Graphes II

Graphes II

1. Implementation
   1. Interface et classes de graphes orientés et non orientés
   2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
   1. Parcours DFS post-ordre et post-ordre inverse
   2. Algorithme d'ordre topologique
3. Composantes connexes
   1. Notion de connexité
   2. Composantes connexes (UG)
   3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
   1. Problématique
   2. Algorithme de Prim
   3. Algorithme de Kruskal

Graphes II

Implémentation

- Un graphe implémentant `Graph` sera formé de sommets auxquels seront associés à des entiers allant de 0 à |V|-1

```java
import java.util.HashSet;

public interface Graph {
   void initialize(int V);
   int V(); // cardinal de l'ensemble des sommets
   int E(); // cardinal de l'ensemble des arcs
   void connect(int v1, int v2);
   HashSet<Integer> adj(int v); // liste d'adjacence
   String toString();
}
```

Implémentation

- `UndirectedGraph` est un graphe non orienté sans poids sur les arcs implémentant `Graph`

```
import java.security.InvalidParameterException;
import java.util.HashSet;

public class UndirectedGraph implements Graph{

   private HashSet<Integer>[] neighbors; // listes d'adjacences
   private int V, E; // cardinal de V et cardinal de E

   public UndirectedGraph(int V){
      initialize(V);
   }
```

Implémentation

```
public void initialize(int V){
   // check parameters
   if(V < 0) throw new InvalidParameterException();

   // initialize members
   E =  0;
   this.V =  V;
   neighbors = new HashSet[V];

   for(int v=0; v<V; v++)
      neighbors[v] = new HashSet<Integer>();
}

public int V(){return V;}
public int E(){return E;}
```

Implémentation

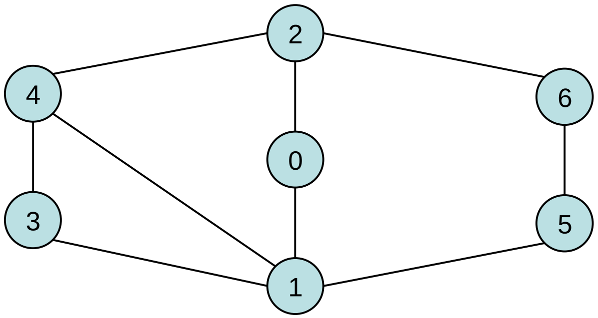
- Un graphe non orienté créera deux arcs pour relier deux sommets

```
public void connect(int v1, int v2){
    // check parameters
    if(v1<0 || v1>=V) return;
    if(v2<0 || v2>=V) return;
    if( neighbors[v1].contains(v2) ) return;

    // connect in both directions
    neighbors[v1].add(v2);
    neighbors[v2].add(v1);
    E++;
}
```

Implémentation

- `toString()` nous servira à définir un graphe non orienté

```
   public HashSet<Integer> adj(int v){
      // check parameters
      if(v<0 || v>=V) return null;
      return neighbors[v];
   }

   public String toString(){
      StringBuilder o = new StringBuilder();
      String ln = System.getProperty("line.separator");
      o.append(V + ln + E + ln);
      for(int v=0; v<V; v++)
         for(int w : neighbors[v])
            o.append(v + "-" + w + ln);
      return o.toString();
   }
}
```




```
7
9
0-1
0-2
1-0
1-3
1-4
1-5
2-0
2-4
2-6
3-1
3-4
4-1
4-2
4-3
5-1
5-6
6-2
6-5
```

Implémentation

- `DirectedGraph` est un graphe orienté sans poids sur les arcs implémentant `Graph`

```
import java.security.InvalidParameterException;
import java.util.HashSet;

public class DirectedGraph implements Graph{

   private HashSet<Integer>[] neighbors; // listes d'adjacences
   private int V, E; // cardinal de V et cardinal de E

   public DirectedGraph(int V){
      initialize(V);
   }
```

Implémentation

- Les méthodes `initialiaze(…)`, `V()` et `E()` sont identiques à celles de `UnirectedGraph`. `connect()` est également similaire, excepté qu’ un seul arc est ajouté, il va de `v1` à `v2`

```
public void initialize(int V){…}
public int V(){return V;}
public int E(){return E;}

public void connect(int v1, int v2){
   // check parameters
   if(v1<0 || v1>=V) return;
   if(v2<0 || v2>=V) return;
   if( neighbors[v1].contains(v2) ) return;

   // connect edge from v1 to v2
   neighbors[v1].add(v2); E++;
}
```

Implémentation

- On ajoutera la méthode `transposed()` qui retourne un graphe orienté dont les arcs sont été inversés:

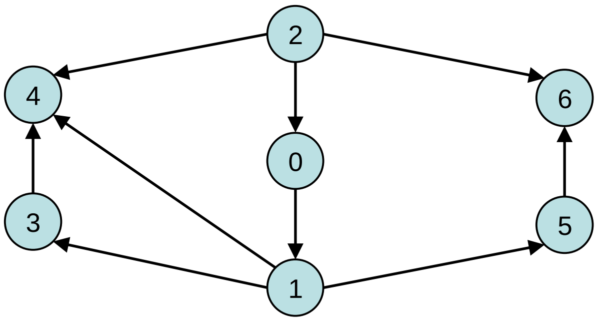
```
public DirectedGraph transposed(){

   DirectedGraph T = new DirectedGraph(V);

   for(int v=0; v<V; v++)
      for(int w : neighbors[v])
         T.connect(w, v);
   return T;
}
```

Implémentation

- `toString()` nous servira à définir un graphe orienté (notez `->`)

```
public HashSet<Integer> adj(int v){
   // check parameters
   if(v<0 || v>=V) return null;
   return neighbors[v];
}

public String toString(){
   StringBuilder o = new StringBuilder();
   String ln = System.getProperty("line.separator");
   o.append(V + ln + E + ln);
   for(int v=0; v<V; v++)
      for(int w : neighbors[v])
         o.append(v + "->" + w + ln);
   return o.toString();
}
}
```

```
7
9
0->1
1->3
1->4
1->5
2->0
2->4
2->6
3->4
5->6
```



Graphes II

1. Implementation
   1. Interface et classes de graphes orientés et non orientés
   2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
   1. Parcours DFS post-ordre et post-ordre inverse
   2. Algorithme d'ordre topologique
3. Composantes connexes
   1. Notion de connexité
   2. Composantes connexes (UG)
   3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
   1. Problématique
   2. Algorithme de Prim
   3. Algorithme de Kruskal

Implémentation

- `Paths` implémente les parcours de graphe

```java
import java.security.InvalidParameterException;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Stack;

public class Paths {

   boolean[] dfsMarked, bfsMarked;
   int[] dfsParent, bfsParent;
   int s;
```

Implémentation

- Le constructeur de `Paths` appelle les deux parcours

```
public Paths(Graph G, int s){
   if(G == null || s < 0 || s>= G.V())
      throw new InvalidParameterException();
   this.s = s;

   // process bfs
   bfsMarked = new boolean[G.V()];
   bfsParent = new int[G.V()];
   bfs(G, s);

   // process dfs
   dfsMarked = new boolean[G.V()];
   dfsParent = new int[G.V()];
   dfs(G, s);
}
```

Implémentation

- BFS se fait au moyen d'une file:

```java
private void bfs(Graph G, int s){
   Queue<Integer> q = new LinkedList<Integer>();

   // add source
   q.add(s); bfsMarked[s] = true;

   while( !q.isEmpty() ){
      // poll vertex and treat neighbors
      int v = q.poll();
      for(int w : G.adj(v)){
         if( !bfsMarked[w] ){
            q.add(w);
            bfsMarked[w] = true;
            bfsParent[w] = v;
         }
      }
   } // end while
}
```


Implémentation

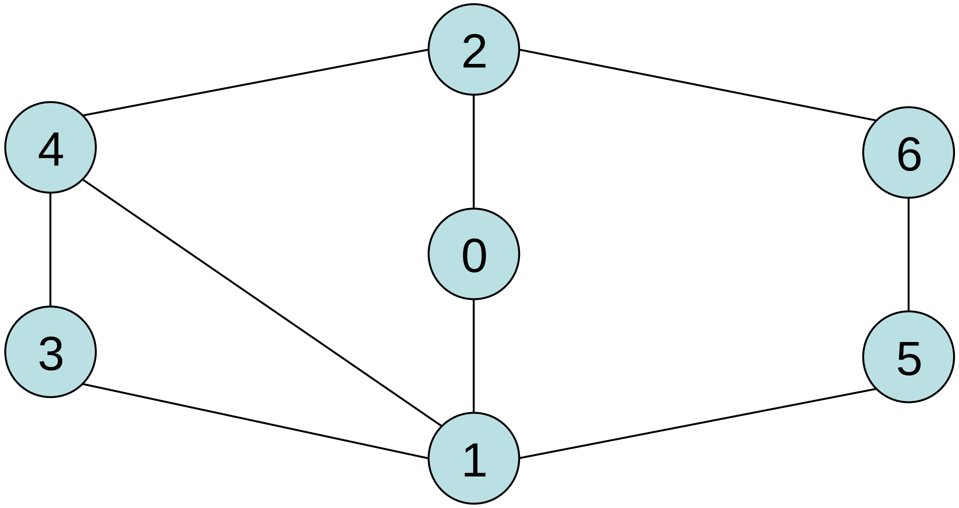
- On récupère le chemin BFS en empilant les parents depuis la destination jusqu'à la source:

```
   public Stack<Integer> bfsPathTo(int v){

      if( !bfsMarked[v] ) return null;

      Stack<Integer> path = new Stack<Integer>();

      for(int x = v; x != s; x = bfsParent[x])
         path.push(x);
      path.push(s);

      return path;
   }
}
```

