|)$

Complexity

Kruskal Algorithm (more details later)

The Kruskal algorithm is of complexity $\mathcal{O}(|E| \log |E|)$

➜ with $|E| \approx |V|^2$ we obtain a complexity in $\mathcal{O}(|V|^2 \log |V|)$

Prim Algorithm

The complexity of Prim's algorithm is the same as the complexity of SP (Dijkstra) algorithm:

➜ With $|E| \approx |V|$ we get a complexity $\mathcal{O}(|V| \log(|V|))$ binary heap

➜ With $|E| \approx |V|^2$ we get a complexity $\mathcal{O}(|V|^2)$ list

Complexity

Kruskal Algorithm (more details later)

The Kruskal algorithm is of complexity $\mathcal{O}(|E| \log |E|)$

➜ with $|E| \approx |V|^2$ we obtain a complexity in $\mathcal{O}(|V|^2 \log |V|)$

Prim Algorithm

The complexity of Prim's algorithm is the same as the complexity of SP (Dijkstra) algorithm:

➜ With $|E| \approx |V|$ we get a complexity $\mathcal{O}(|V| \log(|V|))$ binary heap

➜ With $|E| \approx |V|^2$ we get a complexity $\mathcal{O}(|V|^2)$ list

Corollary

An optimal solution of the MST problem can be computed in polynomial time!

Optimality

How can we be certain that Prim's and Kruskal's algorithms build a minimum spanning tree?

Reminder

Greedy algorithm to build a MST T from $G = (V, E)$:

- Start with $E' = \emptyset$
- At each step, select a new safe edge (u, v)
  ➜ $E' = E' \cup (u, v)$ must always be a subset of a MST of G
- At the end when $|E'| = |V| - 1$, we obtain a MST $T = (V, E_T = E')$

Optimality

How can we be certain that Prim's and Kruskal's algorithms build a minimum spanning tree?

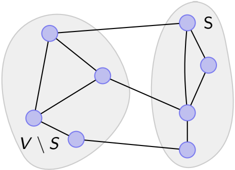
Reminder

Greedy algorithm to build a MST T from $G = (V, E)$:

- Start with $E' = \emptyset$
- At each step, select a new safe edge (u, v)
  ➜ $E' = E' \cup (u, v)$ must always be a subset of a MST of G
- At the end when $|E'| = |V| - 1$, we obtain a MST $T = (V, E_T = E')$

Definition: cut

➜ A cut is a partition of all vertices in two non-empty sets S and $(V \backslash S)$ (disjoint)



Property: safe edge and cut

Definitions

Let us consider $S \subset V$ and $E' \subseteq E$. We say that:

- an edge crosses the cut $(S, V \backslash S)$ if one of its endpoints is in S and the other in $(V \backslash S)$
- the cut $(S, V \backslash S)$ respects E' if no edge in E' crosses the cut

Property: safe edge and cut

Definitions

Let us consider $S \subset V$ and $E' \subseteq E$. We say that:

- an edge crosses the cut $(S, V \backslash S)$ if one of its endpoints is in S and the other in $(V \backslash S)$
- the cut $(S, V \backslash S)$ respects E' if no edge in E' crosses the cut

Property of the greedy algorithm

- E' a subset of a MST T of G $\quad E' \subseteq E_T \subseteq E$
- $(S, V \backslash S)$ a cut that respects E'
- (u, v) an edge of minimal weight which crosses the cut $(S, V \backslash S)$

➜ then (u, v) is a safe edge for E'.

i.e. $E' \cup (u, v)$ will always be a subset of a MST of G

Property: safe edge and cut

Definitions

Let us consider $S \subset V$ and $E' \subseteq E$. We say that:

- an edge crosses the cut $(S, V \backslash S)$ if one of its endpoints is in S and the other in $(V \backslash S)$
- the cut $(S, V \backslash S)$ respects E' if no edge in E' crosses the cut

Property of the greedy algorithm

- E' a subset of a MST T of G — $E' \subseteq E_T \subseteq E$
- $(S, V \backslash S)$ a cut that respects E'
- (u, v) an edge of minimal weight which crosses the cut $(S, V \backslash S)$

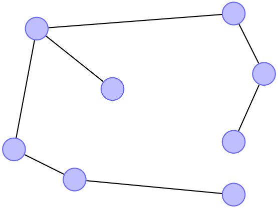
➜ then (u, v) is a safe edge for E'.

i.e. $E' \cup (u, v)$ will always be a subset of a MST of G

That is exactly what Kruskal and Prim do...

