Algorithmes gloutons

3I003 - Algorithmique
Licence d'Informatique

Chargés de cours : Fanny Pascual, Olivier Spanjaard

Algorithme glouton

Algorithme glouton : Algorithme qui effectue une suite de choix, tels qu'à chaque étape un choix "localement" optimal est effectué et n'est pas remis en question par la suite.

Exemples :

- Algorithme de Dijkstra : à chaque étape le sommet ouvert le plus proche de la racine est ajouté à l'arborescence courante.

- Algorithme de Prim : à chaque étape le sommet ouvert le plus proche de l'arbre courant est ajouté à l'arbre.

Algorithme glouton

Pour certains problèmes, il existe des algorithmes gloutons qui retournent des solutions optimales. On a alors les propriétés :

Propriété de choix glouton : il existe toujours une solution optimale qui contient un premier choix glouton.
$\rightarrow$ On peut toujours arriver à une solution optimale en faisant un choix localement optimal.

Propriété de sous-structure optimale : trouver une solution optimale contenant le premier choix glouton se réduit à trouver une solution optimale pour un sous-problème de même nature.

Suite du cours : exemples

- Arbre couvrant de coût minimum : algorithme de Kruskal

- Compression de textes : algorithme de Huffman

- Ordonnancement d’intervalles : algorithme glouton

Algorithme de Kruskal

Joseph Kruskal (1928 - 2010)

Mathématicien ,
chercheur en informatique,
et psychométricien.

Travaille aux laboratoires Bell.

Algorithme de Kruskal

Principe de l'algorithme :
(H est un arbre couvrant de coût minimum à la terminaison de l'algorithme)

Algorithme de Kruskal

Trier les arêtes par coût croissant;

H = arbre vide;

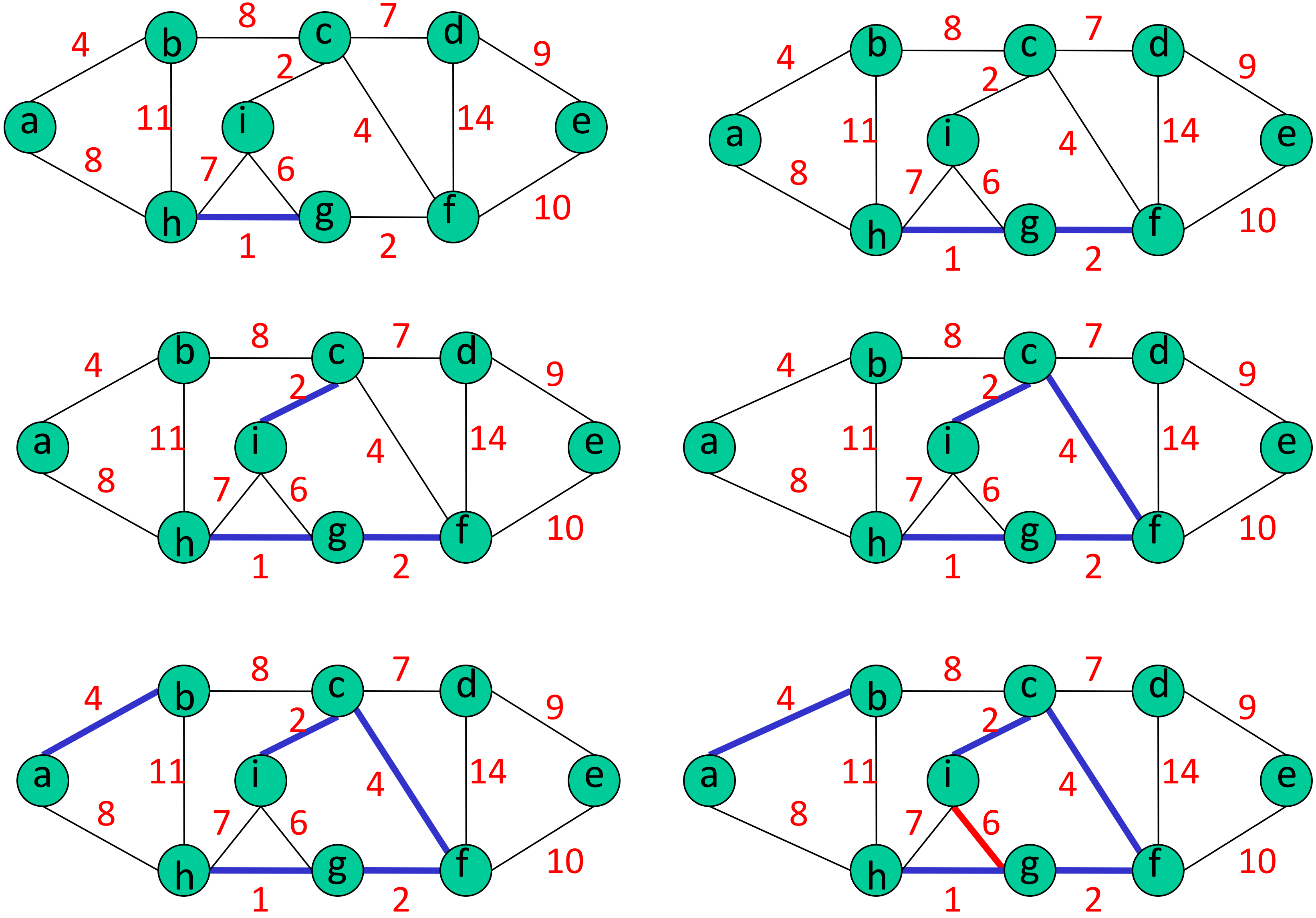
Examiner dans l'ordre chacune des arêtes {x,y} :