Implémentation

- Résultat sur le UndirectedGraph précédent (s == 2):

```
bfsPathTo(3) retournera le chemin 2-4-3
```




```
7
9
0-1
0-2
1-0
1-3
1-4
1-5
2-0
2-4
2-6
3-1
3-4
4-1
4-2
4-3
5-1
5-6
6-2
6-5
```

Implémentation

- DFS se fait de manière récursive:

```
private void dfs(Graph G, int v){
   dfsMarked[v] = true;

   for(int w : G.adj(v))
      if( !dfsMarked[w] ){
         dfs(G, w);
         dfsParent[w] = v;
      }
}
```

Implémentation

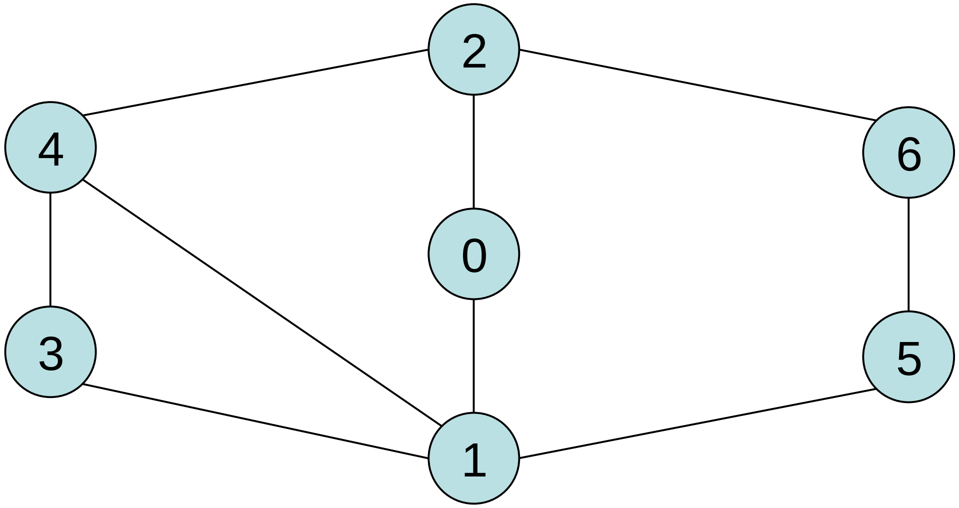
- On récupère le chemin DFS en empilant les parents depuis la destination jusqu'à la source:

```
public Stack<Integer> dfsPathTo(int v){

   if( !dfsMarked[v] ) return null;

   Stack<Integer> path = new Stack<Integer>();

   for(int x = v; x != s; x = dfsParent[x])
      path.push(x);
   path.push(s);

   return path;
}
```

Implémentation

- Résultat sur le UndirectedGraph précédent (s == 2):

```
dfsPathTo(3) retournera le chemin 2-0-1-3
```




```
7
9
0-1
0-2
1-0
1-3
1-4
1-5
2-0
2-4
2-6
3-1
3-4
4-1
4-2
4-3
5-1
5-6
6-2
6-5
```

Graphes II

1. Implementation
   1. Interface et classes de graphes orientés et non orientés
   2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
   1. Parcours DFS post-ordre et post-ordre inverse
   2. Algorithme d’ordre topologique
3. Composantes connexes
   1. Notion de connexité
   2. Composantes connexes (UG)
   3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
   1. Problématique
   2. Algorithme de Prim
   3. Algorithme de Kruskal

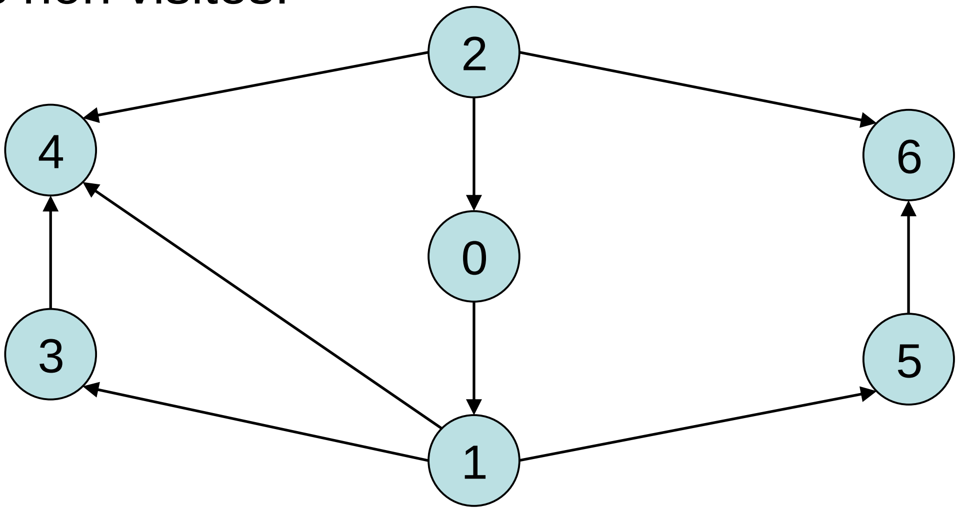
Ordre topologique II

<u>Rappel:</u>

- Nous avons vu au cours précédent un algorithme permettant de déterminer l'ordre topologique d'un graphe dirigé acyclique
- L'algorithme du cours précédent se basait sur une file
- Nous allons voir un nouvel algorithme pour déterminer l'ordre topologique qui se base sur le parcours DFS
- Pour ce faire, nous allons définir le parcours DFS post-ordre

Ordre topologique II

- Un parcours DFS post-ordre est le résultat d'un parcours en profondeur du graphe où un sommet est énuméré dès qu'il n'a plus de voisins non visités:



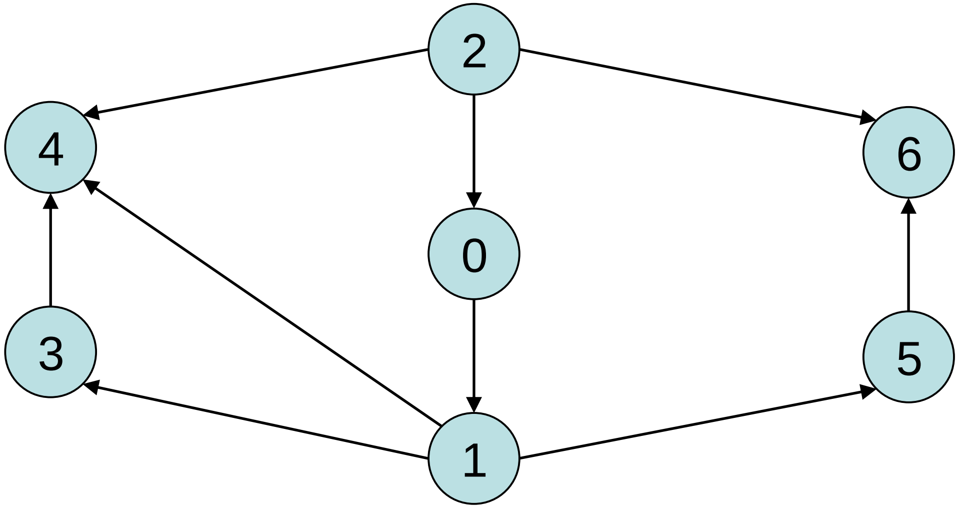

```
7
9
0->1
1->3
1->4
1->5
2->0
2->4
2->6
3->4
5->6
```

- Résultat:

```
4, 3, 6, 5, 1, 0, 2
```

Ordre topologique II

- Le parcours DFS post-ordre inverse est le résultat inverse du parcours DFS post-ordre:




```
7
9
0->1
1->3
1->4
1->5
2->0
2->4
2->6
3->4
5->6
```

- Résultat:

```
2, 0, 1, 5, 6, 3, 4
```

Ordre topologique II

- On modifie DFS comme suit:

```
// new Paths member, must be initialized in constructor
private Stack<Integer> reversePostOrderDfs;

private void dfs(Graph G, int v){
   dfsMarked[v] = true;

   for(int w : G.adj(v))
      if( !dfsMarked[w] ){
         dfs(G, w);
         dfsParent[w] = v;
      }

   // Stack  vertex
   reversePostOrderDfs.push(v);
}
```

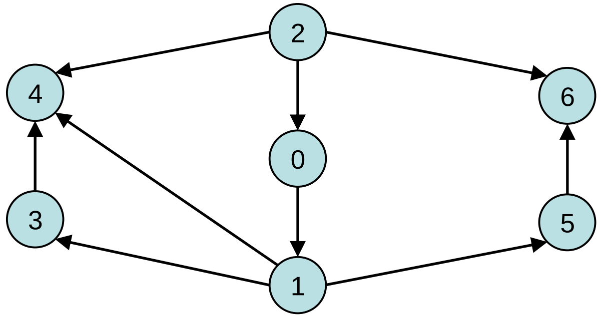
- L'ordre des nœuds du graphe obtenu d'un DFS post-ordre inverse est un ordre topologique:

Ordre topologique II

- Comparons avec l'algorithme vu au cours 10:

```
DFS Post-ordre 2, 0, 1, 5, 6, 3, 4
```

	Indegree						
0							
1							
2							
3							
4							
5							
6							
Entre en file							
Sort de file							

