

# Cours 6: Quelques utilisations des arbres

Olivier Bournez

bournez@lix.polytechnique.fr  
LIX, Ecole Polytechnique

INF421-b

Bases de la programmation et de l'algorithmique

# Aujourd'hui

Arbres de syntaxe abstraite

Union-Find

Arbres couvrant de poids minimal

Problème du voyageur de commerce

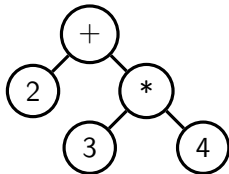
# Qu'est ce qu'une expression arithmétique?

- Une expression arithmétique est
  - ▶ soit un entier,
  - ▶ soit une opération  $e_1 \oplus e_2$  avec  $\oplus \in \{+, *, /, -\}$ , avec  $e_1, e_2$  des expressions arithmétiques.
- Autrement dit, une expression arithmétique est solution de l'équation réursive

$$E = \mathbb{Z} \uplus (E \times \{+, -, *, /\} \times E)$$

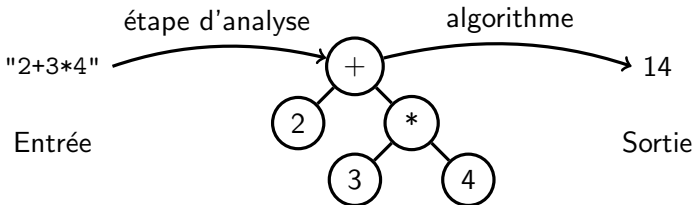
... c'est-à-dire, est un arbre.

- Exemple:  $2 + 3 * 4$  correspond à



# Arbre de syntaxe abstraite

- On a déjà vu comment évaluer une expression arithmétique:
  - ▶ à partir des arguments de la ligne de commande,
  - ▶ à partir d'une suite d'opérations rentrées une à une.
  - ▶ On aurait pu partir d'une chaîne de caractères comme "2+3\*4".
- Plutôt que de travailler directement sur ces entrées, on préfère souvent
  1. d'abord convertir l'entrée (ex: la chaîne de caractères) en un arbre,
  2. et travailler sur l'arbre.



# Digression: qu'est ce qu'un programme JAVA

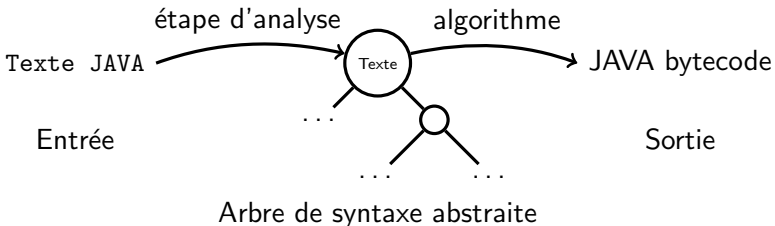
- Un programme .java = un "texte".
- Un "texte" =
  - ▶ une "déclaration de classe ou une instruction import" ("dcii"),
  - ▶ ou une "dcii" suivie d'un texte.
- Une "dcii" =
  - ▶ une "déclaration de classe",
  - ▶ ou une "instruction import".
- Une "déclaration de classe" =
  - ▶ un nom de classe <nom> suivi d'un "bloc d'instructions".
- Un "bloc d'instructions" = une "liste d'instructions".
- Une "liste d'instructions" =
  - ▶ une "instruction",
  - ▶ ou une "instruction" suivie d'une "liste d'instructions".
- Une "instruction" =
  - ▶ une "déclaration de classe",
  - ▶ ou une "déclaration de variable",
  - ▶ ou une "affectation".
- Une "déclaration de variable" =
  - ▶ Un "nom de type" suivi d'une "liste de variables+initialisations".

...
- Une "variable+initialisation" =
  - ▶ un "nom de variable",
  - ▶ un "nom de variable" suivi d'une "valeur" d'initialisation.

...
- Une "affectation" =
  - ▶ un "nom de variable" suivi d'une "valeur".

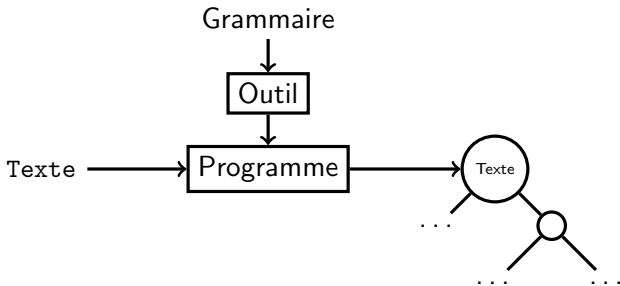
## Digression: Comment on compile

- Bref: un programme est un arbre.
- Comment fonctionne un compilateur JAVA:



## Pourquoi passer par les arbres de syntaxe abstraite?

- Parce que manipuler un arbre est facile.
- Mais aussi parce que l'on possède des outils qui transforment une description d'un langage (ce que l'on appelle *une grammaire*) en un programme qui produit un arbre à partir d'un texte conforme à cette grammaire.



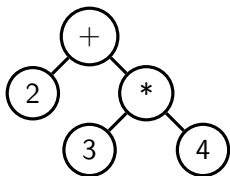
- Voir cours INF431.

## Retour sur les expressions arithmétiques

- Coder un arbre de syntaxe abstraite:

```
class Exp {  
    final static int INT=0, ADD=1, SUB=2, MUL=3, DIV=4 ;  
    int tag ;  
    int asInt ; // Utilisé si tag == INT  
    Exp e1, e2 ; // Utilisés si tag ∈ {ADD, SUB, MUL, DIV}  
  
    Exp(int i) { tag = INT ; asInt = i ; }  
  
    Exp(Exp e1, int op, Exp e2) {  
        tag = op; this.e1 = e1; this.e2 = e2; }  
}
```

- Construire un arbre de syntaxe abstraite:



```
Exp e= new Exp(new Exp(2),  
    Exp.ADD,  
    new Exp (new Exp(3),Exp.MUL,new Exp(4)));
```

# Construire un arbre de syntaxe abstraite

- En s'inspirant de la calculatrice en notation polonaise inverse

```
import java.util.* ;

class Traduit {
    public static void main (String [] args) {
        Stack<Exp> p = new Stack<Exp> () ;
        for (int i = 0 ; i < args.length ; i++) {
            String cmd = args[i] ;
            if (cmd.equals("+")) {
                Exp x = p.pop(); Exp y = p.pop();
                p.push(new Exp(y,Exp.ADD, x));
            }
            else if
                ... // Même chose pour *,/, -
            else
                p.push(new Exp(Integer.parseInt(cmd))) ; }
        Exp t = p.pop();
        // ici t est construit.
    }
}
```

# Evaluer un arbre

## ■ Evaluer un arbre

```
static int eval(Exp e) {  
    switch (e.tag) {  
        case Exp.INT: return e.asInt;  
        case Exp.ADD: return eval(e.e1)+eval(e.e2);  
        case Exp.SUB: return eval(e.e1)-eval(e.e2);  
        case Exp.MUL: return eval(e.e1)*eval(e.e2);  
        case Exp.DIV: return eval(e.e1)/eval(e.e2); }  
    throw new Error("tag incorrect"); }  
}
```

## ■ Remarque JAVA: sans le “**throw new** Error” le compilateur n’accepte pas, car il pourrait y avoir des cas sans **return**.