S'il n'existe pas de chaîne de x à y dans H :

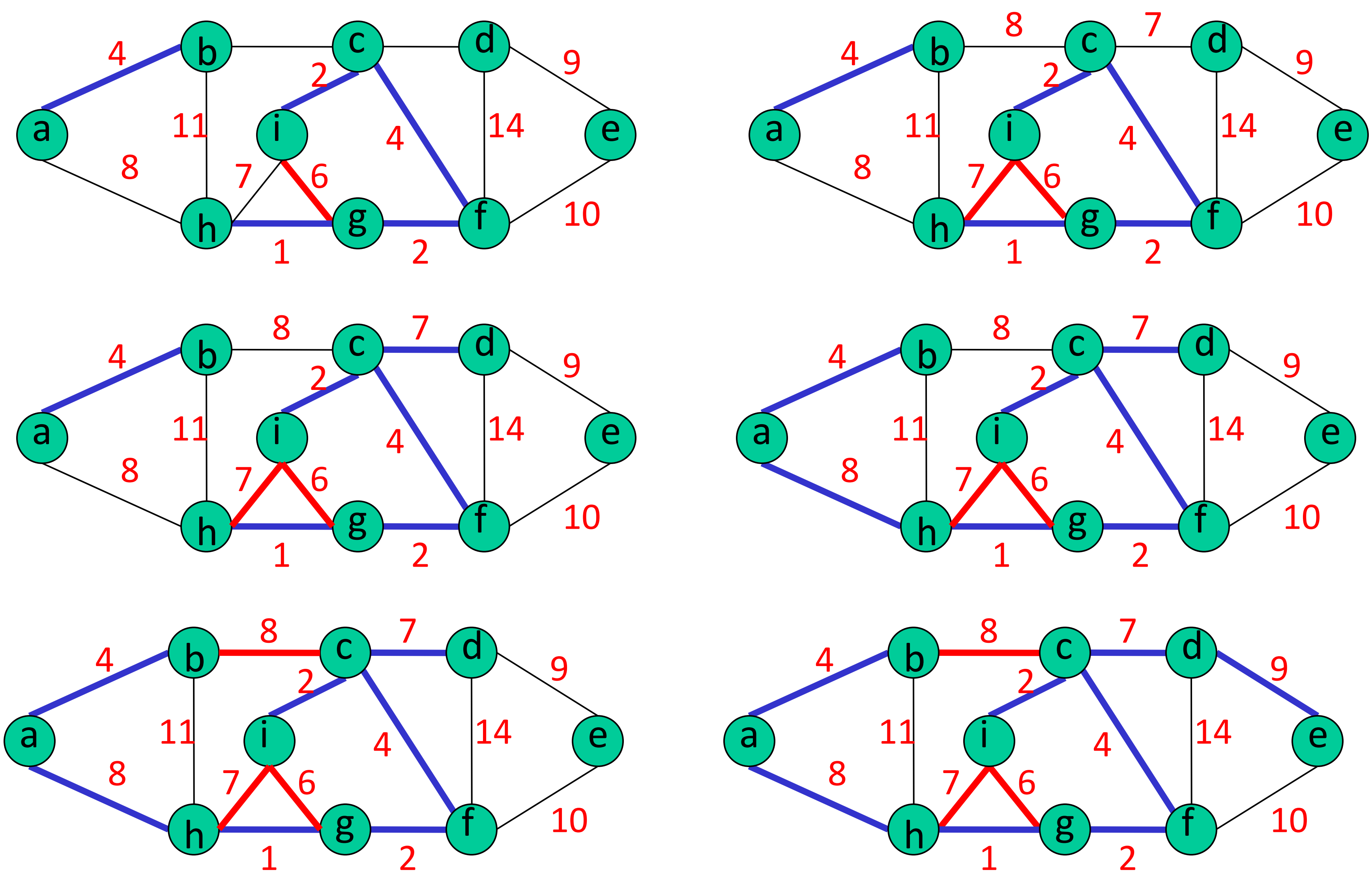
H = H U {x,y}

FinSi

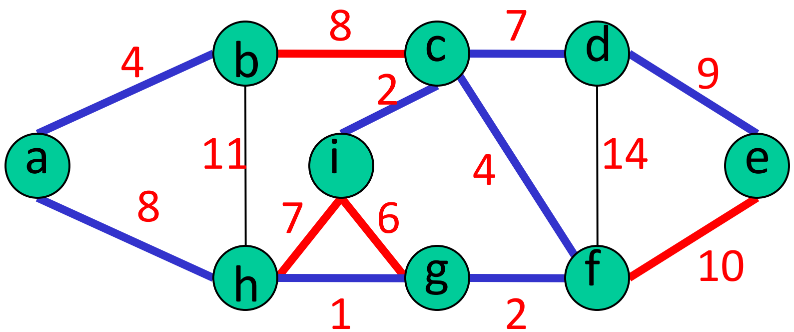
Une exécution de l'algorithme de Kruskal