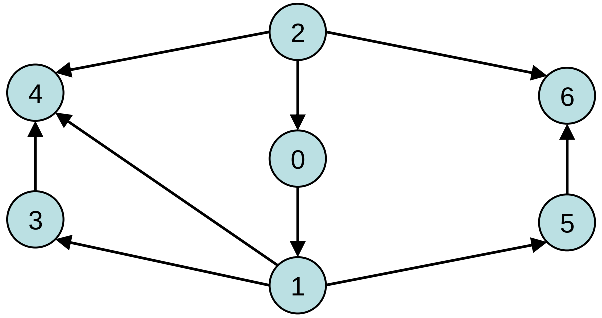


Ordre topologique II

- Comparons avec l'algorithme vu au cours 10:

```
DFS Post-ordre 2, 0, 1, 5, 6, 3, 4
```

	Indegree						
0	1	0	-	-	-	-	-
1	1	1	0	-	-	-	-
2	0	-	-	-	-	-	-
3	1	1	1	0	-	-	-
4	3	2	2	1	0	-	-
5	1	1	1	0	-	-	-
6	2	1	1	1	1	0	-
Entre en file	2	0	1	3, 5	4	6	-
Sort de file	2	0	1	3	5	4	6



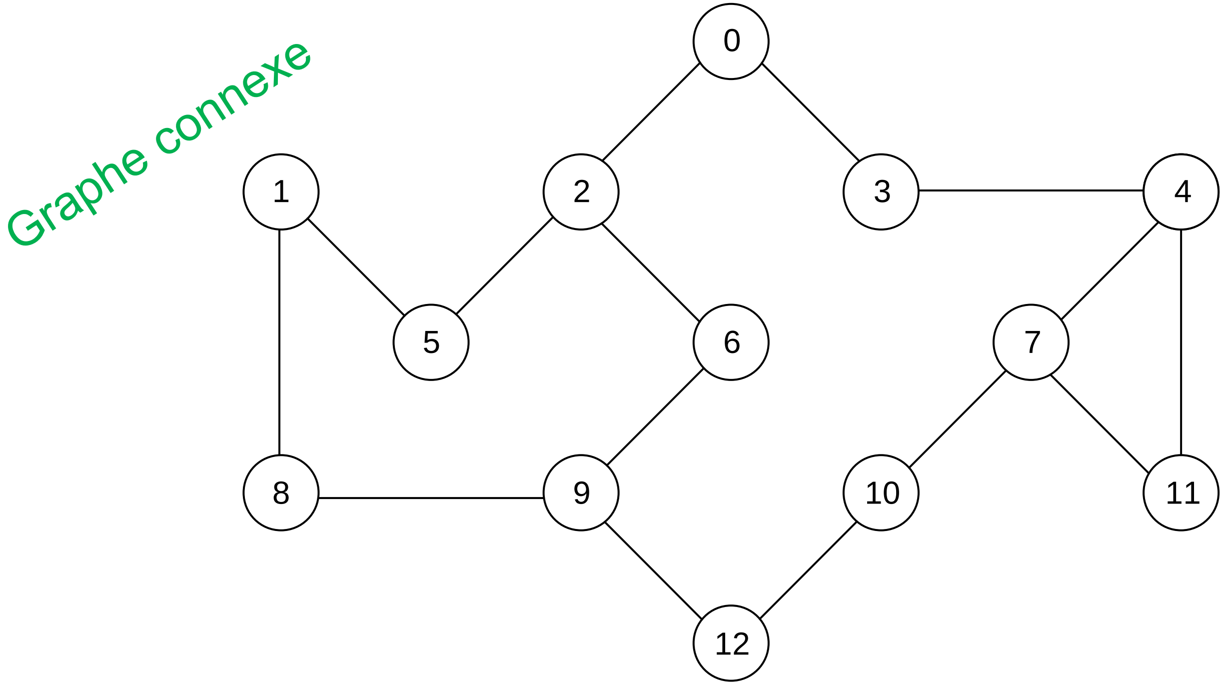
Graphes II

1. Implementation
   1. Interface et classes de graphes orientés et non orientés
   2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
   1. Parcours DFS post-ordre et post-ordre inverse
   2. Algorithme d’ordre topologique
3. Composantes connexes
   1. Notion de connexité
   2. Composantes connexes (UG)
   3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
   1. Problématique
   2. Algorithme de Prim
   3. Algorithme de Kruskal

Composantes connexes

Rappel:

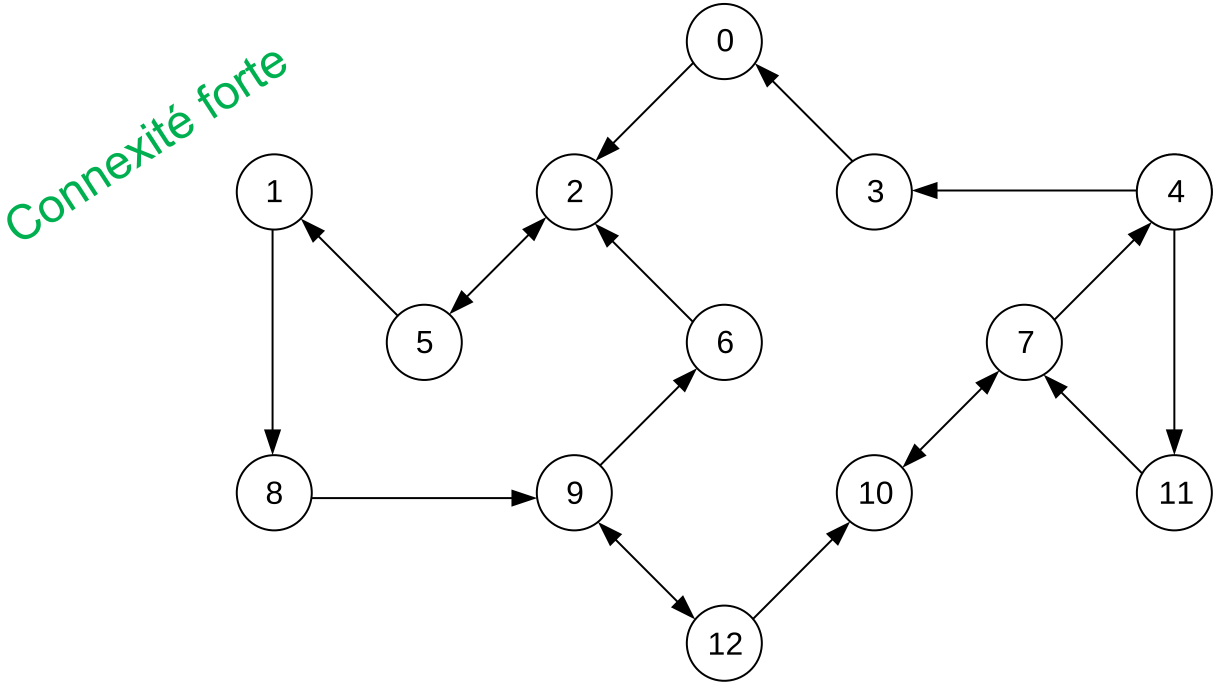
- Graphe connexe → un chemin pour chaque paire de nœuds



Composantes connexes

<u>Rappel:</u>

- Si un graphe orienté est connexe → on dit qu'il a une connexité forte



Graphes II

1. Implementation
   1. Interface et classes de graphes orientés et non orientés
   2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
   1. Parcours DFS post-ordre et post-ordre inverse
   2. Algorithme d’ordre topologique
3. Composantes connexes
   1. Notion de connexité
   2. Composantes connexes (UG)
   3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
   1. Problématique
   2. Algorithme de Prim
   3. Algorithme de Kruskal

Composantes connexes

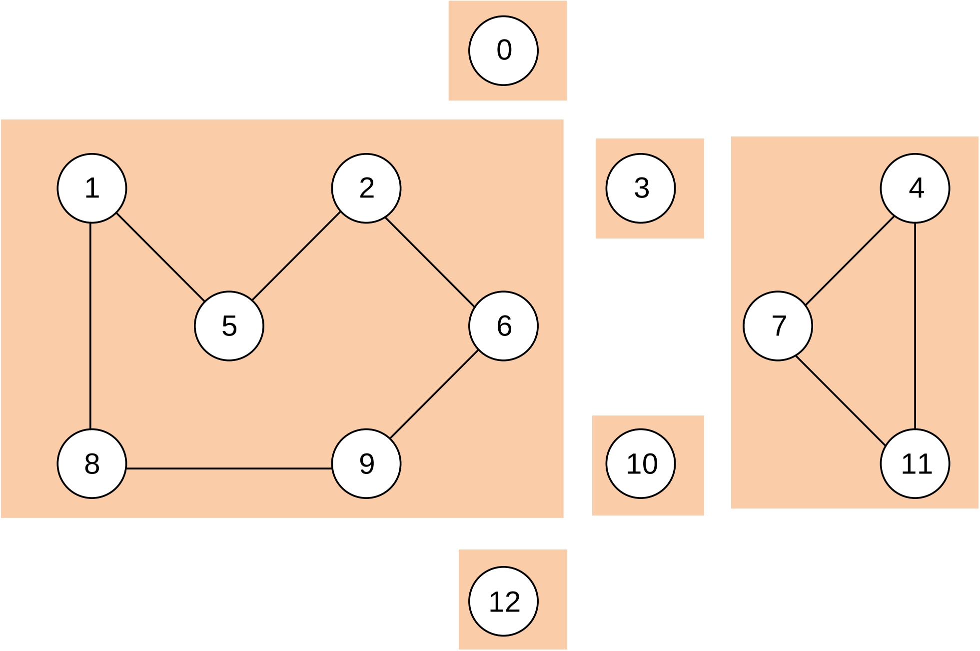
Objectif:

- Dans un graphe non orienté, identifier les composantes connexes. Par définition, un graphe connexe ne possèdera qu’une seule composante connexe.