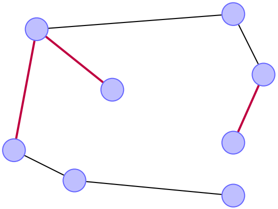
Proof

- Let T be a MST of $G = (V, E)$



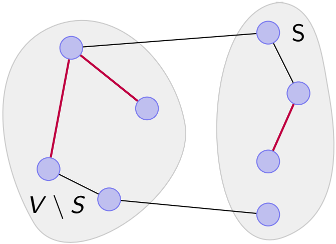
Proof

- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$



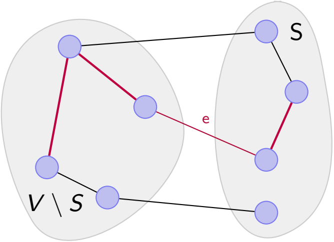
Proof

- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$
- Let $(S, V \backslash S)$, be a cut that respects E'



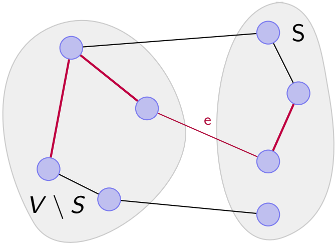
Proof

- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$
- Let $(S, V \backslash S)$, be a cut that respects E'
- Let e be the minimum weight edge that crosses the cut: suppose that T does not contain e



Proof

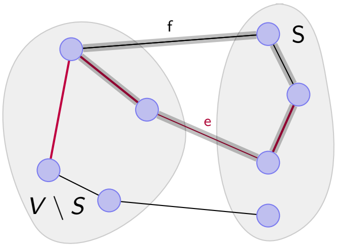
- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$
- Let $(S, V \backslash S)$, be a cut that respects E'
- Let e be the minimum weight edge that crosses the cut: suppose that T does not contain e



We will show that it is possible to build a tree T', including $E' \cup \{e\}$

Proof

- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$
- Let $(S, V \backslash S)$, be a cut that respects E'
- Let e be the minimum weight edge that crosses the cut: suppose that T does not contain e

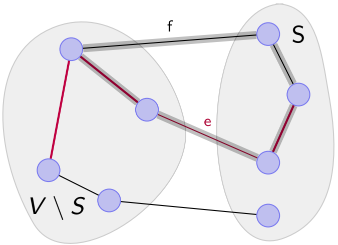


> We will show that it is possible to build a tree T', including $E' \cup \{e\}$

- Adding e to T implies a cycle (denoted C).
- Let $f \neq e$ be an edge of C between S and $V - S$.

Proof

- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$
- Let $(S, V \backslash S)$, be a cut that respects E'
- Let e be the minimum weight edge that crosses the cut: suppose that T does not contain e



We will show that it is possible to build a tree T', including $E' \cup \{e\}$

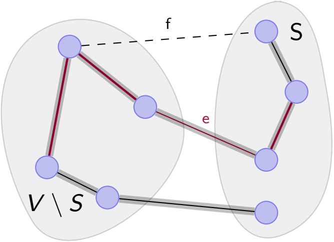
- Adding e to T implies a cycle (denoted C).
- Let $f \neq e$ be an edge of C between S and $V - S$.

➜ By definition of e, we have $\omega(e) \leq \omega(f)$

➜ The cut respects E', thus f is not in E'.

Proof

- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$
- Let $(S, V \backslash S)$, be a cut that respects E'
- Let e be the minimum weight edge that crosses the cut: suppose that T does not contain e



We will show that it is possible to build a tree T', including $E' \cup \{e\}$

- Adding e to T implies a cycle (denoted C).
- Let $f \neq e$ be an edge of C between S and $V - S$.