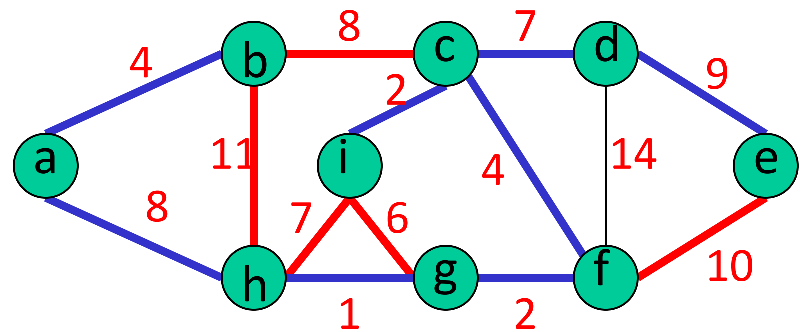


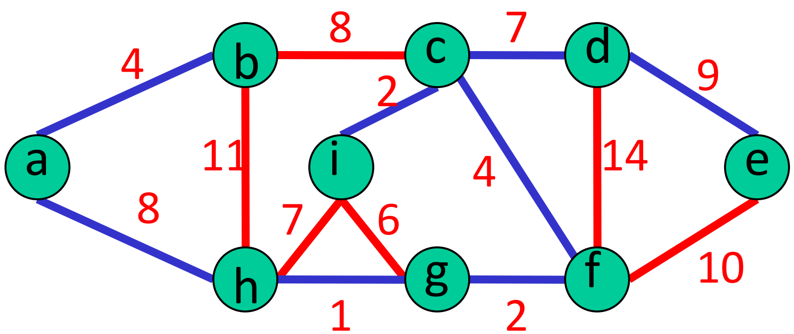
Une exécution de l'algorithme de Kruskal (suite)



Une exécution de l'algorithme de Kruskal (fin)





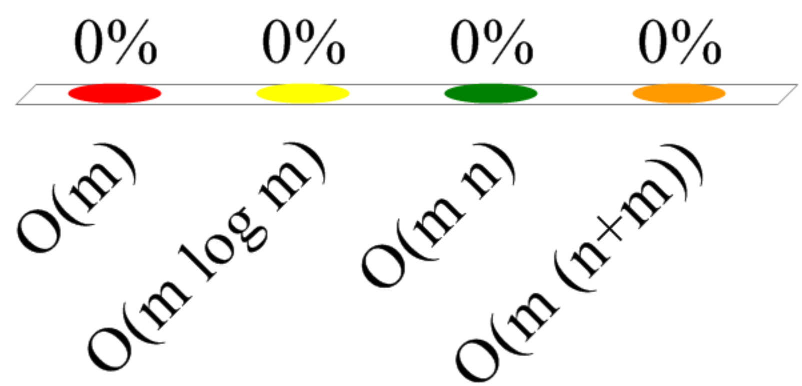


Quiz

Quelle est la complexité de l’algorithme de Kruskal ?

Algorithme de Kruskal:
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner l’arête {x,y} :
s’il n’existe pas de chaîne de x à y dans H
H = H U {x,y}
FinSi

A. O(m)
B. O(m log m)
C. O(m n)
D. O(m (n+m))



Complexité de l'algorithme de Kruskal

Algorithme de Kruskal

Trier les arêtes par coût croissant;

H = arbre vide;

Examiner dans l'ordre chacune des arêtes {x,y} :

S'il n'existe pas de chaîne de x à y dans H :

H = H U {x,y}

FinSi

Trier (ensemble des arêtes) se fait en O(m log m)
Soit α la complexité de la comparaison
« Composante connexe (x) = composante connexe (y) »

La complexité de l'algorithme est $O(m \log m + m\,\alpha)$

Montrons que la complexité est en $O(m \log m)$

Union-Find

On souhaite travailler sur les partitions de E={1, …, n}

Exemple :

Partition = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }

classe d'équivalence 1 2 3 4

Etant donné une partition, on souhaite

- Trouver la classe d'équivalence d'un élément : méthode Find
- Fusionner deux classes d'équivalence : méthode Union

Dans l'algorithme de Kruskal : E = ensemble des sommets.

- Deux sommets x et y sont dans une même composante connexe si Find(x)= Find(y)
- Si on choisit l'arête {x,y}, on fusionne deux composantes connexes en une : Union(x,y)

Algorithme de Kruskal avec Union-Find

```
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l’ordre chacune des arêtes {x,y} :
   s’il n’existe pas de chaîne de x à y dans H
       H = H U {x,y}
   FinSi
```

```
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l’ordre chacune des arêtes {x,y} :
  si Find(x) ≠ Find(y)
        H = H U {x,y}
        Union(x,y)
  FinSi
```

Une solution peu efficace

On représente la partition par un tableau tab tel que tab[i] est le numéro de la classe d'équivalence de l'élément i.