Composantes connexes

<u>Exemples:</u>

- Ce graphe non dirigé possède 6 composantes connexes



Composantes connexes

Solution:

- Il suffit d'exécuter un parcours DFS. À chaque interruption, on début une nouvelle composante connexe.

```
public class ConnectedComponents {
   private boolean[] marked;
   private int[] id;
   private int count;

   public ConnectedComponents(UndirectedGraph G){
      if(G == null) throw new InvalidParameterException();

      marked = new boolean[G.V()];
      id     = new int[G.V()];

      for(int v=0; v<G.V(); v++)
         if( !marked[v] ){
            dfs(v, G);
            count++; // new component
         }
   }
```

```
private void dfs(int v, Graph G){
   marked[v] = true;

   // identify component
   id[v] = count;

   for(int w : G.adj(v))
      if(!marked[w])
         dfs(w, G);
}
```

Graphes II

1. Implementation
   1. Interface et classes de graphes orientés et non orientés
   2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
   1. Parcours DFS post-ordre et post-ordre inverse
   2. Algorithme d’ordre topologique
3. Composantes connexes
   1. Notion de connexité
   2. Composantes connexes (UG)
   3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
   1. Problématique
   2. Algorithme de Prim
   3. Algorithme de Kruskal

Composantes fortement connexes

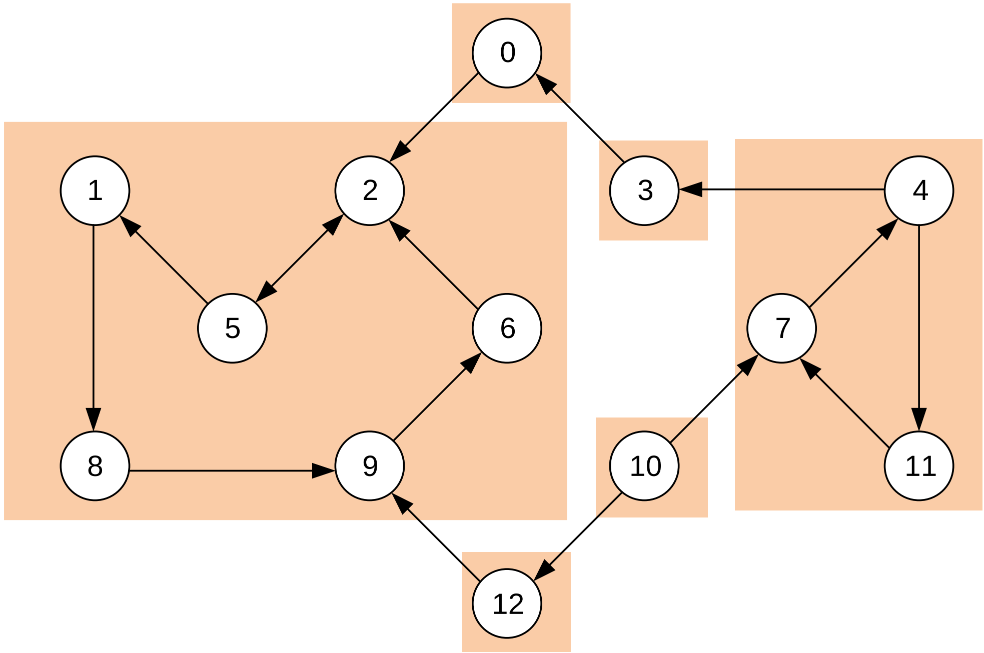
Objectif:

- Dans un graphe orienté, identifier les composantes fortement connexes. Par définition, un graphe orienté fortement connexe ne possèdera qu’une seule composante connexe.

Composantes fortement connexes

Exemples:

- Ce graphe orienté possède également 6 composantes fortement connexes



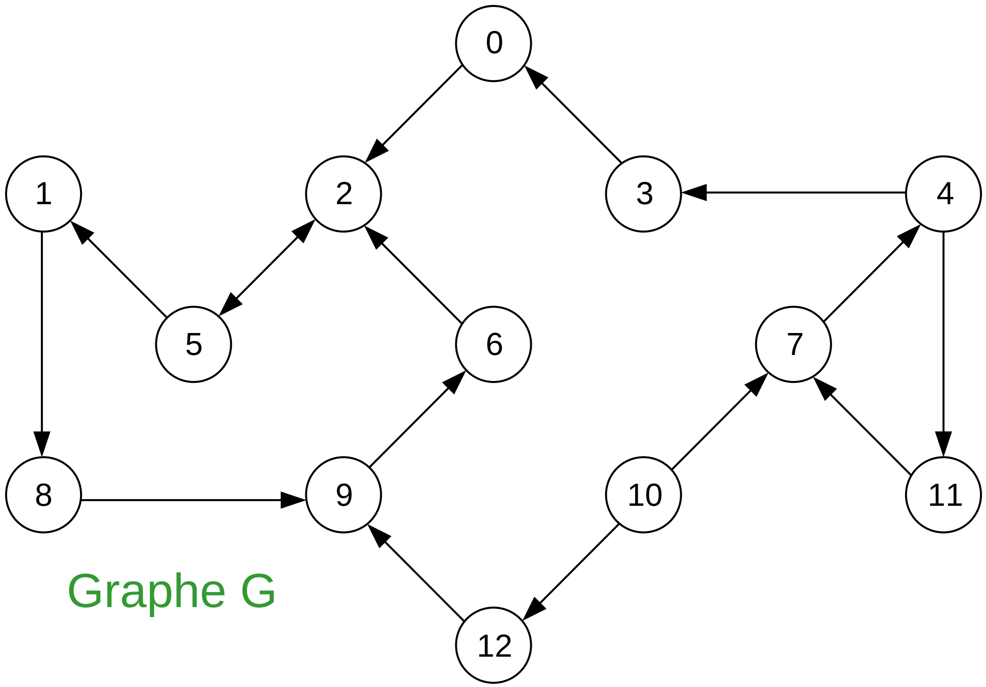
Composantes fortement connexes

<u>Solution:</u>

- Évidemment, un parcours DFS ne suffit pas.
- On remarquera cependant que les composantes fortement connexes de G le sont également de G^T.
- Un algorithme se basant sur cette observation et dû à S. Rao Kosaraju permet de résoudre le problème

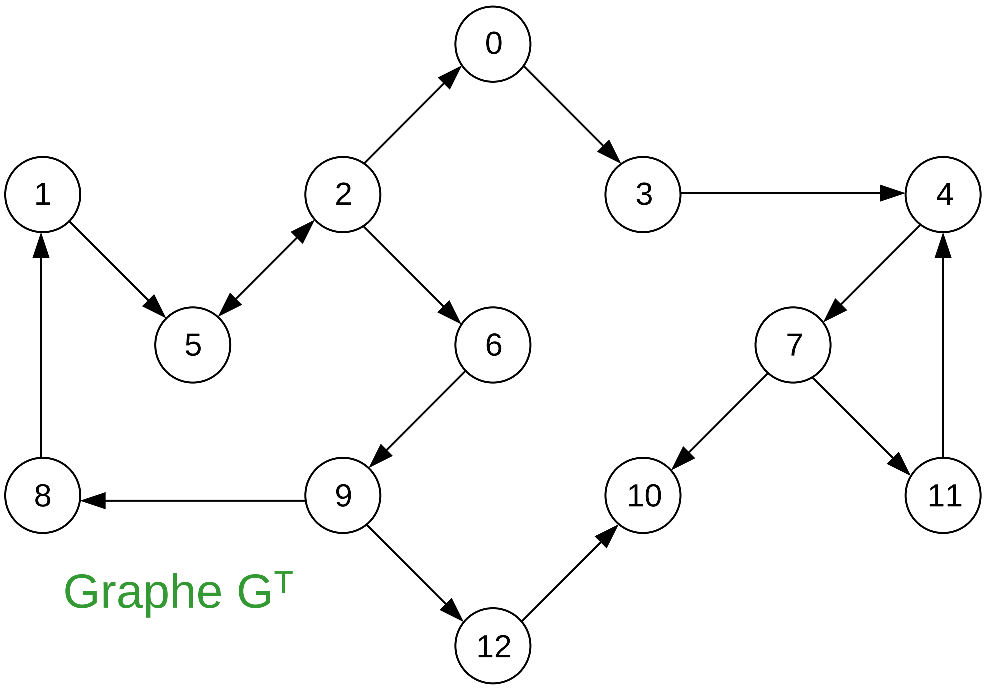
Composantes fortement connexes

1. Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de G^T
2. Parcourir G en DFS suivant l'ordre obtenu en (1.)



