



B. WERNER

F. MORAIN



## Graphes II : parcours

10 mars 2010

- I. Motivations.
- II. Définitions et premiers algorithmes.
- III. Parcours en largeur d'abord.
- IV. Parcours en profondeur d'abord.
- V. Arbre couvrant de poids minimal.

### I. Motivations

Comment détecter la connexité d'un graphe non orienté, énumérer ses composantes connexes ?

Classer les sommets d'un graphe en fonction de leur distance (nombre minimal de voisins intermédiaires) à un sommet donné ?

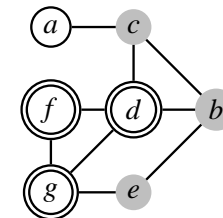
Généralisation aux graphes orientés : recherche de circuits, tri topologique.

Tous les graphes considérés sont **simples**.

On commence avec des graphes non orientés.

### II. Définitions et premiers algorithmes

**L'idée** : à partir d'un sommet  $s$ , on explore une partie des sommets du graphe, les voisins de  $s$ . De proche en proche, nous sommes conduits à choisir les sommets suivants dans la **bordure** du graphe.



**Déf.** La **bordure**  $\mathcal{B}(T)$  de  $T \subset \mathcal{S}$  est l'ensemble des sommets de  $\mathcal{S} - T$  adjacents à (un des sommets de)  $T$ .

# Parcours

**Déf.** Un **parcours** de  $\mathcal{G} = (\mathcal{S}, \mathcal{A})$  connexe à partir du sommet  $s$  est une liste de sommets  $L$  telle que :

- le premier sommet de  $L$  est  $s$  ;
- chaque sommet de  $\mathcal{S}$  apparait une fois et une seule dans  $L$  ;
- tout sommet de la liste (sauf le premier) est adjacent dans  $\mathcal{G}$  à au moins un sommet placé avant lui dans la liste.

**Rem.** Il n'y a pas de parcours canonique.

**Déf.** Le **support**  $\sigma(L)$  de  $L$  est l'ensemble des sommets contenus dans  $L$ . Par abus de notation, on notera souvent  $\mathcal{B}(L) = \mathcal{B}(\sigma(L))$ .

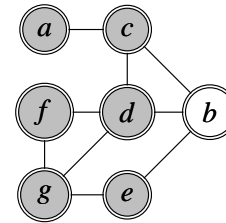
# L'algorithme générique de parcours

```

fonction parcoursGenerique(G, s)
1. L <- (s);
   B <- V(s); // les voisins de s
2. tantque B ≠ ∅ faire
   2.1 choisir u dans B; // exploration
   2.2 L <- L # u;
   2.3 B <- B - {u};
   2.4 B <- B ∪ (V(u) \ L);

```

**Ex.**  $L = (b, d, e, g, f, c, a)$  :



$L$	$B$
$(b)$	$\{c, d, e\}$
$(b, d)$	$\{c, e, f, g\}$
$(b, d, e)$	$\{c, f, g\}$
$(b, d, e, g)$	$\{c, f\}$
$(b, d, e, g, f)$	$\{c\}$
$(b, d, e, g, f, c)$	$\{a\}$
$(b, d, e, g, f, c, a)$	$\emptyset$

# Propriétés fondamentales

**Prop.** Avant 2.1,  $L \cap B = \emptyset$ .

**Thm.** Si les opérations ensemblistes coûtent  $O(1)$ , alors la complexité du parcours est  $O(|\mathcal{S}| + |\mathcal{A}|) = O(n + m)$ .