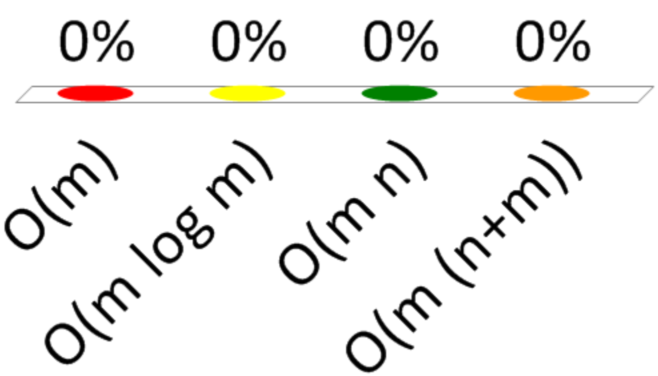
P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }
classe 1 2 3 4

1	2	3	4	5	6	7	8	9	10
1	**3**	**1**	**2**	**4**	**4**	**3**	**4**	**3**	**4**

Quiz Quelle est la complexité de l'algorithme de Kruskal si on code Union-Find avec ce tableau ?

```
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l'ordre chacune des arêtes {x,y}
    si Find(x) ≠ Find(y)
        H = H U {x,y};  Union(x,y)
    FinSi
```

A. O(m)
B. O(m log m)
C. O(m n)
D. O(m (n+m))



Une solution peu efficace

On représente la partition par un tableau tab tel que tab[i] est le numéro de la classe d'équivalence de l'élément i.

P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }
classe 1 2 3 4

1	2	3	4	5	6	7	8	9	10
1	**3**	**1**	**2**	**4**	**4**	**3**	**4**	**3**	**4**

Find : complexité en $O(1)$

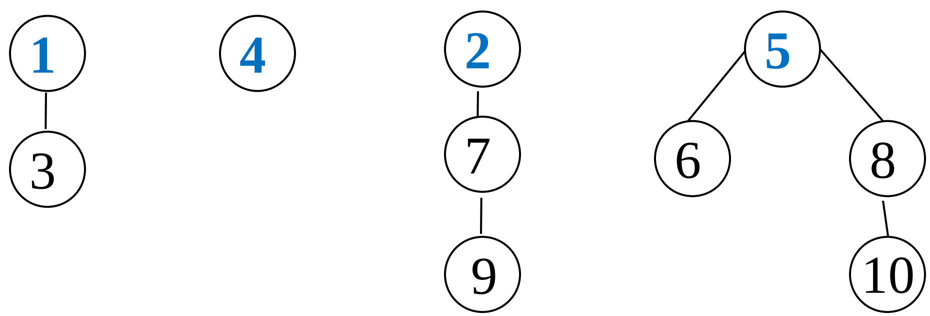
Union : complexité en $O(n)$

Une meilleure solution

On représente la partition par une forêt.
- Chaque arbre correspond à une classe d'équivalence.
- La racine de chaque arbre est le "représentant" de la classe.

On représente la forêt par un tableau père, tel que père[i] est le père de l'élément i, en posant père[i]=i pour une racine.

P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }

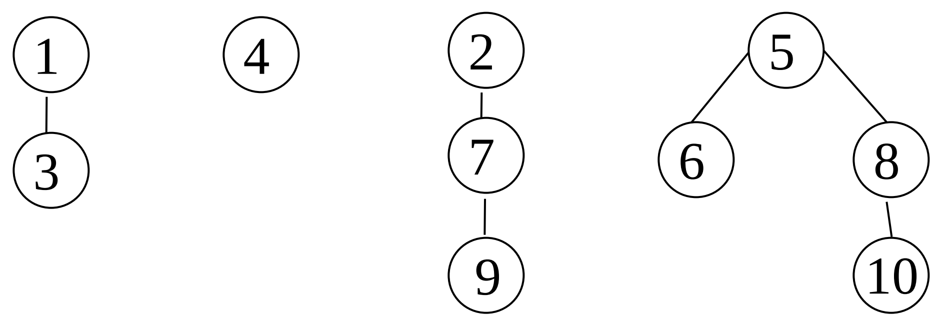


	1	2	3	4	5	6	7	8	9	10
père	**1**	**2**	**1**	**4**	**5**	**5**	**2**	**5**	**7**	**8**

Find

```
fonction Find(entier x): entier;
Tant que (x ≠ père[x])
        x := père[x];
FinTantque
Retourner(x)
```