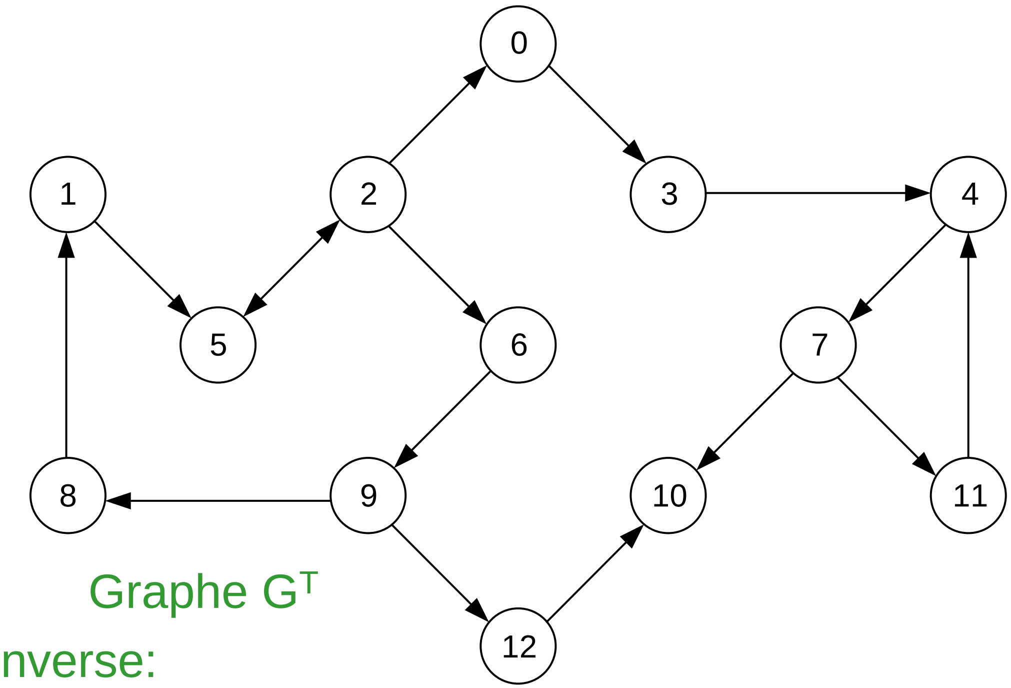
Composantes fortement connexes

1. Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de G^T
2. Parcourir G en DFS suivant l'ordre obtenu en (1.)



Composantes fortement connexes

1. **Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de G^T**
2. Parcourir G en DFS suivant l'ordre obtenu en (1.)



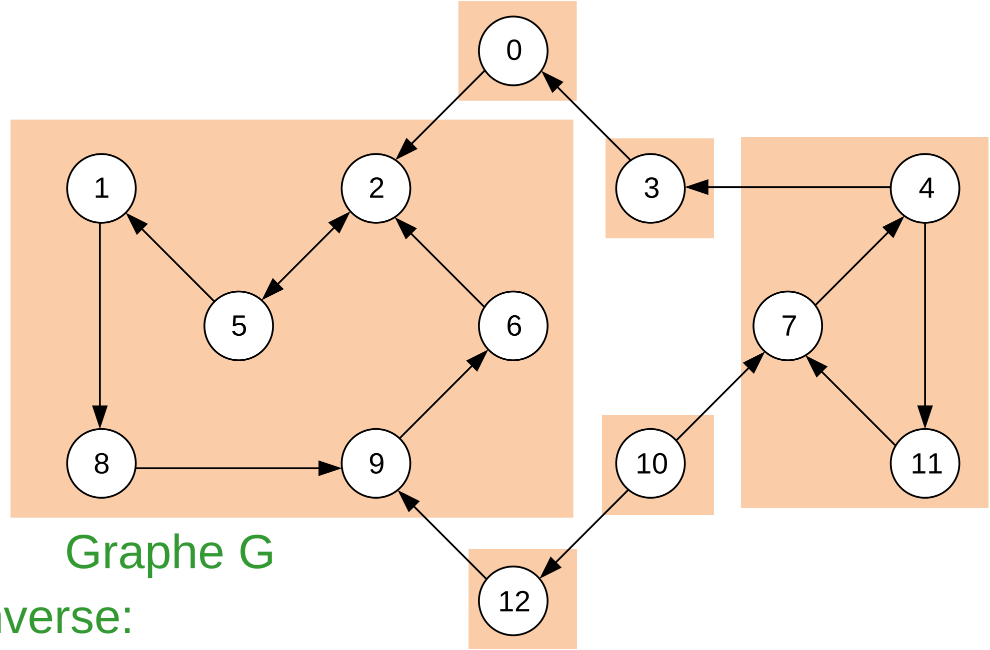
Graphe G^T

Post-ordre inverse:

`1, 5, 2, 6, 9, 12, 8, 0, 3, 4, 7, 11, 10`

Composantes fortement connexes

1. **Ordonner les nœuds du graphe obtenus d'un DFS post-ordre inverse de G^T**
2. Parcourir G en DFS suivant l'ordre obtenu en (1.)



Graphe G

Post-ordre inverse:

`1, 5, 2, 6, 9, 12, 8, 0, 3, 4, 7, 11, 10`

Composantes fortement connexes

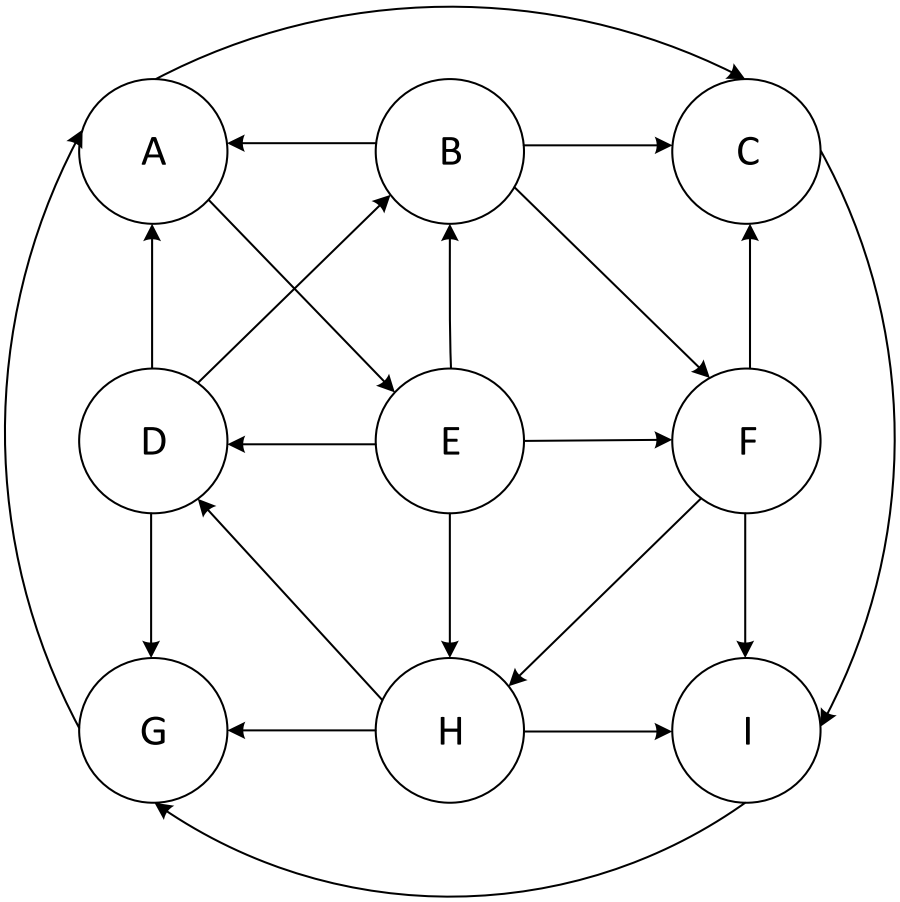
- Algorithme de KosarajuSharir.

```
public class StrongConnectedComponents {
   private boolean[] marked;
   private int[] id;
   private int count;

   public StrongConnectedComponents(DirectedGraph G){
      if(G == null) throw new InvalidParameterException();

      DepthFirstOrder dfo = new DepthFirstOrder(G.transposed());

      marked = new boolean[G.V()];
      id     = new int[G.V()];

      for( int v=0 : dfo.reversePost() )
         if( !marked[v] ){
            dfs(v, G);
            count++; // new component
         }
   }
```

```
private void dfs(int v, Graph G){
   marked[v] = true;

   // identify component
   id[v] = count;

   for(int w : G.adj(v))
      if(!marked[w])
         dfs(w, G);
}
```

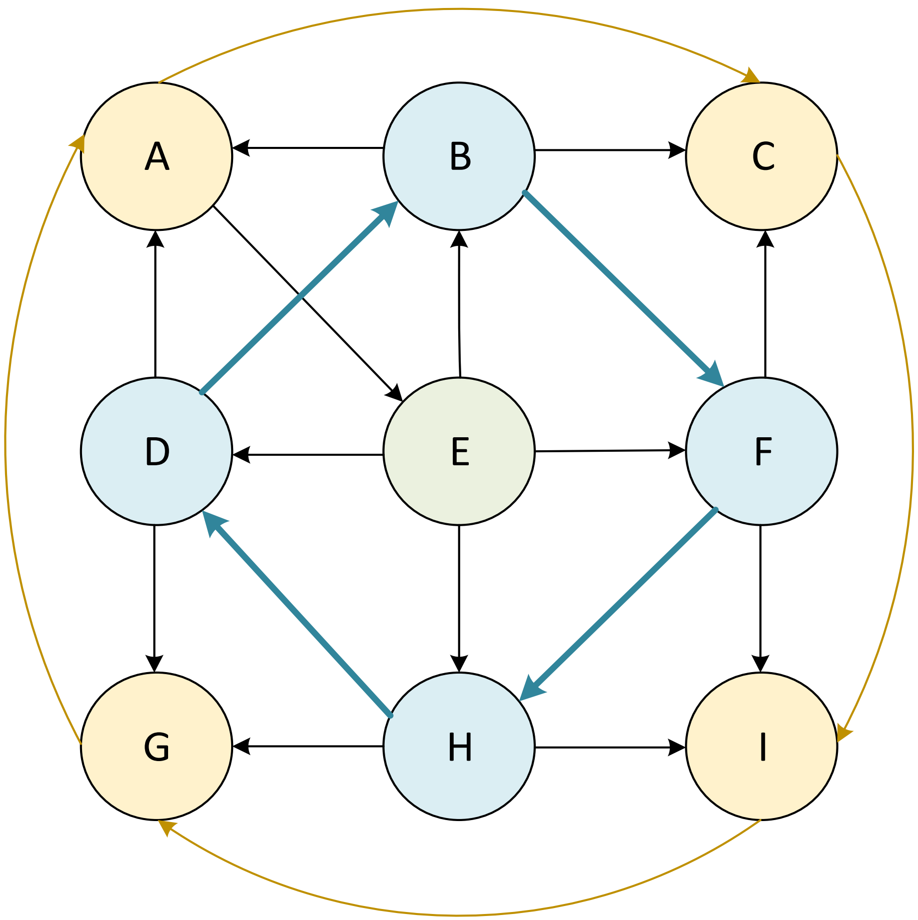
Composantes fortement connexes

- Exemple



Composantes fortement connexes

- Exemple



Graphes II

1. Implementation
   1. Interface et classes de graphes orientés et non orientés
   2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
   1. Parcours DFS post-ordre et post-ordre inverse
   2. Algorithme d’ordre topologique
3. Composantes connexes
   1. Notion de connexité
   2. Composantes connexes (UG)
   3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
   1. Problématique
   2. Algorithme de Prim
   3. Algorithme de Kruskal