**Dém.** Chaque sommet de  $\mathcal{G}$  est choisi une fois et une seule dans un parcours.

Le coût total est

$$\leq \sum_{u \in \mathcal{S}} \left( O(1) + \sum_{v \text{ voisin de } u} O(1) \right) = O(n) + O\left(\sum_{u \in \mathcal{S}} n_u\right) = O(n + m). \square$$

# Application à la connexité

```

// affiche toutes les composantes connexes
fonction composantesConnexes(G = (S, A))
T <- copie(S);
tantque T ≠ ∅
  choisir s dans T;
  T <- T - {s};
  L <- uneComposante(G, s, T);
  afficher "Composante : ", L;

// retourne la composante connexe contenant s
fonction uneComposante(G, s, T)
1. L <- (s); B <- V(s);
2. tantque B ≠ ∅ faire
   2.1 choisir u dans B;
   2.2 L <- L # u;
   2.3 B <- B - {u};
   2.4 B <- B ∪ (V(u) \ L);
   2.5 T <- T - {u};
3. retourner L;

```

# Numérotation

Tout parcours induit une (re)numérotation des sommets de  $\mathcal{G}$ , par ordre d'apparition dans le parcours.

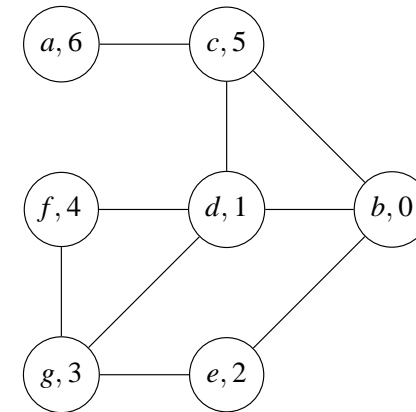
fonction `parcoursNumérotation(G, s)`

```

1. num ← 0;
2. 2.1 L ← (s); B ← V(s);
   2.2 numéro[s] ← num++;
3. tantque B ≠ ∅ faire
   3.1 choisir u dans B;
   3.2 L ← L # u;
   3.3 B ← B - {u};
   3.4 B ← B ∪ (V(u) \ L);
   3.5 numéro[u] ← num++;

```

$L = (b, d, e, g, f, c, a) :$

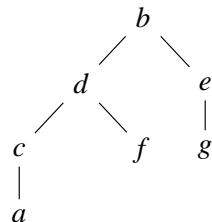
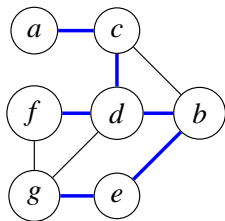


## Arbre couvrant

**Prop.** Soit  $L = (x_0, \dots, x_{n-1})$  un parcours de  $\mathcal{G}$ . Pour tout  $k$ ,  $1 \leq k < n$ , soit  $x'_k$  un sommet de  $(x_0, \dots, x_{k-1})$  adjacent à  $x_k$ . Le sous-graphe induit par les arêtes  $(x_k, x'_k)$ ,  $1 \leq k < n$ , est un arbre, appelé **arbre couvrant** de  $G$  (relatif à  $L$ ).

**Déf.** Les arêtes  $(x_k, x'_k)$  sont appelées **arêtes de liaison**.

**Ex.**  $L = (b, d, e, g, f, c, a) :$



**Démonstration.** On note  $G_p$  le graphe induit par  $k \in \{0, \dots, p\}$  et on raisonne par récurrence sur  $p$ .

$p = 1$  : seule arête  $(x_0, x_1)$ , donc  $x'_1 = x_0$ .

$p > 1$  :  $G_{p-1}$  connexe  $\Rightarrow G_p$  connexe car on a rajouté une arête avec un sommet dans  $G_{p-1}$ ;  $G_p$  a  $p - 1$  arêtes et  $p$  sommets, n'a pas de cycle, donc est un arbre.  $\square$

# Implantation réaliste

**Problème** : quelle structure choisir pour  $\mathcal{B}$  pour choisir et réaliser l'insertion/suppression et test d'appartenance en temps minimal ?

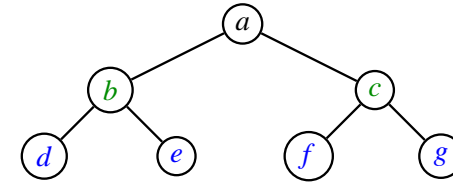
## Rappels :

- Un ensemble représenté par table de hachage peut réaliser l'insertion/suppression et l'appartenance en temps  $O(1)$ . Mais comment choisir ?
- Une liste de sommets à visiter permet facilement l'insertion/suppression et le choix (e.g. le premier élément). Mais comment tester l'appartenance ?
- Une solution : dupliquer en utilisant à la fois liste et table de hachage (mais on perd en temps et mémoire même si en  $O(1)$ ).
- Autre solution : utiliser une liste et gérer un **état** pour un sommet qui indique s'il est déjà dans la liste ou pas.