Exemple: Find(9) P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



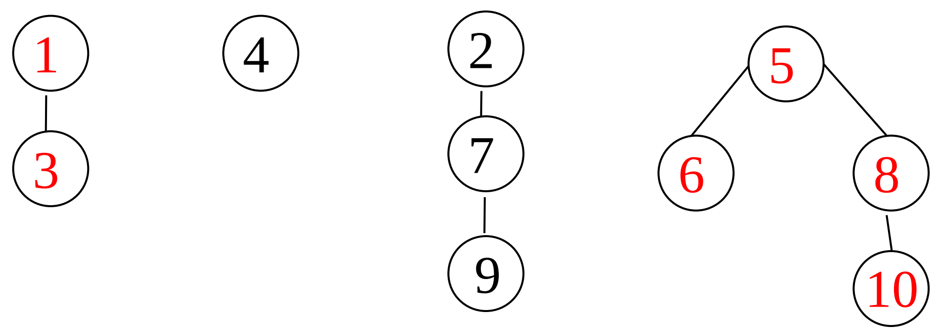
	1	2	3	4	5	6	7	8	9	10
père	**1**	**2**	**1**	**4**	**5**	**5**	**2**	**5**	**7**	**8**

père[9]=7 ; père[7]=2; père[2]=2; retourner(2)

Union

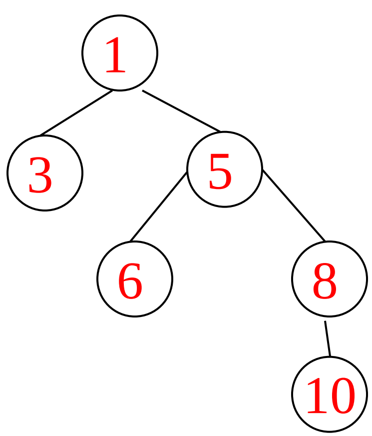
```
procédure Union (entier x, entier y): entier;
r1= Find(x);
r2 = Find(y);
Si (r1 ≠ r2) alors
        père[r2] = r1;
FinSi
```

Exemple: Union(3,8)   P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



Find(3)=1
Find(8)=5

père[5]=1

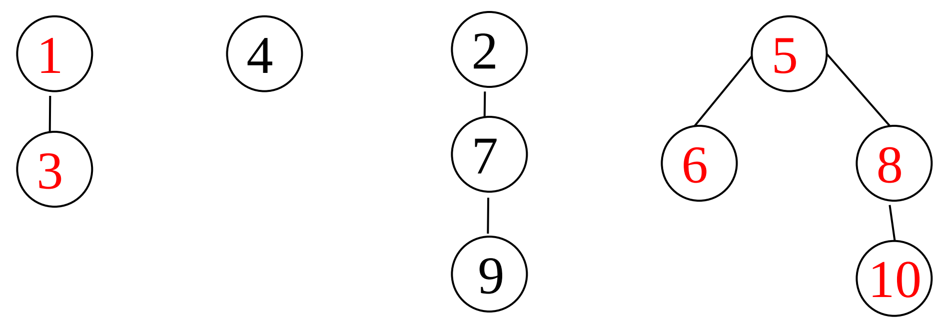


L'opération Find s'effectue en temps O(h), où h est la hauteur de l'arbre. Pire cas : h= O(n)

Union pondérée

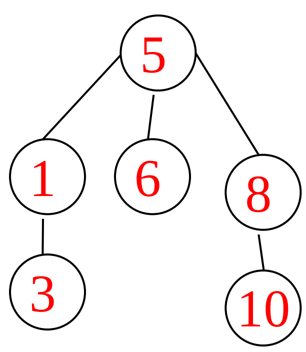
Lors de l'union de deux arbres, la racine de l'arbre avec le moins de sommets devient fils de la racine de l'autre.

Exemple: Union(3,8) P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



Find(3)=1
Find(8)=5

père[1]=5



Union pondérée

On maintient un tableau taille (initialisé à 1 si l'on part de classes singleton) : si i est une racine, taille[i] est le nombre de sommets de l'arbre de racine i.

```
procédure Union (entier x, entier y): entier;
r1= Find(x);
r2 = Find(y);
Si (r1  ≠ r2) alors
        Si (taille[r1])>taille[r2]) alors
                père[r2] := r1;
                taille[r1] :=taille[r1]+taille[r2]
        sinon
                père[r1] := r2;
                taille[r2] :=taille[r1]+taille[r2]
        FinSi
FinSi
```

Union pondérée

La hauteur d'un arbre à n sommets créé par une suite d'unions pondérées est $\leq 1 + \lfloor \log_2(n) \rfloor$

Preuve: Par récurrence sur n

- *Cas de base :* vrai pour n=1.

-*Induction :*
Soit T un arbre obtenu par union pondérée d'un arbre à x sommets (avec $1 \leq x \leq n/2$) et d'un arbre à (n-x) sommets.
Hauteur(T) $\leq \max(1 + \lfloor \log_2(n-x) \rfloor, 2 + \lfloor \log_2(x) \rfloor)$.