Arbre sous-tendant minimum

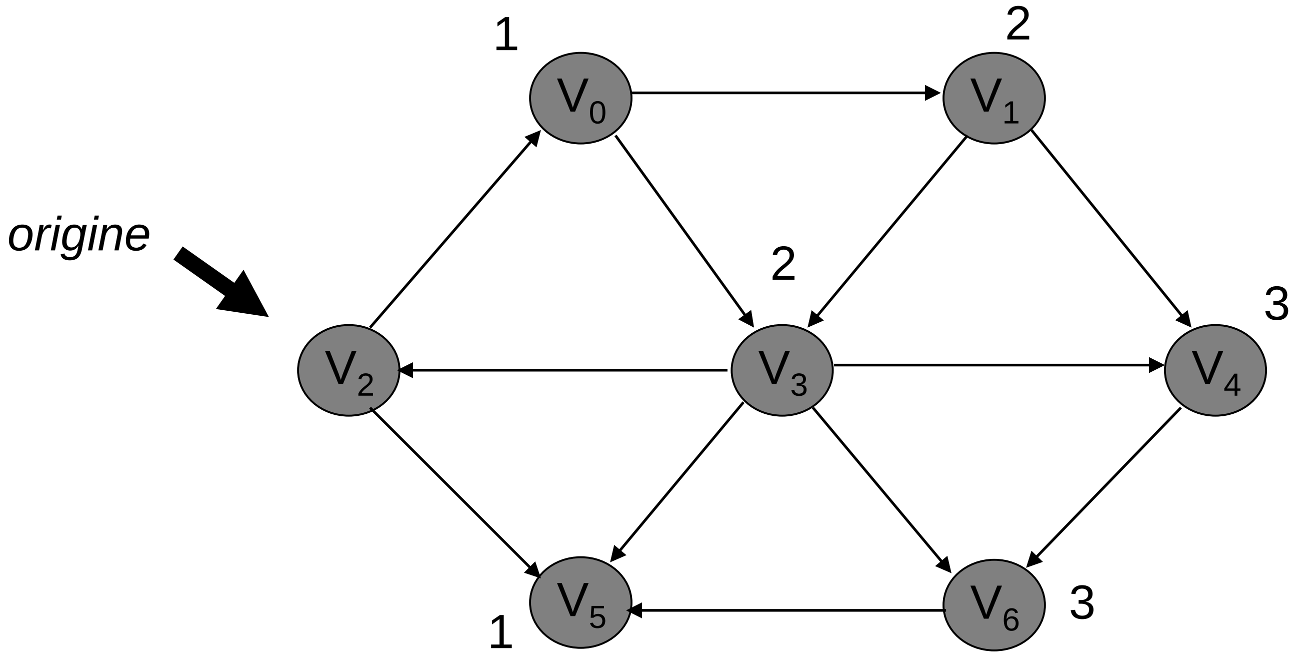
Problématique:

On cherche à relier toutes les sommets d'un graphe valué non orienté en ne retenant que certaines de ses arêtes, de sorte à réduire le coût total associé aux arêtes choisies.

Ce faisant, on définit un arbre sous-tendant le graphe. Cet arbre sous-tendant est dit minimum car le coût qui lui est associé est le plus bas sur l'ensemble des arbres sous-tendant ledit graphe.

Arbre sous-tendant minimum

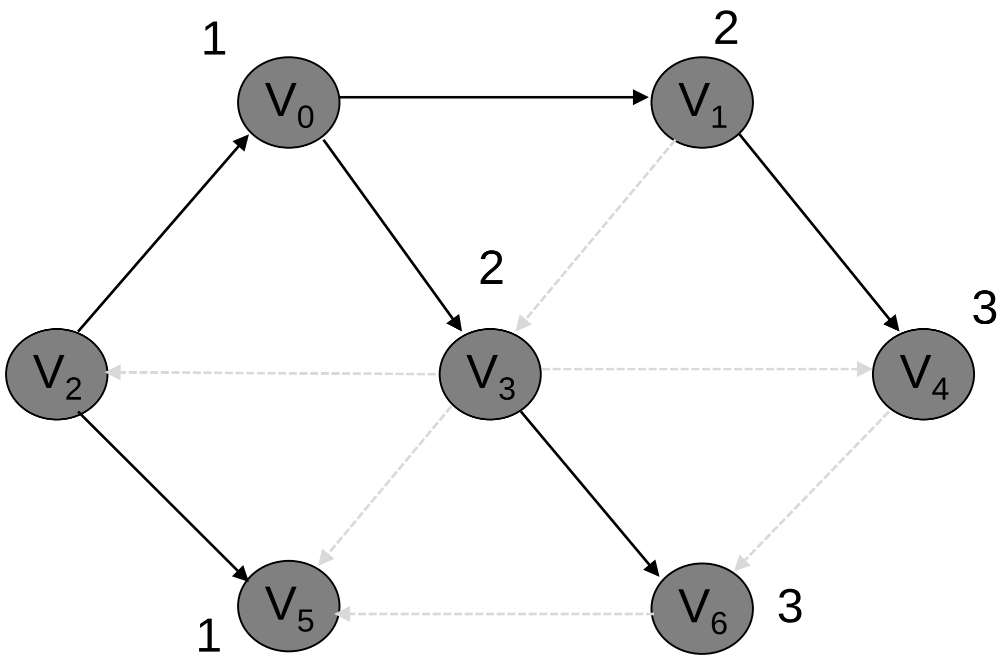
Rappel: Nous avions vu le rapprochement entre l'algorithme de chemin le plus court exécuté sur un graphe non valué:

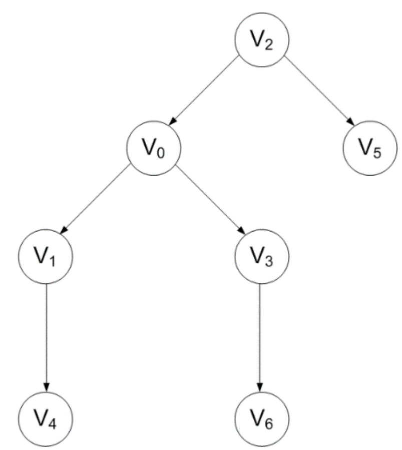


File: vide

Arbre sous-tendant minimum

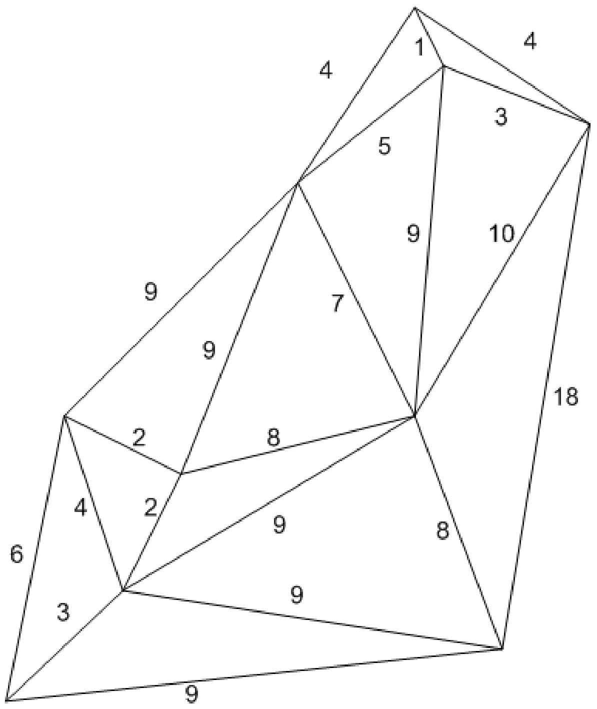
Le concept d’arbre sous-tendant d’un graphe n’est pas différent: il consiste à créer un arbre depuis un graphe en soustrayant certaines arêtes.





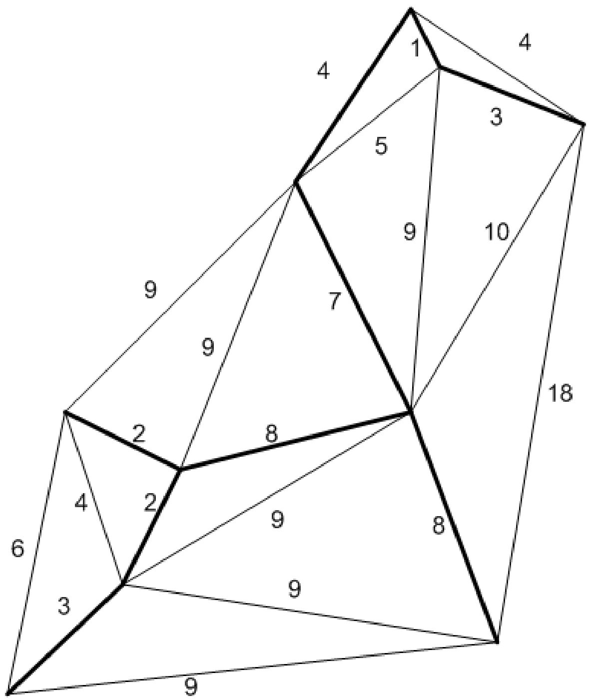
Arbre sous-tendant minimum

Le concept d’arbre sous-tendant **minimum** s’applique à un arbre valué non dirigé:



Arbre sous-tendant minimum

dont on veut **minimiser** le coût:



Arbre sous-tendant minimum

Cas type d'application – réseau de communication:

Fig. 1 — Example of a shortest connection network.

