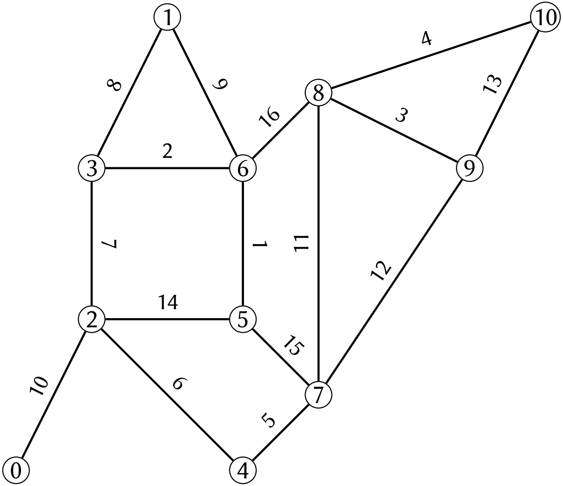
Algorithmique 2 (INFO-F203)
Arbres Couvrants

Jean Cardinal

Mars 2021

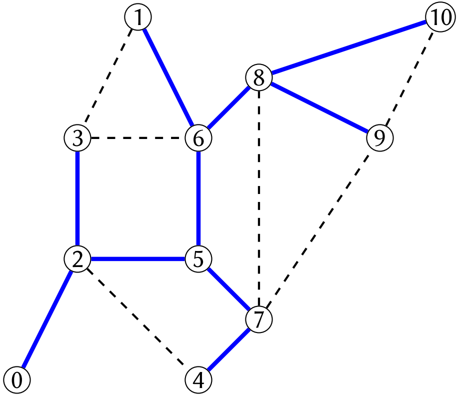
Graphes Pondérés



On note $w(e)$ le poids de l'arête e.

Arbres Couvrants

Un *arbre couvrant* (spanning tree) dans un graphe G est un sous-graphe connexe et acyclique de G contenant tous les sommets de G. Un arbre couvrant dans un graphe à V sommets contient exactement $V - 1$ arêtes.



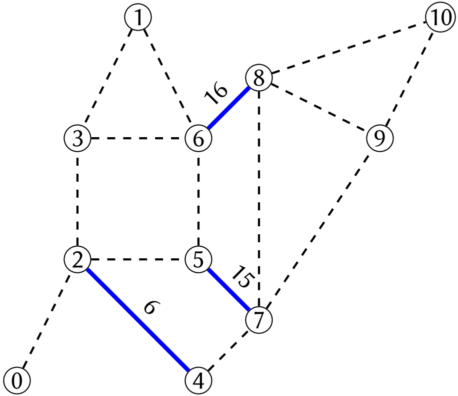
Exemples d’applications

- Distribution d’électricité (Borůvka 1926)
- Protocoles Réseaux (e.g. *Spanning Tree Protocol*, Perlman 1988)
- Analyse de partitionnement de données (“Cluster analysis”)
- Recalage automatique d’images médicales (Ma et al. 2000)
- Segmentation d’images (Felzenszwalb 2004)

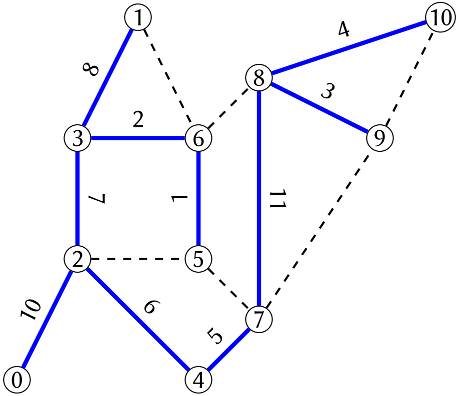
Propriété des coupes

Proposition 1

Dans un graphe G pondéré sur les arêtes, où toutes les arêtes ont un poids positif distinct, pour toute coupe C de G, l'arête de poids minimum de C fait partie de l'arbre couvrant de poids minimum de G.



Arbre couvrant de poids minimal

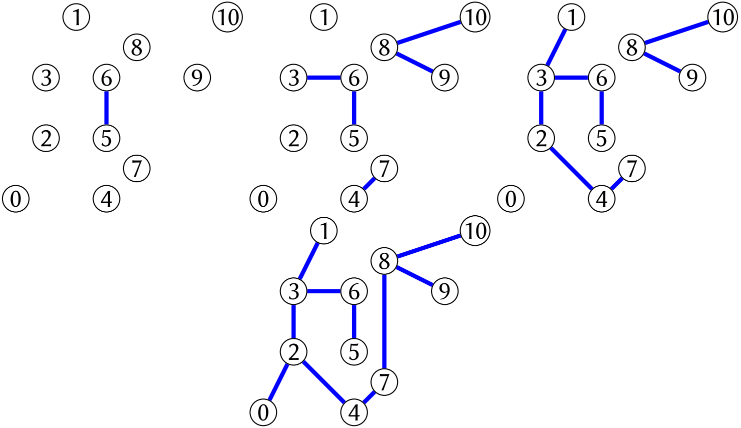


Algorithme Glouton (Kruskal)

- On considère les arêtes une à une, dans l'ordre croissant de leurs poids.
- Chacune des arêtes est choisie si elle ne forme pas de cycle avec les arêtes déjà choisies précédemment.
- Dans le cas contraire, elle est ignorée, et on procède avec l'arête suivante dans la liste.
- Lorsque toutes les arêtes ont été examinées, les arêtes choisies forment l'arbre couvrant minimum.