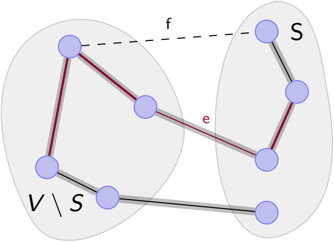
➜ By definition of e, we have $\omega(e) \leq \omega(f)$

➜ The cut respects E', thus f is not in E'.

- Let T' be the tree passing through e instead of f $E_{T'} = (E_T \backslash \{f\}) \cup \{e\}$.
  - ➜ T' is covering
  - ➜ T' is minimal $\Omega(T') = \Omega(T) - \omega(f) + \omega(e) \leq \Omega(T)$
  - ➜ T' includes $E' \cup \{e\}$ $E' \cup \{e\} \subseteq E_{T'}$

Proof

- Let T be a MST of $G = (V, E)$, including $E' \subseteq E_T \subseteq E$
- Let $(S, V \backslash S)$, be a cut that respects E'
- Let e be the minimum weight edge that crosses the cut: suppose that T does not contain e



We will show that it is possible to build a tree T', including $E' \cup \{e\}$

- Adding e to T implies a cycle (denoted C).
- Let $f \neq e$ be an edge of C between S and $V - S$.

➜ By definition of e, we have $\omega(e) \leq \omega(f)$

➜ The cut respects E', thus f is not in E'.

- Let T' be the tree passing through e instead of f $\quad E_{T'} = (E_T \backslash \{f\}) \cup \{e\}$.
  - ➜ T' is covering
  - ➜ T' is minimal $\quad \Omega(T') = \Omega(T) - \omega(f) + \omega(e) \leq \Omega(T)$
  - ➜ T' includes $E' \cup \{e\}$ $\quad E' \cup \{e\} \subseteq E_{T'}$

➜ Selecting e with the minimum weight amongst all edges that cross a cut respecting E' always provides a safe edge!

Prim and Kruskal

Choice of a safe edge

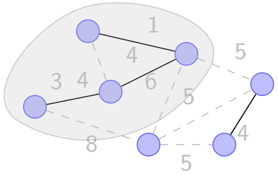
Prim

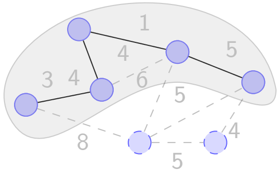
- Cut: $S =$ set of all vertice that are endpoints of one edge in E'

Kruskal

- Minimal edge $\rightarrow$ defines the cut!

➜ Cut between the two subgraphs that we will regroup





Plan

1. Introduction to the problem
2. Problem solving
3. Implementation of Kruskal algorithm
4. Clustering
5. Conclusion

Implementation of Kruskal algorithm

Choosing the data structure

We need to store T and its sub-trees.

Constraints

We have to perform the following operations:

- Initialize the data structure (singletons)
- For each vertex v, find the set that contains v
- Merge two sets

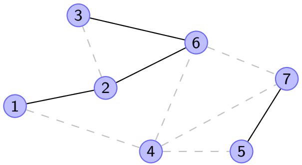
Naive approach

Most simple structure

An array `Kruskal_tab` of n integer values giving the number of the set that contains the vertex i.

```
Kruskal_tab = [1,1,1,4,5,1,5]
```

and we keep the array of selected edges `[(3,6),(1,2),...]`



Naive approach

Most simple structure

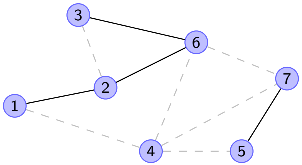
An array `Kruskal_tab` of n integer values giving the number of the set that contains the vertex i.

```
Kruskal_tab = [1,1,1,4,5,1,5]
```

and we keep the array of selected edges `[(3,6),(1,2),...]`

Complexity

- Initialization : $\mathcal{O}(|V|)$
- Find the set containing a vertex : $\mathcal{O}(1)$
- Merge : $\mathcal{O}(|V|)$



Union-Find approach

Principle

A data structure to implement a partition with two primitives:

- **Find** the set of the partition containing a given element
- **Union** to merge 2 sets of the partition

Union-Find approach

Principle

A data structure to implement a partition with two primitives:

- **Find** the set of the partition containing a given element
- **Union** to merge 2 sets of the partition

2 concrete implementations

- Linked lists
- Trees

Union-Find approach

Principle

A data structure to implement a partition with two primitives:

- **Find** the set of the partition containing a given element
- **Union** to merge 2 sets of the partition

2 concrete implementations

- Linked lists
- Trees

Best complexity : Trees

- Initialization : $\mathcal{O}(|V|)$
- Find the set containing a vertex : $\mathcal{O}(\log|V|)$ (the tree height)
- Merge : $\mathcal{O}(\log|V|)$ (to balance the tree)

Complexity of the Kruskal algorithm

total cost

- Initial sorting of edges : $\mathcal{O}(|E| \log |E|)$

  so $\mathcal{O}(|E| \log |V|)$ *as* $|E| \leq |V|^2$
- Initializing the T structure: $\mathcal{O}(|V|)$
- Find the set containing a vertex called at most $2 \times |E|$ times
  - With a naive array : $\mathcal{O}(|E|)$
  - With trees : $\mathcal{O}(|E| \log |V|)$
- Merge called in the worst case $|V| - 1$ times
  - With a naive array : $\mathcal{O}(|V|^2)$
  - With trees : $\mathcal{O}(|V| \log |V|)$

Plan

Motivation for Clustering

- Given a set of objects and distances between them.
  - Objects can be images, web pages, people, documents . . . .
- Distance function
  - Increasing distance corresponds to decreasing similarity.

Motivation for Clustering

- Given a set of objects and distances between them.
  - Objects can be images, web pages, people, documents . . . .
- Distance function
  - Increasing distance corresponds to decreasing similarity.

Clustering Goal

Group objects into clusters, where each cluster is a set of similar objects.

Formalizing the Clustering Problem

- Let O be the set of n objects labeled $o_1, o_2, \ldots, o_n$.
- For every pair o_i and o_j, we have a positive distance $d(o_i, o_j)$.

Formalizing the Clustering Problem

- Let O be the set of n objects labeled $o_1, o_2, \ldots, o_n$.
- For every pair o_i and o_j, we have a positive distance $d(o_i, o_j)$.
- Given a positive integer k, a k-clustering of O is a partition of O into k non-empty subsets $C_1, C_2, \ldots C_k$ called clusters.

Formalizing the Clustering Problem

- Let O be the set of n objects labeled $o_1, o_2, \ldots, o_n$.
- For every pair o_i and o_j, we have a positive distance $d(o_i, o_j)$.
- Given a positive integer k, a k-clustering of O is a partition of O into k non-empty subsets $C_1, C_2, \ldots C_k$ called clusters.
- The spacing of a clustering is the smallest distance between objects in two different clusters:

$$\text{spacing}(C_1, C_2, \ldots C_k) = \min\{d(a, b),\ i \neq j \wedge a \in C_i \wedge b \in C_j\}$$

Formalizing the Clustering Problem

- Let O be the set of n objects labeled $o_1, o_2, \ldots, o_n$.
- For every pair o_i and o_j, we have a positive distance $d(o_i, o_j)$.
- Given a positive integer k, a k-clustering of O is a partition of O into k non-empty subsets $C_1, C_2, \ldots C_k$ called clusters.
- The spacing of a clustering is the smallest distance between objects in two different clusters:

$$\text{spacing}(C_1, C_2, \ldots C_k) = \min\{d(a, b),\ i \neq j \wedge a \in C_i \wedge b \in C_j\}$$

Clustering of Maximum Spacing Problem

Find a k-clustering of O whose spacing is maximum over all possible k-clusterings.

Algorithm for Clustering of Maximum Spacing

- Intuition?

Algorithm for Clustering of Maximum Spacing

- Intuition?

Apply Kruskal's algorithm but do not add last $k-1$ edges in MST.

Algorithm for Clustering of Maximum Spacing

- Intuition?

Apply Kruskal's algorithm but do not add last $k-1$ edges in MST.

- What is the spacing value?