Extrait de : R. C. Prim: *Shortest connection networks and some generalizations* In: *Bell System Technical Journal*, 36 (1957), pp. 1389–1401

Graphes II

1. Implementation
    1. Interface et classes de graphes orientés et non orientés
    2. Class Paths (BFS + DFS)
2. Ordre topologique (version DFS)
    1. Parcours DFS post-ordre et post-ordre inverse
    2. Algorithme d'ordre topologique
3. Composantes connexes
    1. Notion de connexité
    2. Composantes connexes (UG)
    3. Composantes fortement connexes (DG)
4. Arbre sous-tendant minimum
    1. Problématique
    2. Algorithme de Prim
    3. Algorithme de Kruskal

Algorithme de Prim

L'algorithme que nous allons voir est dû à Robert Prim.
Ce dernier l'a publié en 1957. Il s'agit d'un algorithme **glouton**:
un choix optimal est réalisé étape par étape, jusqu'à obtenir la solution:

L'algorithme de Prim a le même comportement que Dijkstra, à quelques différences près.

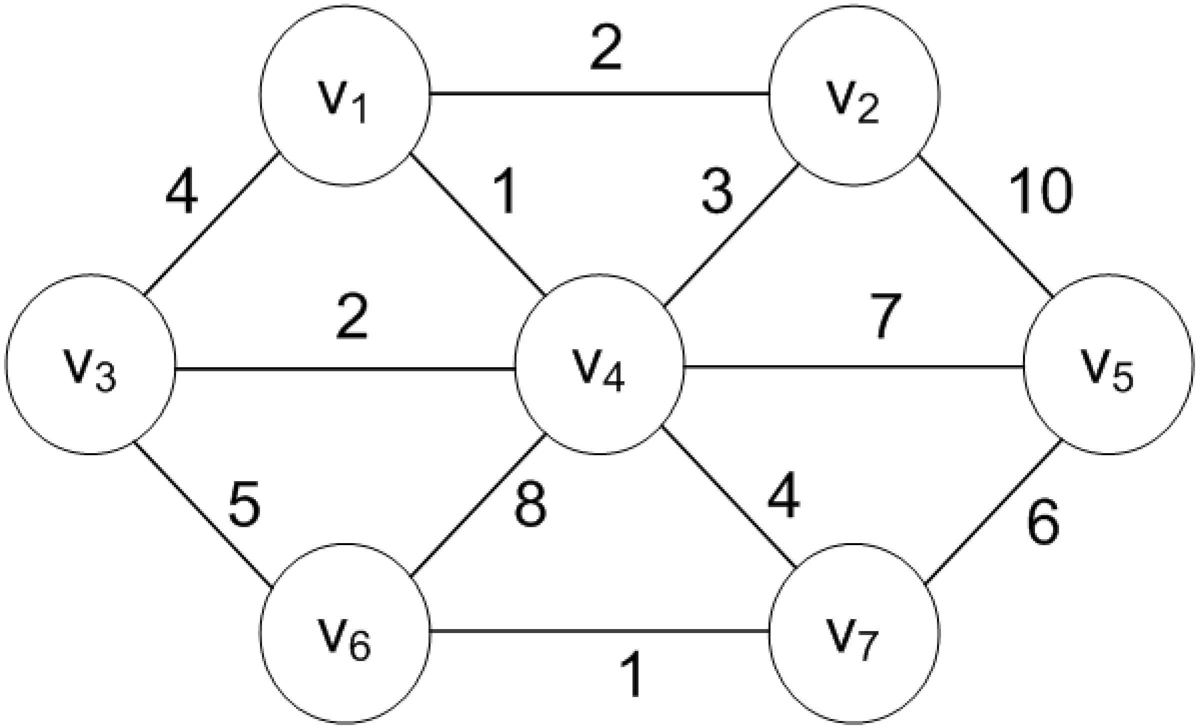
On maintient les informations suivantes pour chaque nœud:

1. La distance de l'arête arrivant sur *v* depuis le sommet parent (d_v);
2. Un booléen informant si le sommet est connu
3. Le parent à date du sommet *v* (p_v)

Une file de priorité est également utilisée

Algorithme de Prim

Reprenons notre exemple:



Algorithme de Prim

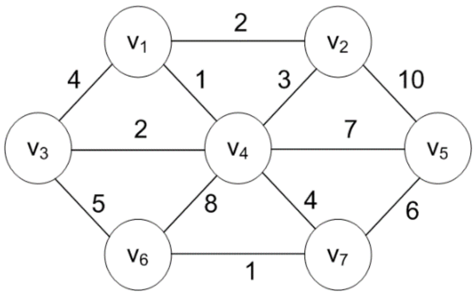
Nœuds	Distance	Connu?	Parent
V_1	∞	Faux	-
V_2	∞	Faux	-
V_3	∞	Faux	-
V_4	∞	Faux	-
V_5	∞	Faux	-
V_6	∞	Faux	-
V_7	∞	Faux	-

$\longrightarrow$

Nœuds	Distance	Connu?	Parent
V_1	0	Vrai	-
V_2	2	Faux	V_1
V_3	4	Faux	V_1
V_4	1	Faux	V_1
V_5	∞	Faux	-
V_6	∞	Faux	-
V_7	∞	Faux	-

File de priorité

Entre (V_1, 0)



File de priorité

Sort

(V_1, 0)

Entrent

(V_2, 2)
(V_3, 4)
(V_4, 1)

Algorithme de Prim

Nœuds	Distance	Connu?	Parent
V_1	0	Vrai	-
V_2	2	Faux	V_1
V_3	**2**	**Faux**	**V_4**
V_4	1	Vrai	V_1
V_5	**7**	**Faux**	**V_4**
V_6	**8**	**Faux**	**V_4**
V_7	**4**	**Faux**	**V_4**

→

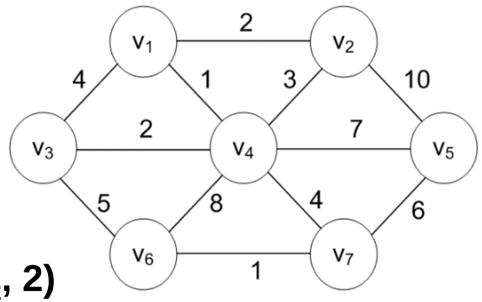
Nœuds	Distance	Connu?	Parent
V_1	0	Vrai	-
V_2	2	Vrai	V_1
V_3	2	Faux	V_4
V_4	1	Vrai	V_1
V_5	7	Faux	V_4
V_6	8	Faux	V_4
V_7	4	Faux	V_4

File de priorité

Sort
(V_4, 1)

Entrent
(V_5, 7)
(V_6, 8)
(V_7, 4)

Inchangé **(V_2, 2)**
Change (V_3, 2)



File de priorité

Sort
(V_2, 2)

Inchangé (V_5, 7)

Algorithme de Prim

Nœuds	Distance	Connu?	Parent
V_1	0	Vrai	-
V_2	2	Vrai	V_1
V_3	2	Vrai	V_4
V_4	1	Vrai	V_1
V_5	7	Faux	V_4
V_6	**5**	**Faux**	**V_3**
V_7	4	Faux	V_4

→

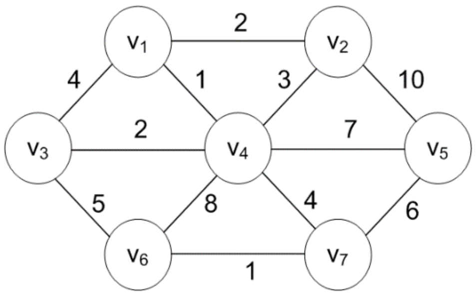
Nœuds	Distance	Connu?	Parent
V_1	0	Vrai	-
V_2	2	Vrai	V_1
V_3	2	Vrai	V_4
V_4	1	Vrai	V_1
V_5	**6**	**Faux**	**V_7**
V_6	**1**	**Faux**	**V_7**
V_7	4	Vrai	V_4

File de priorité

Sort

(V_3, 3)

Change (V_6, 5)



File de priorité

Sort

(V_7, 4)

Change (V_6, 1)

(V_5, 6)

Algorithme de Prim

Nœuds	Distance	Connu?	Parent
V_1	0	Vrai	-
V_2	2	Vrai	V_1
V_3	2	Vrai	V_4
V_4	1	Vrai	V_1
V_5	6	Faux	V_7
V_6	1	Vrai	V_7
V_7	4	Vrai	V_4