Voir la vidéo sur l'algorithme de Kruskal.

Exemple

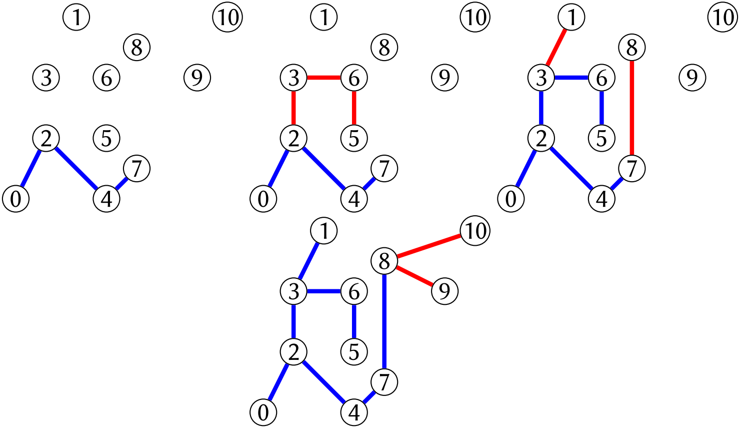


Algorithme de Prim

Contrairement à l'algorithme de Kruskal, l'algorithme de Prim maintient à tout moment un ensemble d'arêtes induisant un seul arbre. À chaque étape, on ajoute à cet ensemble l'arête de poids minimum ayant exactement une extrémité dans l'arbre. Plus précisément :

1. Initialiser l'arbre actuel T au seul sommet initial 0,
2. ajouter à T l'arête de poids minimum parmi toutes les arêtes ayant exactement une extrémité dans T,
3. répéter l'opération précédente jusqu'à ce que T contienne $V - 1$ arêtes.

Exemple



Mise en œuvre efficace

- On peut sélectionner l'arête suivante en la choisissant dans une file de priorité. On obtient une complexité en $O(E \log E)$.
- Meilleure idée :
  - on maintient une file de priorité `pq` contenant des *sommets*.
  - La clé associée à un sommet v sera le poids minimum d'une arête connectant v à T.

Détail

- Trouver le sommet v de clé minimum dans la file de priorité $\mathtt{pq}$, et son arête $e = uv$ associée,
- ajouter e à T,
- mettre à jour la file de priorité $\mathtt{pq}$ en considérant toutes les arêtes vx incidentes à v :
    - Si x est déjà dans T, ignorer;
    - ajouter x dans $\mathtt{pq}$ s'il n'est pas déjà présent,
    - *décroître la clé* de x dans $\mathtt{pq}$ si l'arête vx devient l'arête de poids minimum connectant x à T.

Décroître une clé

- Cette opération peut être mise en œuvre efficacement dans la structure de tas vue précédemment.
- On maintient un tableau supplémentaire permettant de retrouver l'indice d'une clé dans le tableau représentant le tas.
- On peut alors décroître la priorité via l'opération `swim`, en temps au pire cas $\log V$.

Code

```
public PrimMST(EdgeWeightedGraph G)
{
  edgeTo = new Edge[G.V()];
  distTo = new double[G.V()];
  marked = new boolean[G.V()];
  for (int v = 0; v < G.V(); v++)
    distTo[v] = Double.POSITIVE_INFINITY;
  pq = new IndexMinPQ<Double>(G.V());
  distTo[0] = 0.0;
  pq.insert(0, 0.0);    // Initialize pq with
     0, weight 0.
  while (!pq.isEmpty())
    visit(G, pq.delMin()); // Add closest
       vertex to tree.
}
```

Code (suite)

```
private void visit(EdgeWeightedGraph G, int v)
{
  marked[v] = true;
  for (Edge e : G.adj(v))
  {
    int w = e.other(v);
    if (marked[w]) continue;
    if (e.weight() < distTo[w])
    { // Edge e is new best connection from
       tree to w.
      edgeTo[w] = e;
      distTo[w] = e.weight();
      if (pq.contains(w)) pq.change(w,
         distTo[w]);
      else               pq.insert(w, distTo[w]);
     }
   }
}
```

Complexité

Proposition 2

L'algorithme de Prim calcule l'arbre couvrant de poids minimal en temps proportionnel à $E \log V$ au pire cas.

Proof.

La file de priorité ne contient que des sommets, donc chaque opération s'effectue en temps $O(\log V)$. L'algorithme effectue V fois les opérations d'insertion et d'effacement du minimum dans la file de priorité, et au plus E fois l'opération *décroître une clé.* Le temps total est donc bien proportionnel à $E \log V$. □