Or $\log_2(n-x) \leq \log_2(n)$
et $\log_2(x) \leq \log_2(n/2) \leq \log_2(n)-1$
La hauteur de T est donc majorée par $1 + \lfloor \log_2(n) \rfloor$

Union et Find sont donc en O(log n)

Point sur la complexité de l'algorithme de Kruskal

Algorithme de Kruskal

Trier les arêtes par coût croissant;

H = arbre vide;

Examiner dans l'ordre chacune des arêtes {x,y} :

S'il n'existe pas de chaîne de x à y dans H :

H = H U {x,y}

FinSi

Union et Find sont en O(log n)

Complexité de Kruskal : O(m log m + m log n)= O(m log m)

Et si les coûts des arêtes sont faibles (en O(m)), de façon à ce que Trier(ensemble des arêtes) se fasse en O(m)?

On cherche à améliorer la complexité de Union et Find.

Deuxième amélioration : compresser les chemins

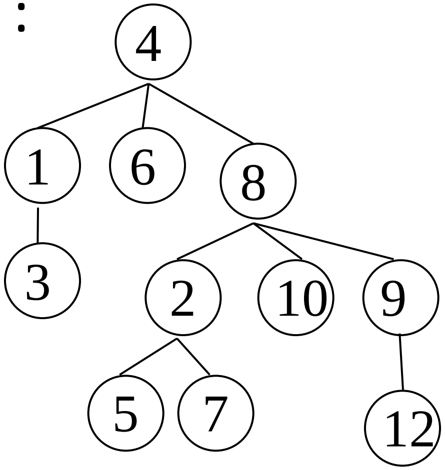
Quand on remonte du sommet x vers sa racine r, on refait le parcours en faisant de chaque sommet rencontré un fils de r.

```
fonction Findsimple(entier x):
entier;
Tant que (x ≠père[x])
          x := père[x];
FinTantque
Retourner(x)
```

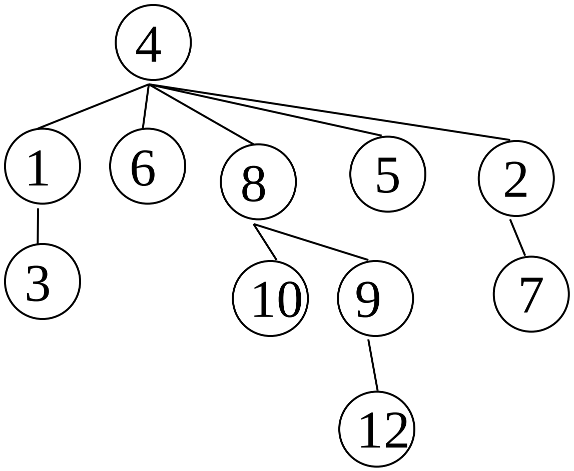
```
fonction Find(entier x): entier;
r :=  Findsimple(x);
Tant que (x ≠ père[x])
          y := père[x]
          père[x]:=r;
          x:=y;
FinTantque
Retourner(x)
```

Exemple : Find(5)

Arbre initial :



Arbre final :



Complexité de Union et Find ?

Complexité amortie : n opérations Union + m opérations Find se réalisent en temps $O(n + m\,\alpha(n,m))$, où α est une fonction qui croît très très lentement.

En effet, $\alpha(n,m)$ est la réciproque de la fonction d'Ackermann qui croît extrêmement vite: on a $\alpha(n,m) \leq 4$ pour $n,m \leq 2^{2048}$.
En pratique, $\alpha(n,m)$ est donc une constante.

Complexité de l'algorithme de Kruskal ?

Si on suppose que Tri(arêtes) est effectué en temps O(m).

On a alors n operations Union et m operations Find pour n éléments : ceci se fait en $O(n + m\,\alpha(n,m))$.

Rappel: dans le cas général la complexité est en $O(m \log m)$.