- ▶ on pourrait aussi renvoyer une valeur par défaut.

## Afficher un arbre

- Afficher un arbre en notation infixe:

```
static void Affiche(Exp e) {
switch (e.tag) {
case Exp.INT: System.out.print(e.asInt); return;
case Exp.ADD: case Exp.SUB:
    System.out.print('(');Affiche(e.e1);
    System.out.print(e.tag==Exp.ADD?'+' : '-'');
    Affiche(e.e2);System.out.print(')');
    return;
case Exp.MUL: case Exp.DIV:
    System.out.print('(');Affiche(e.e1);
    System.out.print(e.tag==Exp.MUL?'*' : '/'');
    Affiche(e.e2);System.out.print(')');}}}
```

Affiche:  $(1+(2*3))$

- Le poly propose une version qui évite les parenthèses inutiles.
- La notation polonaise inverse s'obtient par un parcours postfixe, et s'appelle donc aussi *notation postfixe*.

# Aujourd'hui

Arbres de syntaxe abstraite

**Union-Find**

Arbres couvrant de poids minimal

Problème du voyageur de commerce

# Travailler sur des partitions: Union-Find

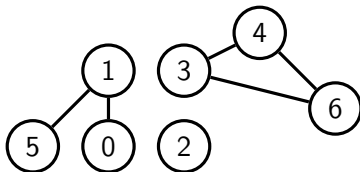
- On souhaite travailler sur les partitions de

$$E = \{0, 1, \dots, n - 1\}.$$

- Etant donnée une partition, on souhaite
  - ▶ trouver la classe d'équivalence d'un élément
    - méthode `find`.
  - ▶ fusionner deux classes d'équivalence
    - méthode `union`.

## Exemple d'application

- Comment déterminer les composantes connexes d'un graphe?

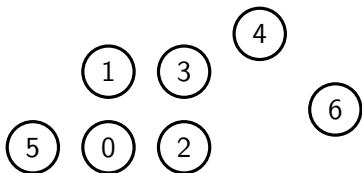


- Méthode:

- ▶ On part de la partition où toutes les classes d'équivalence sont des singletons.
- ▶ Pour chaque arête  $e = (u, v)$ ,
  - on fait `union(u, v)`.
- ▶ Deux sommets sont dans la même composante connexe si et seulement si `find(u) == find(v)`.

## Exemple d'application

- Comment déterminer les composantes connexes d'un graphe?

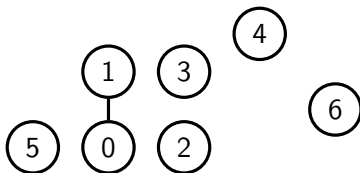


- Méthode:

- ▶ On part de la partition où toutes les classes d'équivalence sont des singletons.
- ▶ Pour chaque arête  $e = (u, v)$ ,
  - on fait `union(u, v)`.
- ▶ Deux sommets sont dans la même composante connexe si et seulement si `find(u) == find(v)`.

## Exemple d'application

- Comment déterminer les composantes connexes d'un graphe?

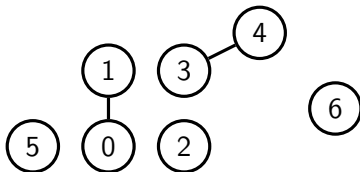


- Méthode:

- ▶ On part de la partition où toutes les classes d'équivalence sont des singletons.
- ▶ Pour chaque arête  $e = (u, v)$ ,
  - on fait `union(u, v)`.
- ▶ Deux sommets sont dans la même composante connexe si et seulement si `find(u) == find(v)`.

## Exemple d'application

- Comment déterminer les composantes connexes d'un graphe?

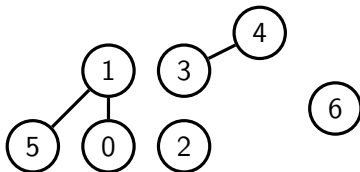


- Méthode:

- ▶ On part de la partition où toutes les classes d'équivalence sont des singletons.
- ▶ Pour chaque arête  $e = (u, v)$ ,
  - on fait `union(u, v)`.
- ▶ Deux sommets sont dans la même composante connexe si et seulement si `find(u) == find(v)`.

## Exemple d'application

- Comment déterminer les composantes connexes d'un graphe?

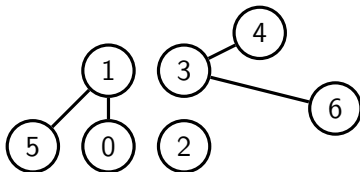


- Méthode:

- ▶ On part de la partition où toutes les classes d'équivalence sont des singletons.
- ▶ Pour chaque arête  $e = (u, v)$ ,
  - on fait `union(u, v)`.
- ▶ Deux sommets sont dans la même composante connexe si et seulement si `find(u) == find(v)`.

## Exemple d'application

- Comment déterminer les composantes connexes d'un graphe?

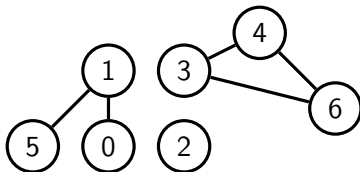


- Méthode:

- ▶ On part de la partition où toutes les classes d'équivalence sont des singletons.
- ▶ Pour chaque arête  $e = (u, v)$ ,
  - on fait `union(u, v)`.
- ▶ Deux sommets sont dans la même composante connexe si et seulement si `find(u) == find(v)`.

## Exemple d'application

- Comment déterminer les composantes connexes d'un graphe?



- Méthode:

- ▶ On part de la partition où toutes les classes d'équivalence sont des singletons.
- ▶ Pour chaque arête  $e = (u, v)$ ,
  - on fait `union(u, v)`.
- ▶ Deux sommets sont dans la même composante connexe si et seulement si `find(u) == find(v)`.

## Une solution inefficace

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

$$P = \{ \{2, 5, 8\}, \{0, 6\}, \{1\}, \{3, 4, 7, 9\} \}$$

classe    1            2            3            4

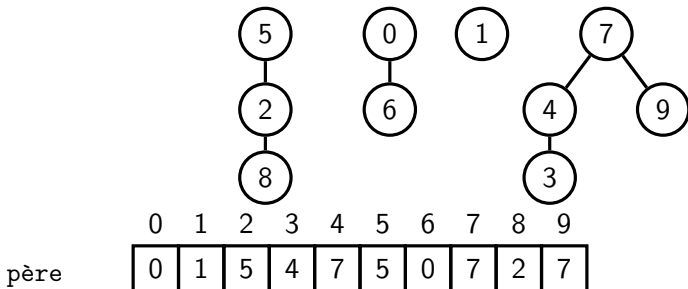
	0	1	2	3	4	5	6	7	8	9
tab	2	3	1	4	4	1	2	4	1	4

- Find: Temps  $O(1)$ .
- Union: Temps  $O(n)$ .