Algorithm for Clustering of Maximum Spacing

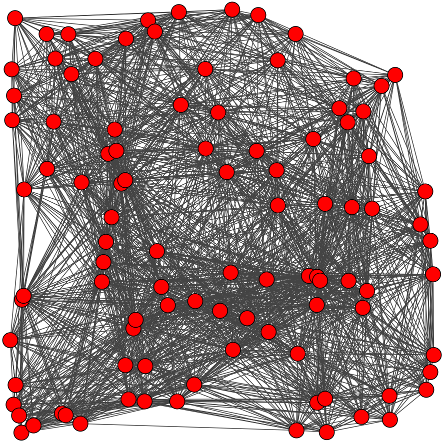
- Intuition?

Apply Kruskal's algorithm but do not add last $k-1$ edges in MST.

- What is the spacing value?
  - It is the weight of the $(k-1)^{st}$ most expensive edge in the MST generated by Kruskal's algorithm.

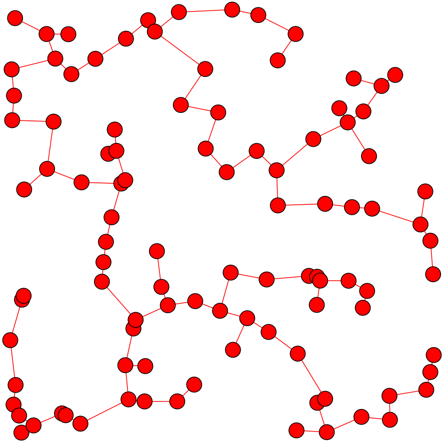
Clustering Example

Consider a complete graph, weights are the Cartesian distances



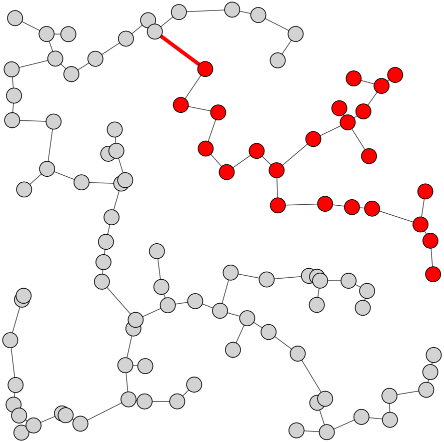
Clustering Example

The minimum spanning tree : Kruskal Algorithm



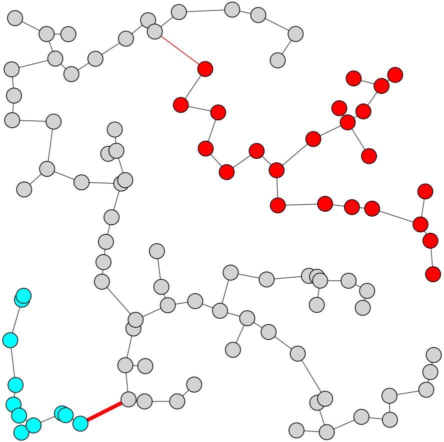
Clustering Example

2 clusters by dropping the last Kruskal edge



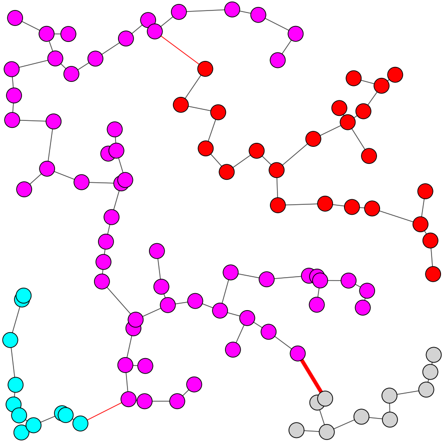
Clustering Example

3 clusters by dropping the 2 last Kruskal edges



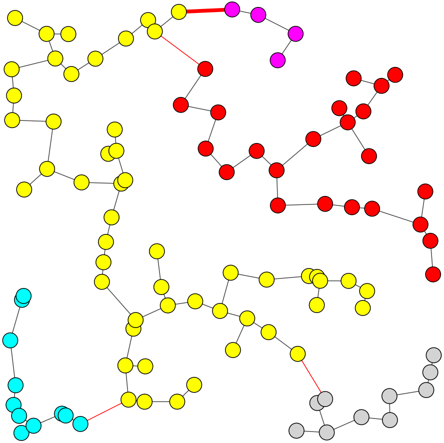
Clustering Example

So on . . .



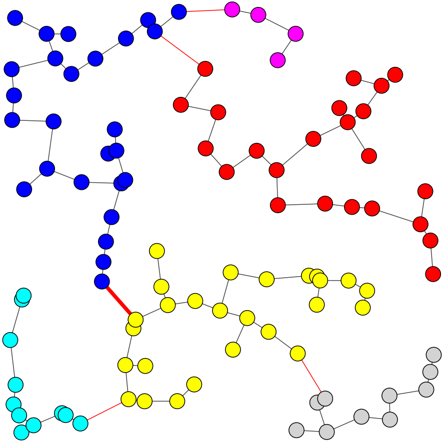
Clustering Example

So on . . .



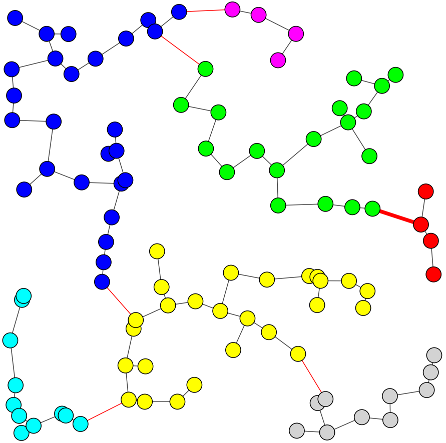
Clustering Example

So on . . .



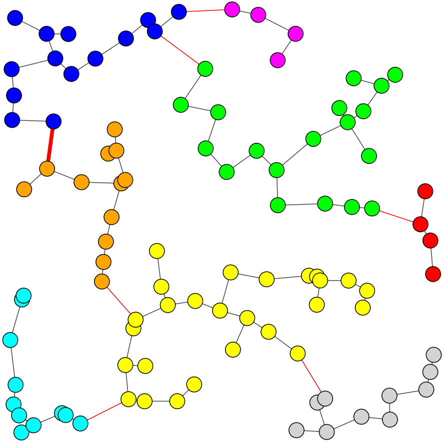
Clustering Example

So on . . .



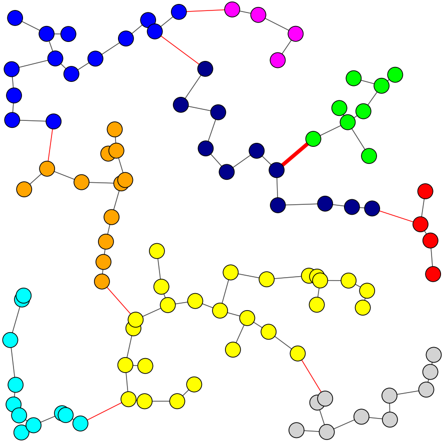
Clustering Example

So on . . .



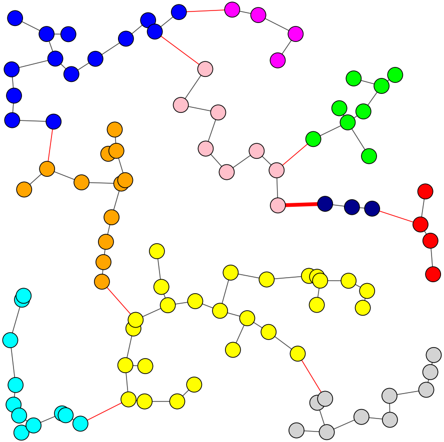
Clustering Example

So on . . .



Clustering Example

10 clusters of maximum spacing!



Plan

To keep in mind

- There are efficient algorithms to compute a minimum spanning tree
  - Kruskal → adds an edge of minimal weight
  - Prim → adds the closest neighbor to the current tree
  - ➜ They may not give the same solution...
    ...but both are optimal!
- Data structure to implement the algorithm
  - ➜ Impact on computing time (time complexity)
  - ➜ Structure of type union-find
- Many applications to computing problems (example: *clustering*)