Ordonnancement d’intervalles

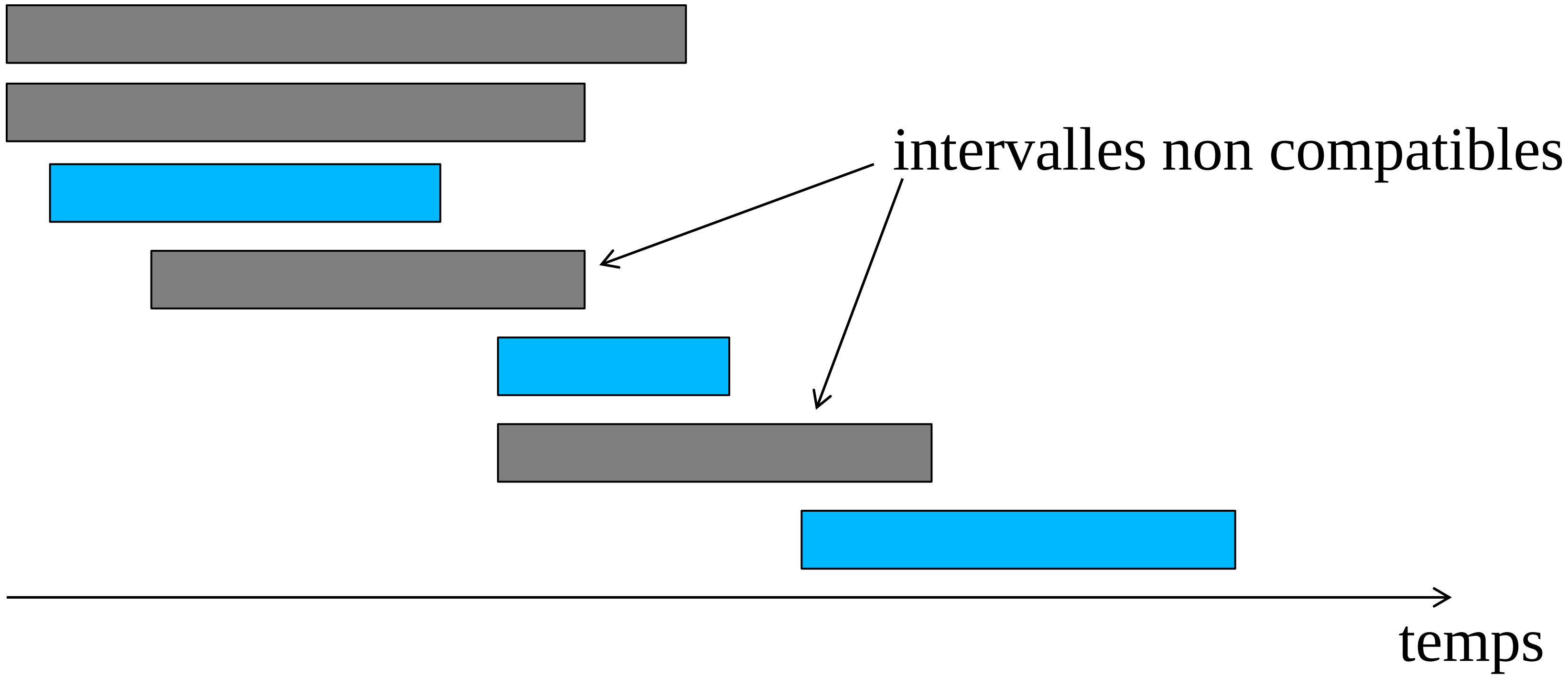
Application : Réservation de salle.

Une salle municipale peut être réservée par diverses associations. Chaque association indique un intervalle durant lequel elle souhaite disposer de la salle.

But : maximiser le nombre d’associations satisfaites.

Ordonnancement d'intervalles

- L'intervalle i commence en d_i et se termine en f_i.
- Deux intervalles sont compatibles s'ils ne s'intersectent pas.
- But : déterminer un sous-ensemble d'intervalles mutuellement compatibles de taille maximale.



Algorithmes gloutons

Algorithme générique :

- Examiner les intervalles dans un ordre spécifique.
- Prendre chaque intervalle dans cet ordre s'il est compatible avec les intervalles déjà pris.

- L'intervalle le plus court d'abord : on considère les intervalles par ordre de (f_i - d_i) croissant.

- L'intervalle qui a le moins de conflits d'abord : pour chaque intervalle i, soit c_i le nombre d'intervalles avec lesquels il est en conflit ; on considère les intervalles par ordre de c_i croissant.

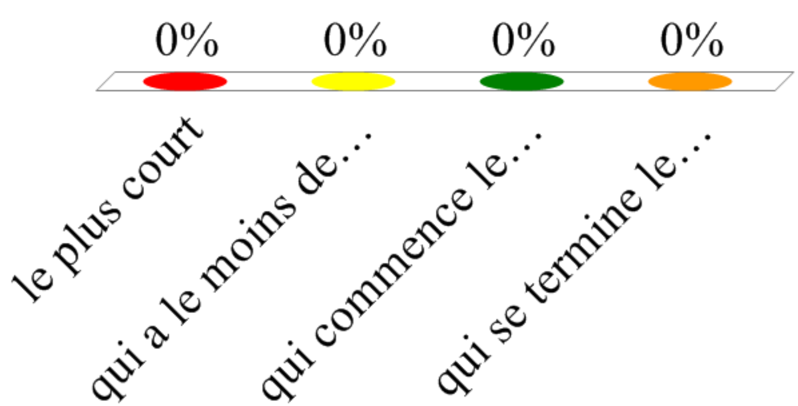
- L'intervalle qui commence le plus tôt d'abord : on considère les intervalles par ordre de d_i croissant.

- l'intervalle qui se termine le plus tôt d'abord : on considère les intervalles par ordre de f_i croissant.

Quiz Quel algorithme glouton est optimal ?

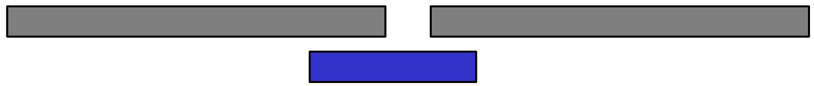
Il s'agit de l'algorithme glouton qui sélectionne d'abord l'intervalle …

A. le plus court
B. qui a le moins de conflits
C. qui commence le plus tôt
D. qui se termine le plus tôt

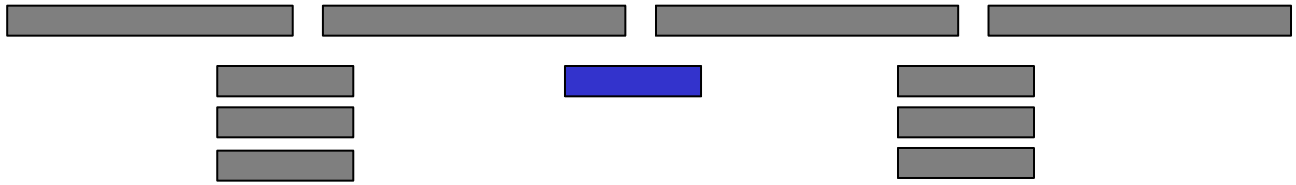


Algorithmes gloutons

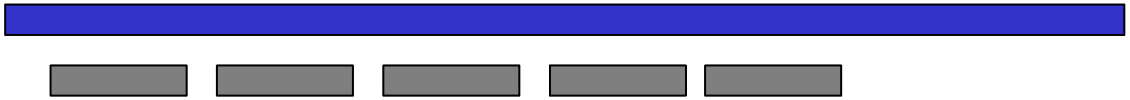
- L'intervalle le plus court d'abord : contre-exemple



- L'intervalle qui a le moins de conflits d'abord : contre-exemple



- L'intervalle qui commence le plus tôt d'abord : contre-exemple



Celui qui se termine le plus tôt d'abord

PlusPetiteDateDeFinDabord (n, d_1 ,d_2 , … , d_n , f_1 , f_2 , … , f_n)

Trier les intervalles par dates de fin t.q. $f_1 \leq f_2 \leq _{\dots} \leq f_n$
IntervallesChoisis := $\varnothing$

Pour i allant de 1 à n
    Si i est compatible avec IntervallesChoisis
        IntervallesChoisis := IntervallesChoisis $\cup$ {i}

Retourner IntervallesChoisis

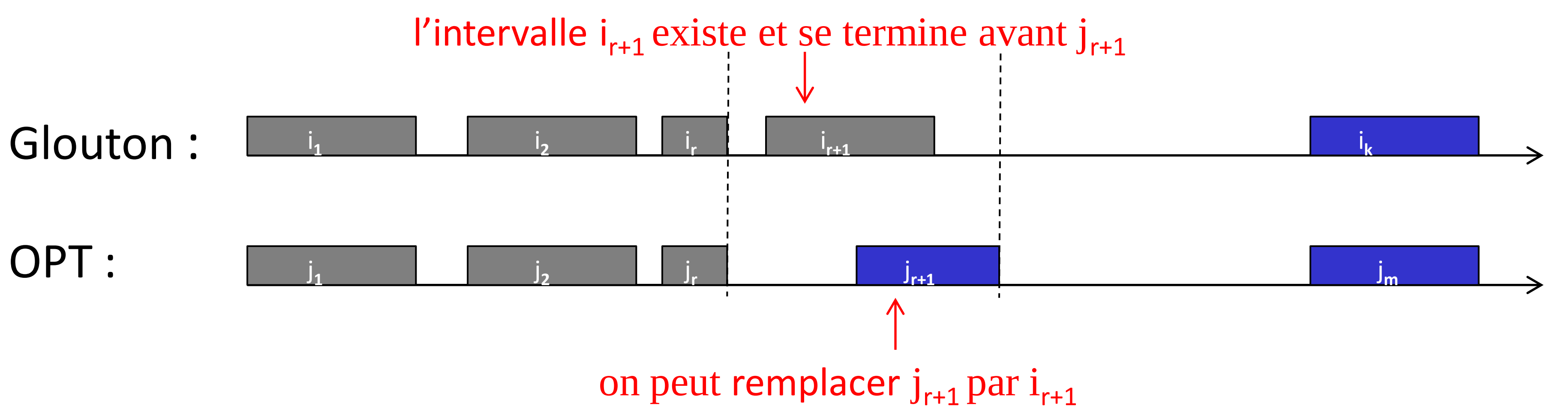
Propriétés :

- On garde en mémoire l'intervalle k qui a été ajouté en dernier à IntervallesChoisis.
- L'intervalle j est compatible avec IntervallesChoisis ssi $d_j \geq f_k$
- Algorithme en O(n log n) (complexité du tri)

Analyse de l'algorithme

Théorème : L'algorithme qui considère les intervalles par ordre des f_i croissant est optimal.

Preuve : par l'absurde.
- Supposons que cet algorithme ne soit pas optimal.
- Soit $i_1, i_2, \ldots, i_k$ les intervalles sélectionnés par cet algorithme.
- Soit $j_1, j_2, \ldots, j_m$ les intervalles sélectionnés par un algorithme optimal avec $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ avec r le plus grand possible.



Analyse de l'algorithme

Théorème : L'algorithme qui considère les intervalles par ordre des f_i croissant est optimal.

Preuve : par l'absurde.
- Supposons que cet algorithme ne soit pas optimal.
- Soit $i_1, i_2, \ldots, i_k$ les intervalles sélectionnés par cet algorithme.
- Soit $j_1, j_2, \ldots, j_m$ les intervalles sélectionnés par un algorithme optimal avec $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ avec r le plus grand possible.