## Une solution plus rusée

- On représente la partition par une forêt.
  - ▶ Chaque arbre correspond à une classe d'équivalence.
  - ▶ La racine de chaque arbre donne un représentant canonique 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.

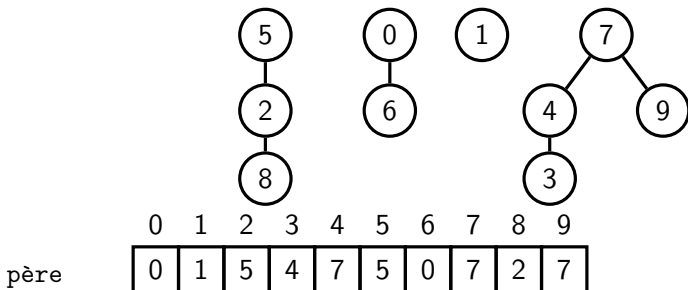
$$P = \{ \{2, 5, 8\}, \{0, 6\}, \{1\}, \{3, 4, 7, 9\} \}$$



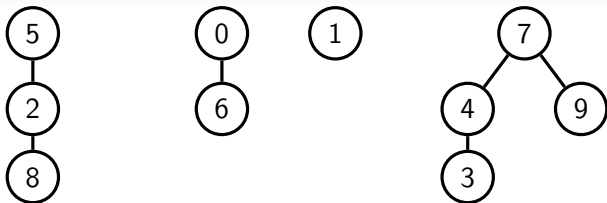
## Une solution plus rusée

- On représente la partition par une forêt.
  - ▶ Chaque arbre correspond à une classe d'équivalence.
  - ▶ La racine de chaque arbre donne un représentant canonique 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.
  - ▶ attention, cette représentation des arbres est rarement judicieuse hors du contexte union/find.

$$P = \{ \{2, 5, 8\} , \{0, 6\} , \{1\} , \{3, 4, 7, 9\} \}$$



## Union des classes de 2 et de 3

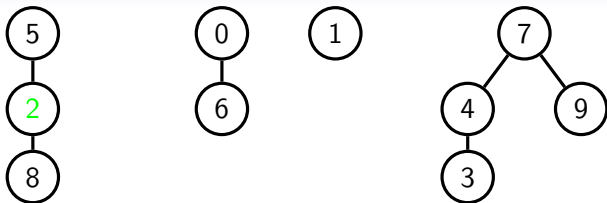


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

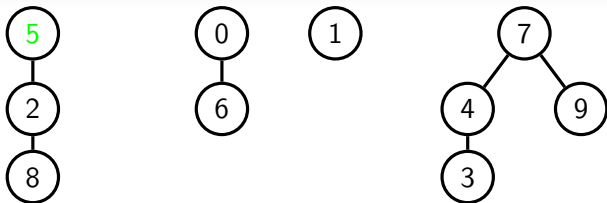


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

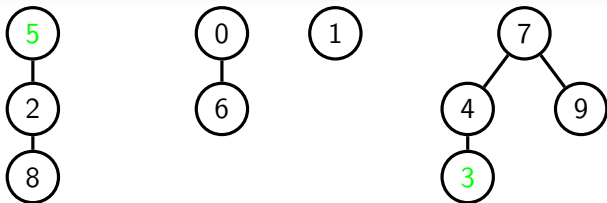


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

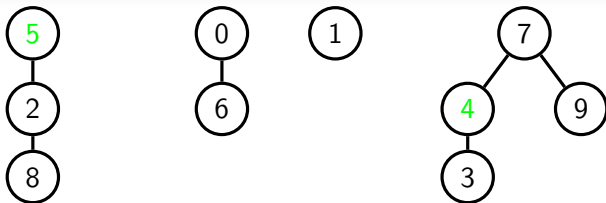


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

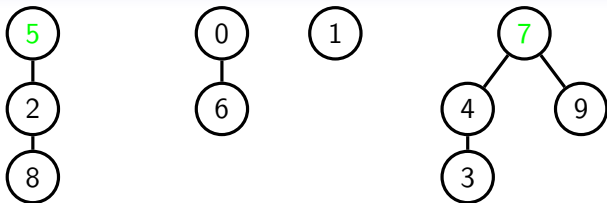


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

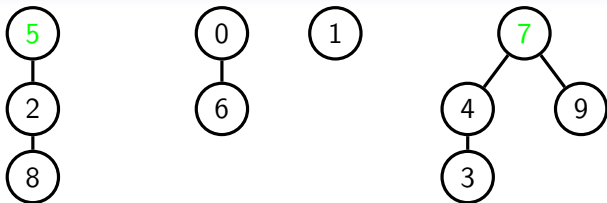


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

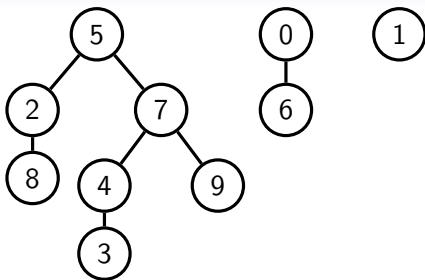


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

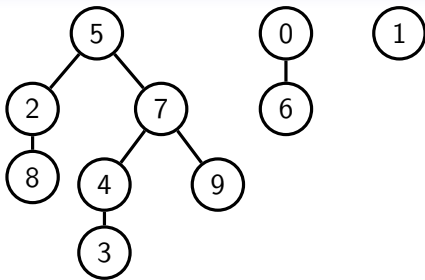


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3

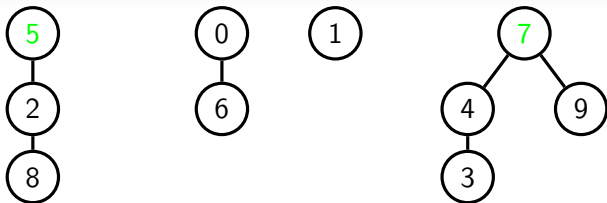


- `union(2,3)`
- Possibilité 1

```
int find(int x) {  
    while (x  $\neq$  père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r  $\neq$  s) père[s] = r;}
```

## Union des classes de 2 et de 3

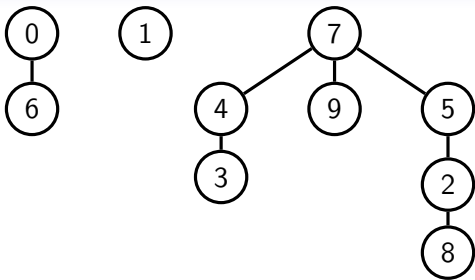


■ union(2,3)

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[s] = r;}
```

## Union des classes de 2 et de 3



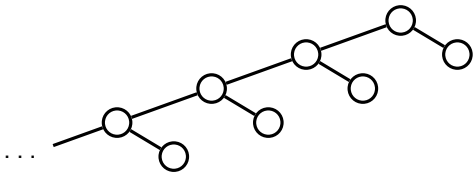
- union(2,3)
- Possibilité 2

```
int find(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
void union(int x, int y) {  
    int r = find(x);  
    int s = find(y);  
    if (r ≠ s) père[r] = s;}
```

## Heuristique 1: Union pondérée

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



- Heuristique 1: 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.

## 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`.

```
void union(int x, int y) {
    int r = find(x);
    int s = find(y);
    if (r == s) return;
    if (taille[r] > taille[s]) {
        père[s] = r;
        taille[r] += taille[s]; }
    else {
        père[r] = s;
        taille[s] += taille[r]; }}
```

## Union pondérée

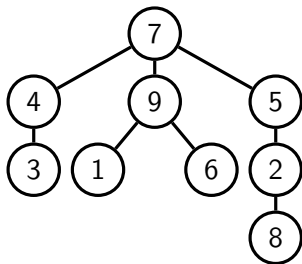
- La hauteur d'un arbre à  $n$  sommets créé par une suite d'unions pondérées est  $\leq 1 + \lfloor \log_2 n \rfloor$ .
- Preuve: Par récurrence sur  $n$ .
  - ▶ Vrai pour  $n = 1$ .
  - ▶ Pour un arbre obtenu par union pondérée d'un arbre à  $m$  sommets et d'un arbre à  $n - m$  sommets, avec  $1 \leq m \leq n/2$ , sa hauteur est majorée par

$$\max(1 + \lfloor \log_2(n - m) \rfloor, 2 + \lfloor \log_2 m \rfloor).$$

- ▶ Comme
  - $\log_2 m \leq \log_2(n/2) = \log_2(n) - 1$ ,
  - et  $\log_2(n - m) \leq \log_2 n$ ,la hauteur est bien majorée par  $1 + \lfloor \log_2 n \rfloor$ .
- Union et Find sont donc en  $O(\log n)$ .

## Heuristique 2: Compresser les chemins

- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)

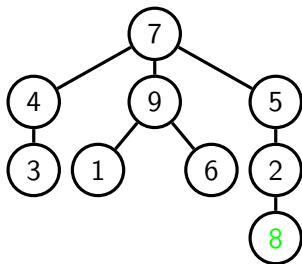


```
int findsimple(int x) {  
    while (x ≠ père[x])  
        x = père[x];  
    return x; }
```

```
int find(int x) {  
    int r = findsimple(x);  
    while (x ≠ père[x]) {  
        int y=père[x];  
        père[x]=r;  
        x = y;}  
    return x;}
```

## Heuristique 2: Compresser les chemins

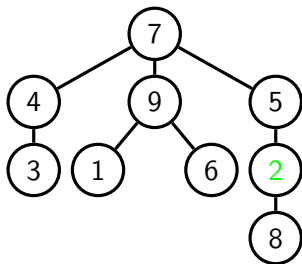
- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Heuristique 2: Compresser les chemins

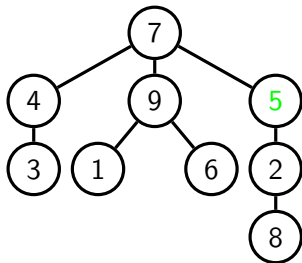
- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Heuristique 2: Compresser les chemins

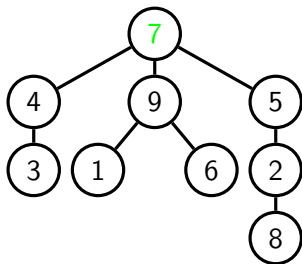
- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Heuristique 2: Compresser les chemins

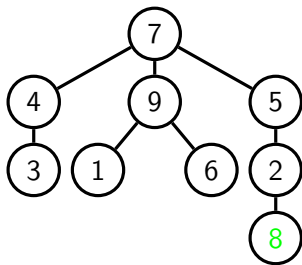
- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Heuristique 2: Compresser les chemins

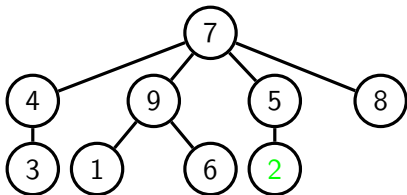
- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Heuristique 2: Compresser les chemins

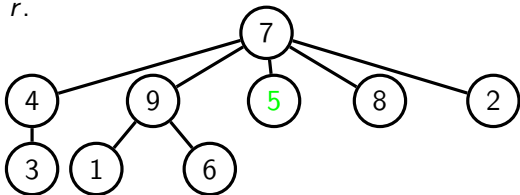
- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Heuristique 2: Compresser les chemins

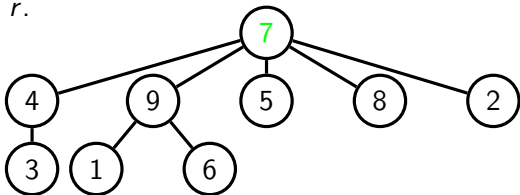
- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Heuristique 2: Compresser les chemins

- Heuristique 2: Lorsque l'on remonte du sommet  $x$  vers sa racine  $r$ , on refait le parcours en faisant de chaque sommet rencontré un fils de  $r$ .
- Exemple: Find(8)



```
int find(int x) {  
    if (x==père[x]) return x;  
    return père[x]=find(père[x]);  
}
```

## Complexité de Union/Find

- Théorème [Tarjan]  $m$  opérations union ou find sur  $n$  éléments, se réalisent en temps  $O(n + m\alpha(n))$  où  $\alpha$  est une fonction qui croît très très lentement.
- Aucun algorithme linéaire connu.
- On a  $\alpha(n) \leq 4$  pour  $n \leq 2^{2048}$ .
- (Le nombre d'atomes dans l'univers est estimé à  $10^{80}$ .)
- En pratique: c'est comme si c'était linéaire.

## Complément: définition précise de $\alpha$

- Soit  $A : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  la fonction définie par
  - ▶  $A_0(j) = j + 1$ ;
  - ▶  $A_k(j) = A_{k-1}^{[j+1]}(j)$  si  $k \geq 1$ .
- Dans cette définition,  $F^{[k]}$  désigne  $F$  itérée  $k$  fois:
  - ▶  $F^{[0]}(j) = j$ ,
  - ▶  $F^{[i+1]}(j) = F(F^{[i]}(j))$ .
- $A_k[j]$  est une fonction strictement croissante de  $j$  et de  $k$ .
- On a par exemple

$$A_3(1) = 2047.$$

$$A_4(1) \gg 2^{2048} \gg 10^{80}.$$

- $\alpha(n) = \min\{k \mid n \leq A_k(1)\}$ .

# Aujourd'hui

Arbres de syntaxe abstraite

Union-Find

**Arbres couvrant de poids minimal**

Problème du voyageur de commerce

# Arbre couvrant de poids minimal

- On se donne un graphe connexe  $G = (V, E)$ , avec un poids  $w(e)$  pour chaque arête  $e = (u, v) \in E$ .
- On veut calculer un arbre couvrant de  $G$  de poids minimal:
  - ▶ un *arbre couvrant* est un arbre (graphe connexe sans cycle)  $(V, E')$  avec  $E' \subset E$ .
  - ▶ Le poids d'un arbre est la somme des poids de ses arêtes.

## Exemple d'application

- On a des villes à connecter par un réseau.
- Le graphe:
  - ▶ ses sommets sont les villes;
  - ▶ il y a une arête entre toute paire de sommets (le graphe est dit *complet*);
  - ▶ les poids d'une arête est le coût de connection (ex: distance).
- Un arbre couvrant de poids minimal
  - ▶ une façon de connecter les villes
  - ▶ qui minimise le coût total.
- Remarque: si les poids sont des distances, on est dans un cas particulier, appelé *le cas euclidien*:
  - = les poids vérifient l'inégalité triangulaire.

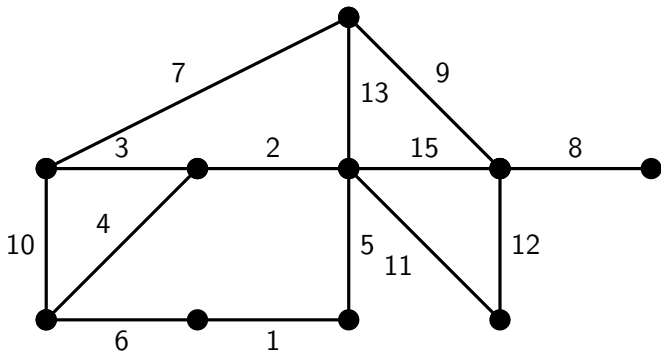
# Algorithme de Kruskal

Algorithme *Kruskal*( $G, w$ )

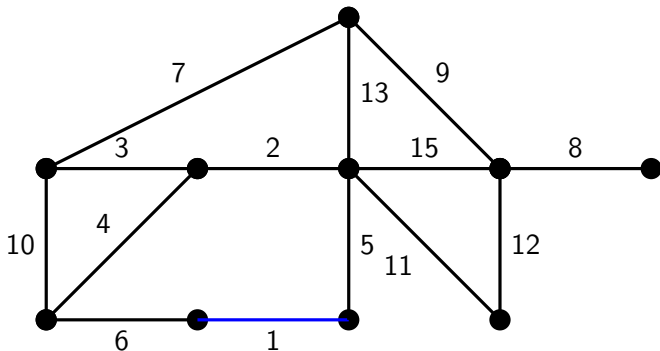
- Trier les arêtes par ordre de poids croissants  
 $w(e_1) \leq w(e_2) \leq \dots \leq w(e_{|E|})$
- $T := \emptyset$
- Pour  $i := 1$  à  $|E|$ 
  - ▶ si  $T \cup \{e_i\}$  est acyclique alors
    - $T := T \cup \{e_i\}$
- Renvoyer  $T$ .

Remarque: à chaque étape  $T$  est une forêt.

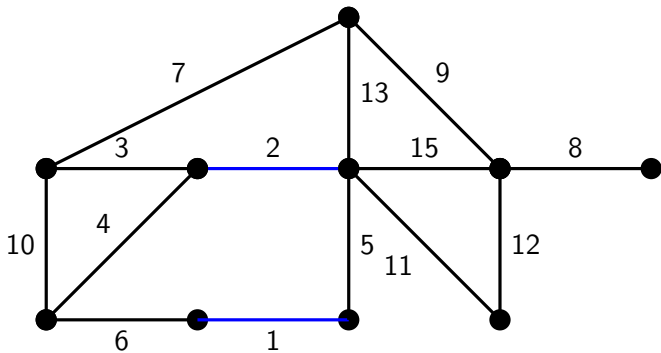
# Exemple



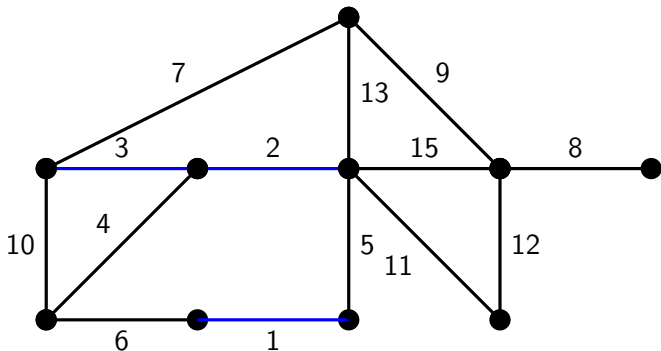
# Exemple



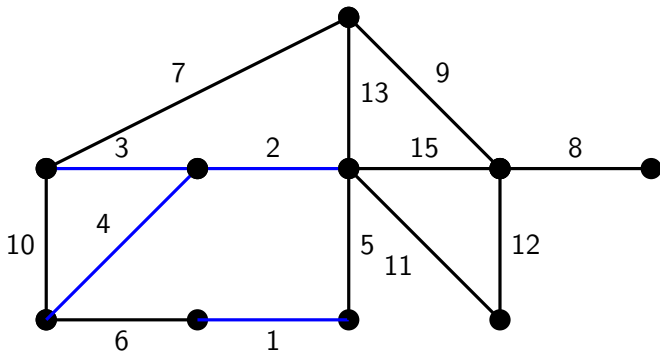
# Exemple



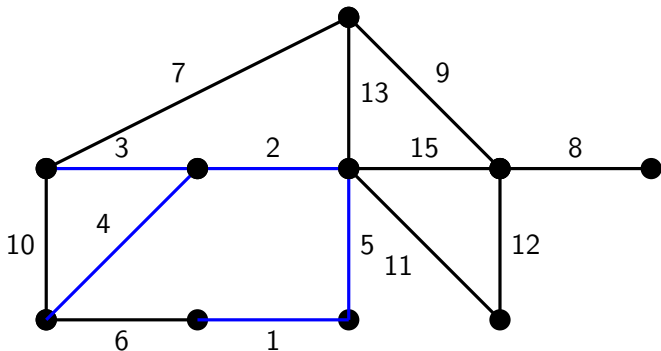
# Exemple



# Exemple

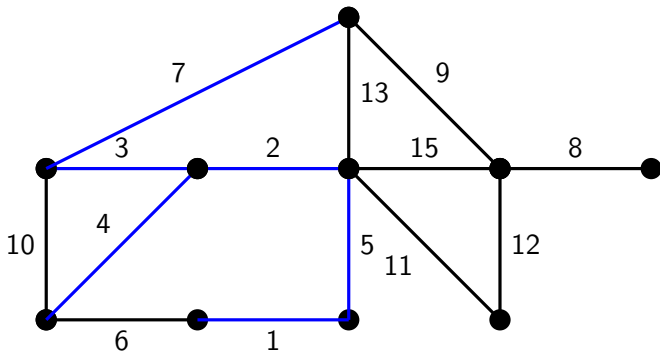


# Exemple

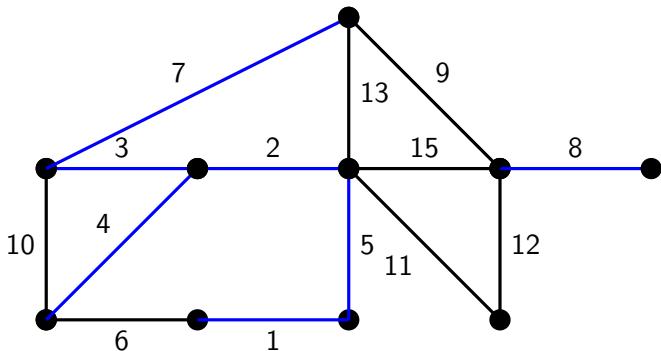




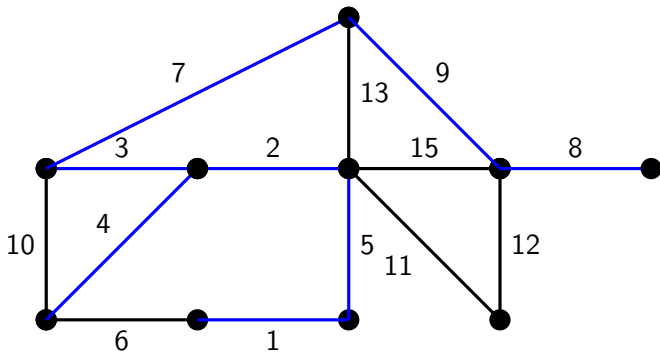
# Exemple



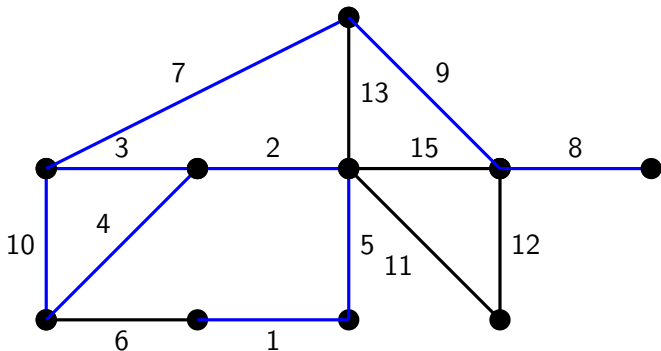
# Exemple



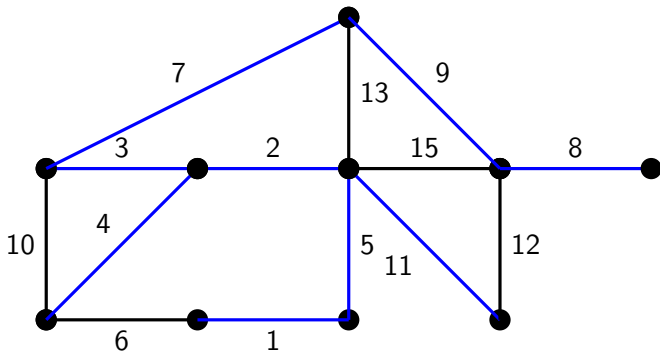
# Exemple



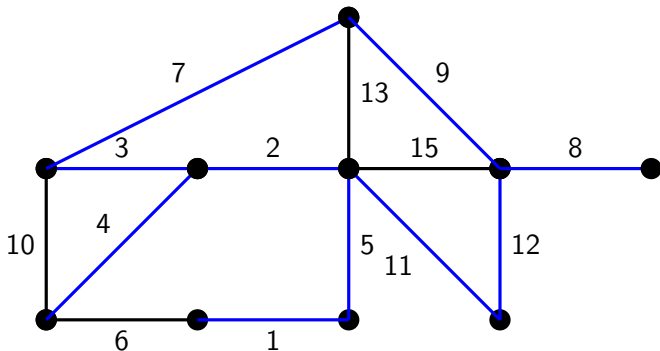
# Exemple



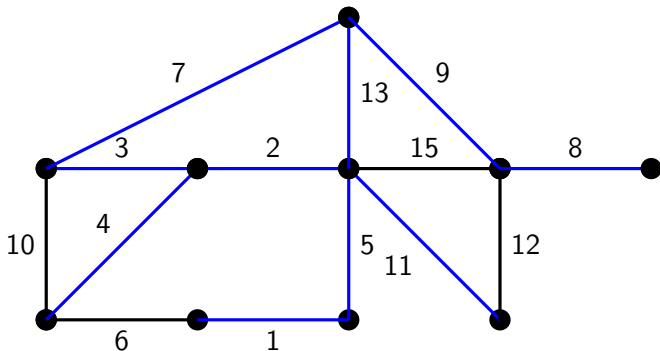
# Exemple



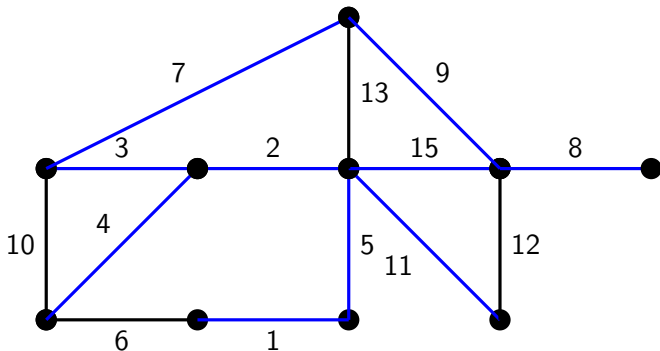
# Exemple



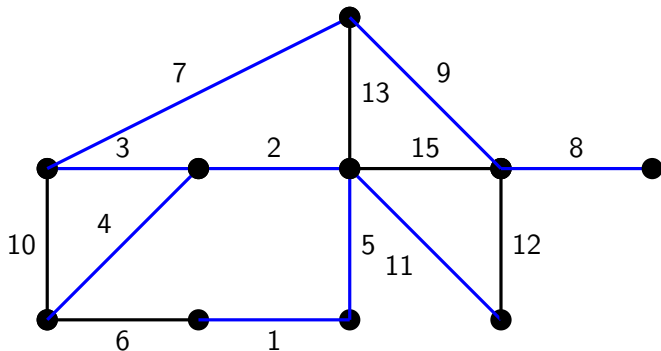
# Exemple



# Exemple



# Exemple



Résultat

# Implémentation

```
class Arête {  
    int extrémité1,extrémité2;  
    int coût;}
```

```
Arête[] arêtes= new Arête[nb_arêtes];  
...  
// lecture des arêtes: omis  
// tri des arêtes: omis  
// initialisation des tableaux père et taille union/find: omis  
...  
int coût=0;  
for (int i=0; i<arêtes.length;i++) {  
    if (find(arêtes[i].extrémité1)≠find(arêtes[i].extrémité2)) {  
        union(arêtes[i].extrémité1,arêtes[i].extrémité2);  
        coût+=arêtes[i].coût;}}
```

# L'algorithme de Kruskal est correct

- Il construit bien un arbre couvrant:
  - ▶ à chaque étape,  $T$  est une forêt.
  - ▶ chaque sommet a au moins une des arêtes qui lui est adjacente dans le  $T$  final.
- Il reste à montrer que l'arbre couvrant produit est de poids minimal.

## Lemme

- Lemme: Soient  $T$  et  $U$  deux arbres couvrants de  $G$ . Soit  $a$  une arête telle que  $a \in U$ ,  $a \notin T$ . Alors, il existe une arête  $b \in T$  telle que  $U - a + b$  soit un arbre couvrant. De plus,  $b$  peut être choisi dans le cycle formé par les arêtes de  $T$  et de  $a$ .
- Preuve:
  - ▶ la suppression de  $a$  dans  $U$  divise l'arbre en 2 composantes connexes  $U_1$  et  $U_2$ .
  - ▶ Le cycle  $\gamma$  créé par  $a$  dans  $T + a$  contient nécessairement un nombre pair d'arêtes reliant ces deux composantes.
  - ▶ Comme  $a$  est une telle arête, il en existe au moins une autre  $b$  dans  $T$ .
  - ▶ Cette arête  $b$  satisfait bien aux conditions du lemme.

## L'arbre couvrant produit est optimal

- Soit  $T$  l'arbre donné par l'algorithme de Kruskal.
- Si l'algorithme ne donne pas l'optimal c'est qu'il existe un arbre couvrant de poids inférieur. Choisissons en un  $U$  dont le nombre d'arêtes communes avec  $T$  est maximal.
- Soit  $a$  l'arête de  $U$  de plus petit poids qui n'est pas dans  $T$ , et  $b$  l'arête donnée par le lemme pour ce cas.
- Comme  $a$  n'a pas été choisie par l'algorithme de Kruskal, c'est qu'elle crée un cycle avec les éléments de  $T$ , et que ce cycle est formé d'arêtes toutes de poids inférieur ou égal à  $w(a)$ .
- Puisque  $b$  est dans ce cycle, on a  $w(b) \leq w(a)$ .
- $Poids(U - a + b) \leq Poids(U) < Poids(T)$ , mais  $U - a + b$  est un arbre couvrant qui a une arête commune avec  $T$  de plus que  $U$ .
- Ceci contredit le choix de  $U$ .

# Complexité de Kruskal

- Complexité de Kruskal:
  - ▶ Tri des arêtes:  $O(|E| \log |E|)$ .
  - ▶ Parcours des arêtes:  $O(|E|)$  finds et unions, soit  $O(|V| + |E|\alpha(|V|))$ .
- Total:  $O(|E| \log |E|) + O(|V| + |E|\alpha(|V|))$ .

# Complexité de Kruskal

- Complexité de Kruskal:
  - ▶ Tri des arêtes:  $O(|E| \log |E|)$ .
  - ▶ Parcours des arêtes:  $O(|E|)$  finds et unions, soit  $O(|V| + |E|\alpha(|V|))$ .
- Total:  $O(|E| \log |E|) + O(|V| + |E|\alpha(|V|))$ .
- Dans un graphe connexe,  $|V| - 1 \leq |E| \leq |V|^2$ , et donc  $\log |E| = O(\log |V|)$  et  $\log |V| = O(\log |E|)$ .

# Complexité de Kruskal

- Complexité de Kruskal:
  - ▶ Tri des arêtes:  $O(|E| \log |E|)$ .
  - ▶ Parcours des arêtes:  $O(|E|)$  finds et unions, soit  $O(|V| + |E|\alpha(|V|))$ .
- Total:  $O(|E| \log |E|) + O(|V| + |E|\alpha(|V|))$ .
- Dans un graphe connexe,  $|V| - 1 \leq |E| \leq |V|^2$ , et donc  $\log |E| = O(\log |V|)$  et  $\log |V| = O(\log |E|)$ .
- $\alpha(|V|) = O(\log |V|) = O(\log |E|)$ .

# Complexité de Kruskal

- Complexité de Kruskal:
  - ▶ Tri des arêtes:  $O(|E| \log |E|)$ .
  - ▶ Parcours des arêtes:  $O(|E|)$  finds et unions, soit  $O(|V| + |E|\alpha(|V|))$ .
- Total:  $O(|E| \log |E|) + O(|V| + |E|\alpha(|V|))$ .
- Dans un graphe connexe,  $|V| - 1 \leq |E| \leq |V|^2$ , et donc  $\log |E| = O(\log |V|)$  et  $\log |V| = O(\log |E|)$ .
- $\alpha(|V|) = O(\log |V|) = O(\log |E|)$ .
- Total plus joli:  $O(|E| \log |V|)$ .

# Complexité de Kruskal

- Complexité de Kruskal:
  - ▶ Tri des arêtes:  $O(|E| \log |E|)$ .
  - ▶ Parcours des arêtes:  $O(|E|)$  finds et unions, soit  $O(|V| + |E|\alpha(|V|))$ .
- Total:  $O(|E| \log |E|) + O(|V| + |E|\alpha(|V|))$ .
- Dans un graphe connexe,  $|V| - 1 \leq |E| \leq |V|^2$ , et donc  $\log |E| = O(\log |V|)$  et  $\log |V| = O(\log |E|)$ .
- $\alpha(|V|) = O(\log |V|) = O(\log |E|)$ .
- Total plus joli:  $O(|E| \log |V|)$ .
- Total plus joli:  $O(|E| \log |E|)$ .

# Aujourd'hui

Arbres de syntaxe abstraite

Union-Find

Arbres couvrant de poids minimal

Problème du voyageur de commerce

## Le problème du voyageur de commerce

Un représentant doit visiter  $n$  villes. Le représentant souhaite faire une tournée en visitant chaque ville au moins et exactement une fois, et en terminant à sa ville de départ.

## Sous forme de graphe

- On se donne un graphe connexe  $G = (V, E)$ , avec un poids  $w(e)$  pour chaque arête  $e = (u, v) \in E$ .
- On veut déterminer un *cycle hamiltonien* (un cycle simple qui passe par tous les sommets) de  $G$  de poids minimal.

## Sous forme de graphe

- On se donne un graphe connexe  $G = (V, E)$ , avec un poids  $w(e)$  pour chaque arête  $e = (u, v) \in E$ .
- On veut déterminer un *cycle hamiltonien* (un cycle simple qui passe par tous les sommets) de  $G$  de poids minimal.
- On ne connaît aucun algorithme pour résoudre ce problème qui fonctionne en temps polynomial (le problème est NP-complet).

## Complément: difficulté du problème

- En fait, on sait prouver qu'il existe un algorithme en temps polynomial pour un problème NP-complet si et seulement s'il existe un algorithme en temps polynomial pour n'importe lequel des problèmes NP-complets:
  - ▶ par exemple: il existe un algorithme en temps polynomial pour déterminer si un graphe général est 4-coloriable si et seulement s'il existe un algorithme en temps polynomial pour le problème du voyageur de commerce.
- Même si l'on se limite au cas Euclidien (les poids vérifient l'inégalité triangulaire), le problème du voyageur de commerce reste NP-complet.

## Le cas euclidien est 2-approximable

- On se place dans le cas suivant:
  - ▶ le graphe est complet: pour toute paire de sommets  $u, v$ , il y a une arête  $(u, v)$ .
  - ▶ les poids vérifient l'inégalité triangulaire:

pour 3 sommets  $u, v, w$  arbitraires de  $V$ ,

$$w(u, v) \leq w(u, w) + w(w, v).$$

- On ne connaît pas d'algorithme en temps polynomial qui produit une solution optimale.
- Mais on connaît un algorithme en temps polynomial qui produit une solution dont le poids est inférieur au double de l'optimal.

$$\text{Poids}(\text{Solution de l'algorithme}) \leq 2 * \text{Poids}(\text{Cycle optimal}).$$

## Le cas euclidien est 2-approximable

- On se place dans le cas suivant:
  - ▶ le graphe est complet: pour toute paire de sommets  $u, v$ , il y a une arête  $(u, v)$ .

(Quitte à ajouter des arêtes de poids infini (très grand), on peut toujours s'y ramener).

- ▶ les poids vérifient l'inégalité triangulaire:

pour 3 sommets  $u, v, w$  arbitraires de  $V$ ,

$$w(u, v) \leq w(u, w) + w(w, v).$$

- On ne connaît pas d'algorithme en temps polynomial qui produit une solution optimale.
- Mais on connaît un algorithme en temps polynomial qui produit une solution dont le poids est inférieur au double de l'optimal.

$$\text{Poids}(\text{Solution de l'algorithme}) \leq 2 * \text{Poids}(\text{Cycle optimal}).$$

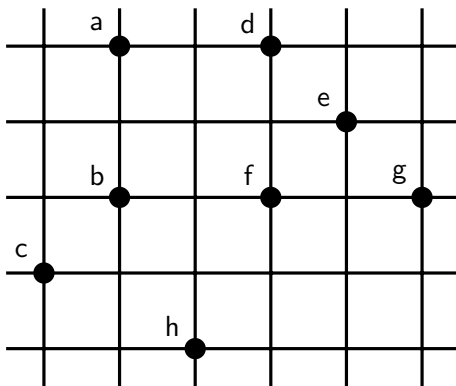
# L'algorithme

Algorithme:

- Construire un arbre couvrant de poids minimal  $T$  de  $G$ .
- Soit  $L$  la liste des sommets de  $G$  dans l'ordre où ils sont visités dans un parcours préfixe de  $T$ .
- Renvoyer le cycle hamiltonien qui visite les sommets dans l'ordre de  $L$ .

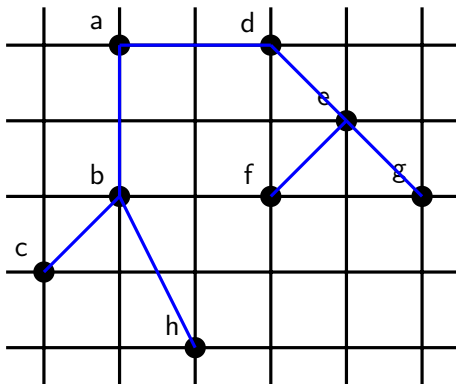
# Exemple

Graphe initial (les poids sont les distances euclidiennes).



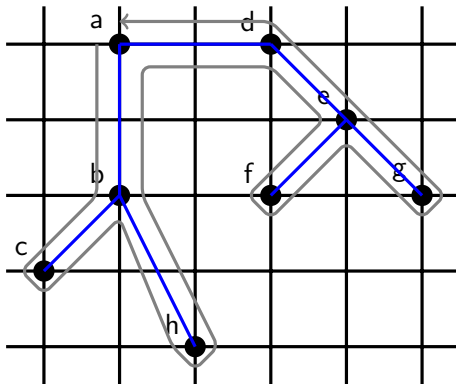
# Exemple

Arbre couvrant  $T$  de poids minimal.

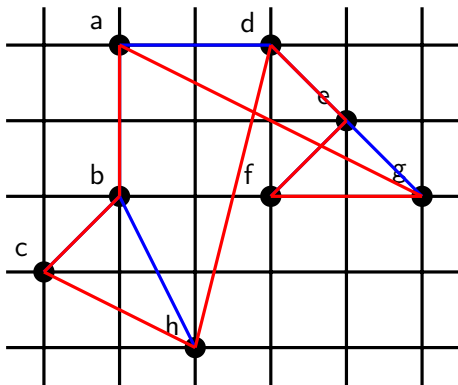


# Exemple

Parcours préfixe de  $T$ :  $abchdefg$ .

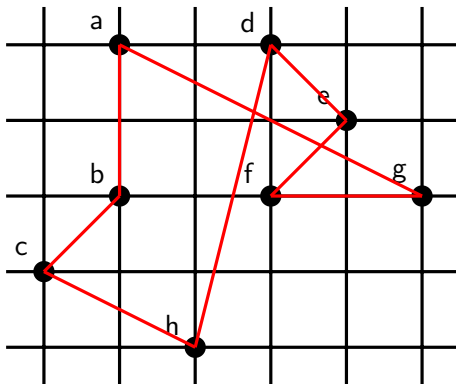


# Exemple



# Exemple

Résultat

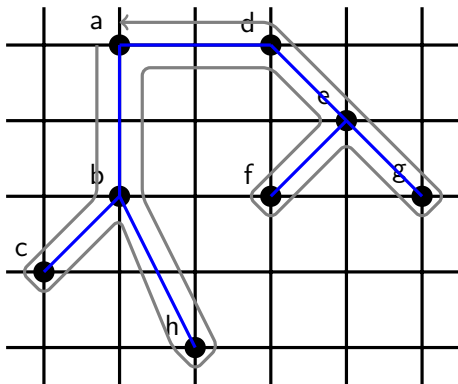


## Preuve du facteur 2 d'approximation

- Soit  $H^*$  une tournée optimale, et  $H$  la tournée retournée par l'algorithme.
- Il reste donc à prouver que  $poids(H) \leq 2 * poids(H^*)$ .
- Soit  $T$  un arbre couvrant minimal:  $poids(T) \leq poids(H^*)$ , car le cycle  $H^*$  privé d'une arête est un arbre.
- Soit  $L$  le résultat d'un parcours de l'arbre  $T$ , en suivant l'ordre préfixe et avec les répétitions.

## Preuve du facteur 2 d'approximation (suite)

Sur l'exemple: parcours  $L = a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$ .



$$\text{poids}(L) = 2 * \text{poids}(T),$$

car chaque arête de  $T$  est visitée 2 fois.

## Preuve du facteur 2 d'approximation (suite)

- Donc  $\text{poids}(L) \leq 2 * \text{poids}(H^*)$ .
- On peut supprimer les sommets en double de  $L$  un par un sans augmenter le coût, grâce à l'inégalité triangulaire.
  - ▶ exemple: plutôt que de passer une deuxième fois par  $b$  dans  $L = a, b, c, b, h, b, a, d, e, f, e, g, e, d, a$ , on peut faire  $L = a, b, c, h, b, a, d, e, f, e, g, e, d, a$ : le poids d'aller de  $c$  à  $h$  est inférieur ou égal au poids d'aller de  $c$  à  $b$  puis de  $b$  à  $h$  par l'inégalité triangulaire.
- Donc  $\text{poids}(H) \leq \text{poids}(L) \leq 2 * \text{poids}(H^*)$ .