# III. Parcours en largeur d'abord

**Idée** : (*Breadth-first search*) les voisins de nos voisins sont nos voisins. On procède ainsi par "cercles" concentriques.

## Exemple avec un arbre :



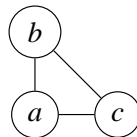
**Règle de choix** : le prochain sommet visité est le plus ancien ajouté à la bordure.

⇒ on utilise une file d'attente pour gérer la bordure.

```
fonction bfs(G, s)
```

```
1. F <- (s);  
2. L <- (s);  
3. tantque F n'est pas vide faire  
   t <- tête(F);  
   L <- L # t;  
   pour u voisin de t faire  
     F <- F # u;
```

... mais va boucler sur



## Première solution

```
fonction bfs(G, s)
```

```
1. F <- (s);  
2. L <- (s);  
3. tantque F n'est pas vide faire  
   t <- tête(F);  
   L <- L # t;  
   pour u voisin de t faire  
     si u n'a pas déjà été vu alors  
       F <- F # u;
```

**Programmation** : on utilise un tableau pour gérer l'état d'un sommet.

## Le pseudocode

```

fonction bfs(G, s)
0. pourtout sommet t faire
    etat[t] <- inexploré;
1. F <- (s); etat[s] <- encours;
2. tantque F ≠ ∅
    t <- tête(F);
    pour u voisin de t faire
        si etat[u] == inexploré alors
            etat[u] <- encours;
            F <- F # u;
    etat[t] <- exploré;

```

**Prop.** La complexité de bfs est  $O(|S| + |A|)$ .

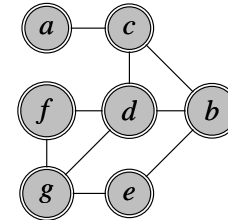
*Dém.* Le théorème générique s'applique, puisque les opérations de file et de gestion des états coûtent  $O(1)$ . □

## Exemple

```

b c d e
c a d b
a c
d c f g b
e b g
f d g
g d f e

```



la file F

$F = \emptyset$   $F = (b)$   $F = (c)$   $F = (c, d)$   $F = (c, d, e)$   $F = (d, e, a)$   
 $F = (e, a, f, g)$   $F = (a, f, g)$   $F = (f, g)$   $F = (g)$   $F = \emptyset$

## Le code Java dans Graphe

```

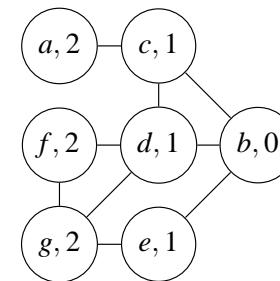
final static
int inexplore = 0, explore = 1, encours = 2;
public void bfs(Sommet s){
    Hashtable<Sommet,Integer> etat
    = new Hashtable<Sommet,Integer>();
    LinkedList<Sommet> f = new LinkedList<Sommet>();

    for(Sommet t : sommets())
        etat.put(t, inexplore);
    etat.put(s, encours);
    f.addLast(s);
    while(! f.isEmpty()){
        Sommet t = f.removeFirst();
        for(Arc<Sommet> a : voisins(t)){
            Sommet u = a.destination();
            if(etat.get(u) == inexplore){
                etat.put(u, encours);
                f.addLast(u);
            }
        }
        etat.put(t, explore);
    }
}

```

## Propriétés du parcours BFS

Le parcours crée des couches successives  $C_0 = \{b\}, C_1, \dots$  :



**Prop.** Pour un parcours BFS(s) :

- (1)  $C_i$  contient les sommets à distance  $i$  de  $s$ ;
- (2) Soit  $T$  l'arbre associé au parcours. Si  $(x, y) \in T \cap A$ , avec  $x \in C_i$ ,  $y \in C_j$ , on a  $|i - j| \leq 1$ .

## IV. Parcours en profondeur d'abord

**BFS** : le prochain sommet exploré est le plus ancien sommet de type **encours**.

**DFS** : (*Depth First Search*) le prochain sommet visité est le plus récent sommet de type **encours**.

```
fonction dfs(G, s)
0. pourtout sommet t faire
    etat[t] <- inexploré;
1. dfsRec(G, s, etat);
```

```
fonction dfsRec(G, s, etat)
si etat[s] == inexploré alors
    etat[s] <- encours;
    pourtout t voisin de s
        dfsRec(G, t, etat);
    etat[s] <- exploré;
```

## Le code Java

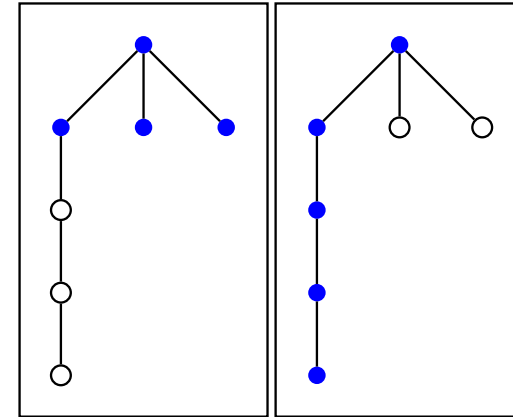
```
void dfsRec(Hashtable<Sommet,Integer> etat,
            Sommet s){
    if(etat.get(s) == inexplorer){
        etat.put(s, encours);
        for(Arc<Sommet> a : voisins(s))
            dfsRec(etat, a.destination());
        etat.put(s, explore);
    }
}
```

**Rem.** On peut se passer de **encours**.

**Ex.** Programmer en utilisant une pile.

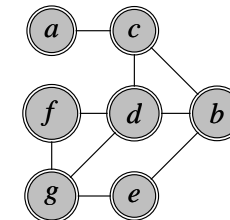
## La différence géographique

Largeur = au plus près ; profondeur = au plus loin.



**Exercice.** À quoi correspond une dfs sur un arbre ?

## Exemple



# Construction de l'arbre couvrant

**Déf.** Une **arborescence** est un arbre dont on a distingué un sommet, la racine.

**Déf.** Nous appellerons **arborescence de Trémaux** un arbre couvrant obtenu lors d'une dfs.

On se donne un type d'arbre  $n$ -aire :

```
class Arbre{
    Sommet racine;
    LinkedList<Arbre> fils;

    Arbre(Sommet r){
        racine = r;
        fils = new LinkedList<Arbre>();
    }
    void ajouterFils(Arbre A){
        fils.addLast(A);
    }
}
```

```
void dfsCouvrant(){
    Sommet ALPHA = new Sommet("ALPHA");
    Arbre F = new Arbre(ALPHA);
    Hashtable<Sommet,Integer> etat =
        new Hashtable<Sommet,Integer>();

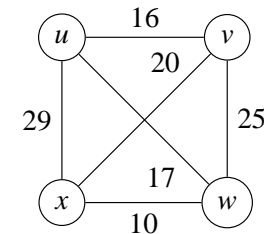
    for(Sommet s : sommets())
        etat.put(s, inexplorer);
    for(Sommet s : sommets())
        if(etat.get(s) == inexplorer)
            F.ajouterFils(dfsRec(s));
}
```

```
// on retourne un arbre couvrant de racine s
Arbre dfsRec(Hashtable<Sommet,Integer> etat,
             Sommet s){
    etat.put(s, encours);
    Arbre A = new Arbre(s);
    System.out.println("J'explore "+s);
    for(Arc<Sommet> a : voisins(s)){
        Sommet t = a.destination();
        if(etat.get(t) == inexplorer)
            A.ajouterFils(dfsRec(etat, t));
    }
    etat.put(s, explore);
    return A;
}
```

## V. Arbre couvrant de poids minimal

**Pb typique** : étant donné un réseau de canaux de capacité connue devant irriguer un nombre donné de villes, minimiser la somme totale des capacités.

**Modélisation** : graphe (non orienté) dont les sommets sont les villes et les arêtes les canaux ;  $val(a)$  = capacité de l'arc  $a$ .

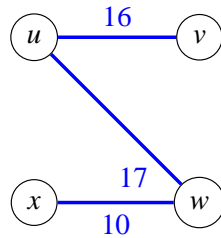


# Principe

On note  $\varpi(\mathcal{G})$  le poids d'un graphe, défini comme la somme des valeurs de ses arcs.

Nous cherchons un sous-graphe couvrant connexe de  $\mathcal{G}$ , qui passe par tous les sommets de  $\mathcal{G}$ , et de poids minimum.

**Rem.** Si ce sous-graphe possédait un cycle, il suffirait de l'enlever pour diminuer son poids.  $\Rightarrow$  on cherche en fait un **arbre couvrant de poids minimum**.



# Implantation

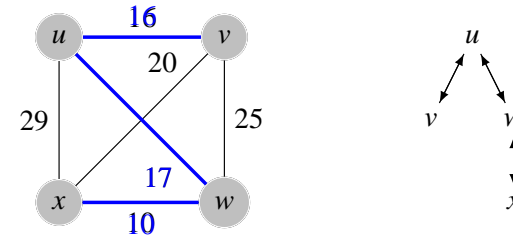
Avec une file de priorité d'arcs classés par ordre croissant de valuation  $\Rightarrow O(m \log m)$ .

```
Prim(G = (S, A))
X <- S;
tantque X  $\neq$   $\emptyset$  faire
  u <- élément suivant de X;
  X <- X - {u};
  T[u] <-  $\emptyset$ ; // arbre couvrant de racine u
  F <- file de priorité avec les arêtes (u, x);
  Y <- {u}; // ensemble des sommets de T[u]
  tantque F  $\neq$   $\emptyset$  faire
    a = (s, t) <- arête minimale dans F;
    si t  $\in$  Y alors
      // (s, t) ferme un cycle
    sinon
      Y <- Y  $\cup$  {t}; X <- X - {t};
      T[u] <- T[u]  $\cup$  {(s, t)};
      pour v voisin de t
        ajouter (t, v) dans F;
```

# A) L'algorithme de Prim

1. Choisir un sommet initial  $s$ ;
2. **tantque** c'est possible
3. Trouver l'arête de poids minimal joignant un sommet déjà sélectionné à un sommet non encore sélectionné.

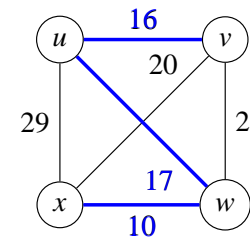
Ex.



# B) L'algorithme de Kruskal

1. Trier les arêtes par ordre croissant:  
B = (a<sub>1</sub>, ..., a<sub>m</sub>);
2. T <-  $\emptyset$ ;
3. **pour** i <- 1 à m **faire**  
    **si** a<sub>i</sub> ne crée pas de cycle  
    ajouter a<sub>i</sub> à T;

Ex.



**Principe** : union-find pour gérer les cycles.

$C[s] = \{t \in \mathcal{S}, \text{ il existe un chemin entre } t \text{ et } s \text{ dans la forêt en construction}\}$ .

```
insérer(C, a)
a = (s, t)
si C[s] == NIL et C[t] == NIL alors
  // l'arête (s, t) est nouvelle
  C[s] <- {s, t};
  C[t] <- C[s]; // partage
sinon si C[s] == NIL alors
  // t est déjà connu
  C[t] <- C[t] ∪ {s};
  C[s] <- C[t]; // partage
sinon si C[t] == NIL alors
  // s est déjà connu
  C[s] <- C[s] ∪ {t};
  C[t] <- C[s];
```

```
sinon // s et t sont déjà connus
  si C[s] == C[t] alors
    // s et t sont dans la même composante,
    // (s, t) forme un cycle, on ne fait rien
  sinon
    fusionner C[s] et C[t].
```

**Prop.** Le coût de Kruskal est  $O(m(\log m + \alpha(m)))$  avec  $\alpha(m)$  l'inverse de la fonction d'Ackermann  $Ack(m, m)$ .

## C) Correction des algorithmes de Prim et Kruskal

**Ex.** On commence par  $wx$  :

$$C[w] = C[x] = \{w, x\}.$$

La deuxième arête est  $uv$ , ce qui crée

$$C[u] = C[v] = \{u, v\}.$$

Quand on veut insérer  $uw$ , on assiste à la fusion des composantes :

$$C[w] = C[x] = C[u] = C[v] = \{w, x, u, v\}.$$

Les trois dernières arêtes ne sont pas insérées, puisque les sommets sont déjà dans  $C$ .

On construit  $T$  avec la liste  $(a_1, a_2, \dots, a_{n-1})$  telle que

$$val(a_1) \leq val(a_2) \leq \dots \leq val(a_{n-1}).$$

Supposons qu'il existe un arbre couvrant  $T'$  tq  $\varpi(T') < \varpi(T)$ .

Soit  $a$  une arête de poids minimum présente dans  $T$  mais pas dans  $T'$  :

$$(a_1, a_2, \dots, a_{r-1}, a_r = a, a_{r+1}, \dots, a_{n-1})$$

avec  $a_i \in T'$  pour  $i < r$ .

$T' + a$  contient un cycle (tous les sommets de  $\mathcal{S}$  sont dans  $T'$ ).

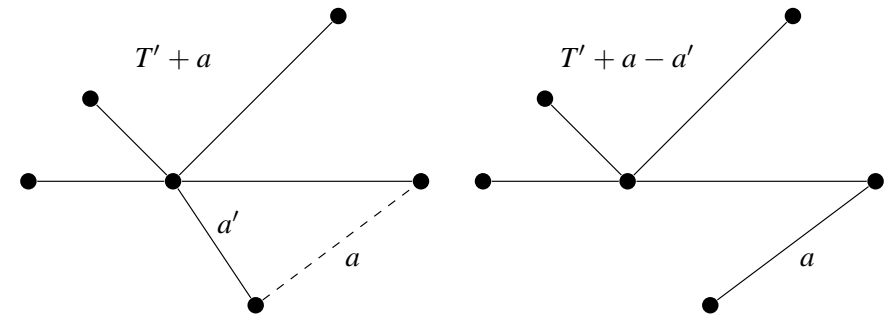
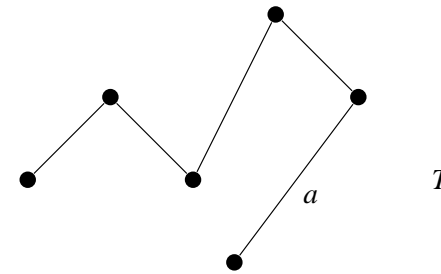
Ce cycle contient une autre arête  $a'$  non présente dans  $T$  : si toutes les arêtes étaient dans  $T$ ,  $a$  n'aurait pu être choisi par l'algorithme (car aurait créé ce cycle).

$T'' = T' + a - a'$  est encore un arbre couvrant.

Si  $val(a') < val(a)$  : comme  $a' \notin T$ , il n'était pas éligible par l'algorithme, donc il y aurait un cycle dans  $\{a_1, a_2, \dots, a', \dots, a_{r-1}\}$  et donc un cycle dans  $T'$ .

D'où  $\varpi(T'') = \varpi(T') + val(a) - val(a') \leq \varpi(T') < \varpi(T)$ .

De proche en proche, on peut donc passer de  $T$  à  $T'$ , alors que le poids de  $T$  est plus grand que celui de  $T'$ , contradiction.



## Résumé du cours

- Parcours : outil fondamental dans l'algorithmique des graphes ; sortie de labyrinthe, etc.
- Cas non orienté.
- Arbre Couvrant de poids minimal

**Prochains rendez-vous :** [PC Graphes cet après-midi.](#)

**Semaine prochaine :** parcours pour graphes orientés