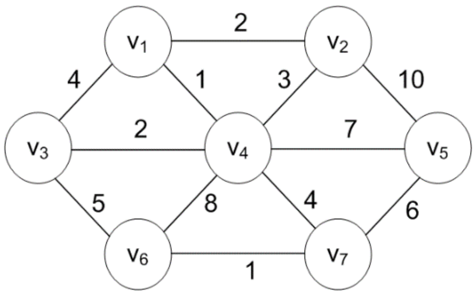
→

Nœuds	Distance	Connu?	Parent
V_1	0	Vrai	-
V_2	2	Vrai	V_1
V_3	2	Vrai	V_4
V_4	1	Vrai	V_1
V_5	6	Vrai	V_7
V_6	1	Vrai	V_7
V_7	4	Vrai	V_4

File de priorité

Sort

(V_6, 1)



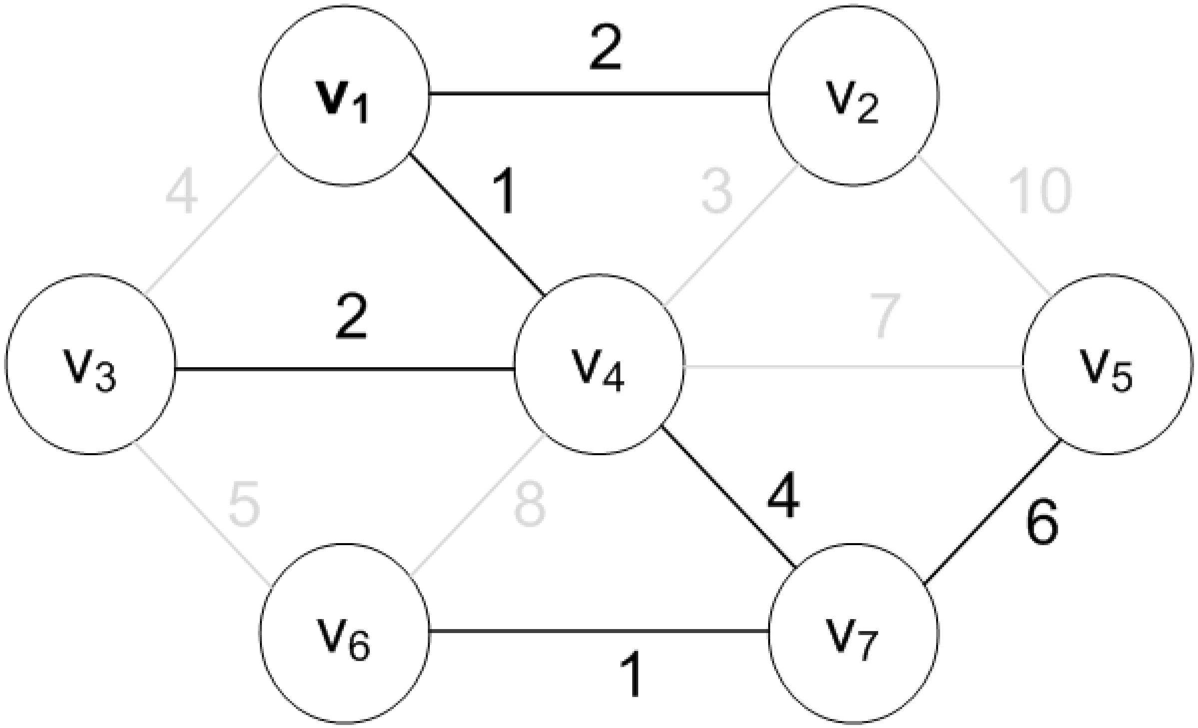
File de priorité

Sort

(V_5, 6)

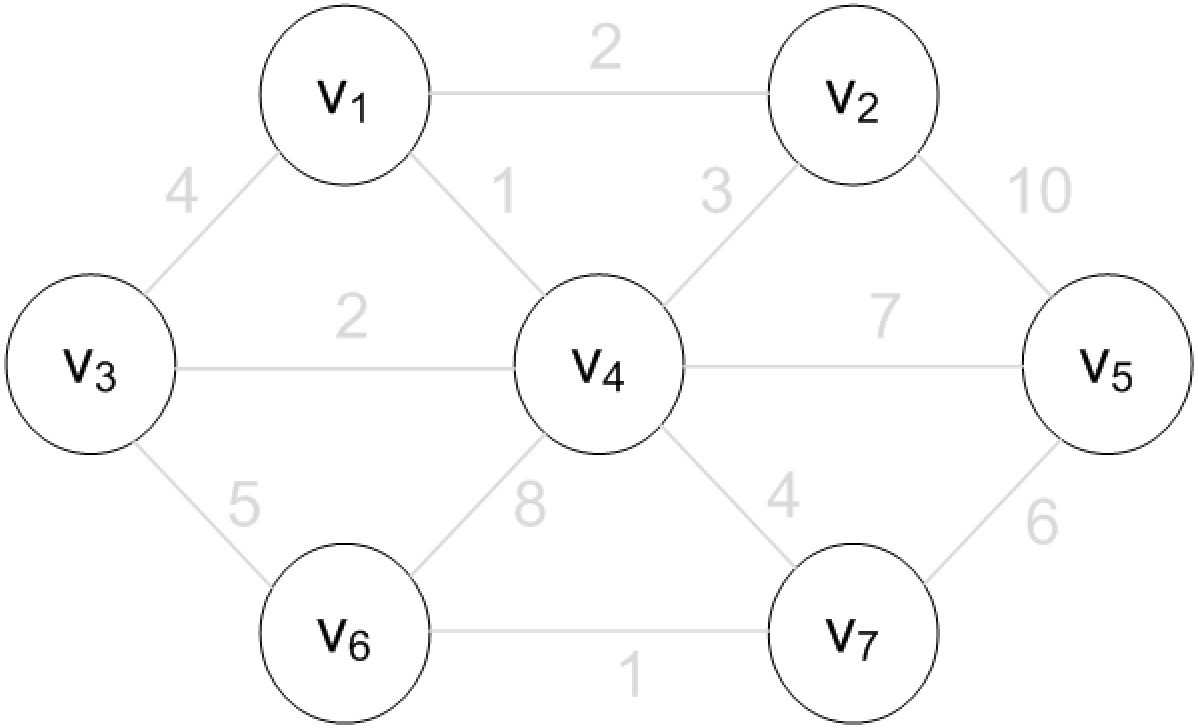
Algorithme de Prim

Et on parvient à la solution:



Algorithme de Prim

Quel résultat aurions-nous si on partait de v_5 ?



Graphes II

Algorithme de Kruskal

Définition

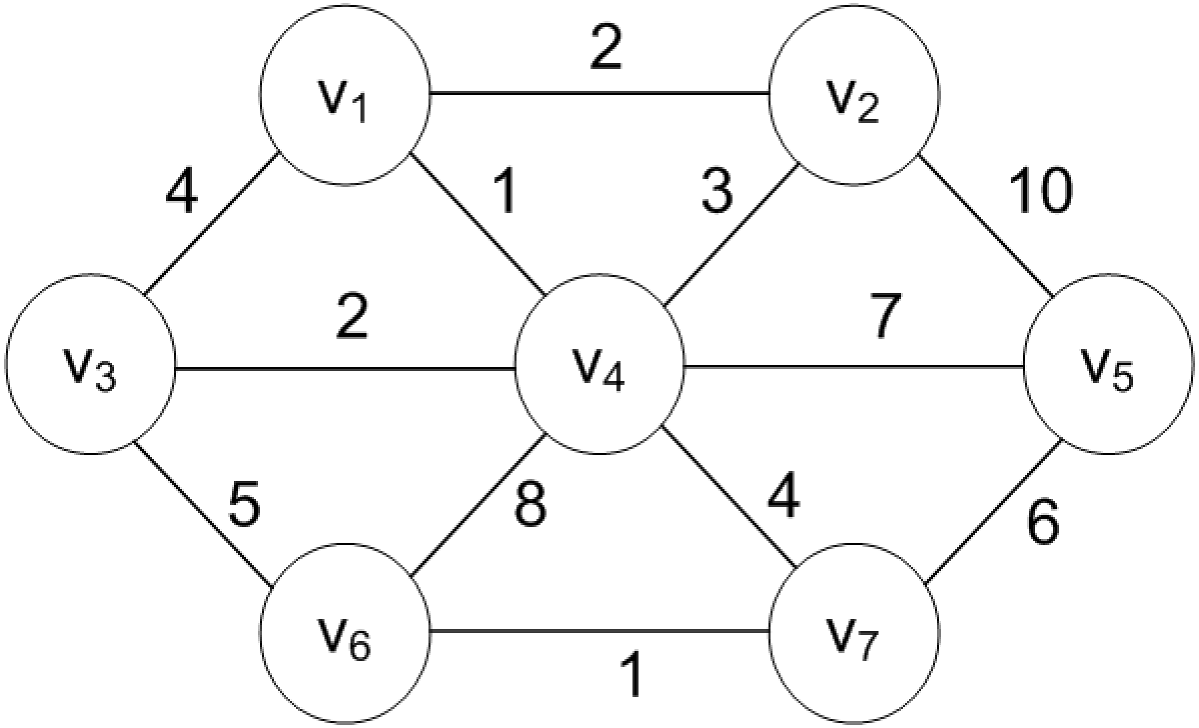
L'algorithme de Kruskal permet de trouver l'arbre sous-tendant minimum d'un graphe et s'exprime comme suit:

1. Créer une forêt **F** à partir des sommets du graphe
2. Créer une collection **A** contenant tous les arcs du graphe

Tant que **A** n'est pas vide et que **F** contient plus d'un arbre
    Retirer de **A** l'arc le plus petit poids
    Si l'arc lie deux arbres différents dans **F**,
        Union des deux arbres

Algorithme de Kruskal

Considérons le graphe valué et non dirigé suivant, pour lequel on cherche l'arbre sous-tendant minimum:



Algorithme de Kruskal

On cherche à obtenir ceci: