

## recherche séquentielle :

```

trouvé ← faux
i ← 0
tant que (non trouvé) et ( $i < n$ )
    si  $T[i] = x$ , trouvé ← vrai
    sinon  $i \leftarrow i + 1$ 
si trouvé = vrai écrire oui
sinon écrire non
Complexité :  $O(n)$ 

```

## recherche dichotomique

Si la liste est triée, complexité en  $O(\log(n))$

## tableau :

C'est une collection de variables de même type.

Accès à chaque case en  $O(1)$

## liste chaînée :

Chaque maillon contient l'adresse du maillon suivant.

Accès au  $i$ -ème maillon en  $O(i)$

## insertion d'un élément en tête : en $O(1)$

créer un maillon d'adresse ad

ad.donnée ← nouvelle donnée

ad → donnée

ad.suivant → début

ad → suivant

début ← ad

## pile et file :

		tableau	Liste chaînée
pile	LIFO	Ajout/suppression en $O(1)$	Ajout en tête en $O(1)$
file	FIFO	Ajout/suppression en $O(1)$ variables début (incrémentée quand on supprime) et fin (incrémentée quand on ajoute)	Ajout/suppression en $O(1)$ Il faut un pointeur début et un pointeur fin

## Arbre :

Structure composée de sommets regroupés de la façon suivante :

- une racine
- des arbres disjoints (sous arbres). Il peut y en avoir autant que l'on veut y compris 0.

### hauteur (n noeuds) :

$$\begin{cases} 0 \text{ si } n = 1 \\ 1 \text{ si } n > 1 \end{cases} \leq h \leq n - 1$$

## Parcours d'un arbre ( $R, A_1, \dots, A_k$ ):

### préfixe :

examiner R

pour i variant de 1 à k

parcourir  $A_i$  en préfixe

### postfixe :

pour i variant de 1 à k

parcourir  $A_i$  en postfixe

examiner R

## Arbre binaire :

Soit vide soit de la forme  $(R, A_g, A_d)$  avec  $R$  un nœud (racine) et  $A_g$  et  $A_d$  deux arbres binaires.

### parcours infixé :

parcourir  $A_g$  en infixé

examiner R

parcourir  $A_d$  en infixé

### hauteur d'un AB (n noeuds) :

$$\log_2(n) \leq h \leq n - 1$$

## Tris :

Thm : Un tri comparatif a une complexité dans le pire des cas d'un moins  $O(n \log(n))$

## sélection :

principe : on cherche le plus petit élément et on le met au début

### pseudo code :

```

❖ pour i allant de 1 à  $n - 1$ 
    > indicePetit ← i
    > min ←  $T[i]$ 
    > pour j allant de  $i + 1$  à  $n$ 
        ■ si  $T[j] < min$ 
            • indicePetit ← j
            • min ←  $T[j]$ 
    > échanger( $T, i, indicePetit$ )

```

Complexité :  $O(n^2)$

### insertion :

principe : on suppose les  $i$  premiers éléments triés, on les compare au  $(i+1)^{\text{e}}$  en commençant par le  $i^{\text{e}}$  jusqu'à trouver sa place.

### pseudo code :

```

❖ pour i allant de 2 à  $n$ 
    >  $j \leftarrow i$ 
    > clé ←  $T[j]$ 
    > tant que  $j \geq 2$  et  $T[j - 1] > clé$ 
        ■  $T[j] \leftarrow T[j - 1]$ 
        ■  $j \leftarrow j - 1$ 
    >  $T[j] \leftarrow clé$ 

```

Complexité :  $O(n^2)$

### Tri rapide :

principe : on utilise une fonction *partition* qui s'intéresse aux données entre  $g$  et  $d$  et forme la liste :

$\leq T[g]$	$T[g]$	$> T[g]$
-------------	--------	----------

### pseudo code (tri Rapide(g,d))

```

❖ Si  $g < d$ 
    >  $j \leftarrow partition(g, d)$ 
    >  $triRapide(g, j - 1)$ 
    >  $triRapide(j + 1, d)$ 
fonction partition(g,d) :
❖  $clé \leftarrow T[g]$ 
❖  $i \leftarrow g + 1$ 
❖  $j \leftarrow d$ 
❖ tant que  $i \leq j$ 
    > tant que  $i \leq j$  et  $T[i] \leq clé$ ,  $i \leftarrow i + 1$ 
    > tant que  $T[j] > clé$ ,  $j \leftarrow j - 1$ 
    > si  $i < j$ 
        ■ échanger( $T, i, j$ )
        ■  $i \leftarrow i + 1$ 
        ■  $j \leftarrow j - 1$ 
    > échanger( $T, g, j$ )
    & renvoyer  $j$ 

```

Complexité :  $O(n^2)$  dans le pire des cas  
 $O(\log(n))$  dans le meilleur des cas

### Tri par Arbre Binaire de Recherche :

Un ABR est un AB dont les nœuds sont pourvus de clés telles que :

la clé de tout nœud est comprise entre celles de son arbre gauche et celles de son sous arbre droit.

intérêt : un parcours infixé trie les données en  $O(n)$

### insertion d'un élément :

on compare l'élément à la racine et on sait alors dans quel sous arbre le mettre et on recommence pour chaque sous arbre. Au pire l'insertion est en  $O(n)$ , au mieux en  $O(\log(n))$

Complexité : construction  $O(n^2)$  et parcours  $O(n)$  donc en tout  $O(n^2)$

Complexité : construction  $O(n^2)$  et parcours  $O(n)$  donc en tout  $O(n^2)$

### Tris Tas :

Arbre binaire parfait : un ABP et un AP dont les niveaux sont saturés sauf éventuellement le dernier et sur le dernier niveau les nœuds sont le plus à gauche possible.

Tas : ABP tel que pour tout nœud N  $clé(N) \geq clés de ses fils$

## Représentation par un tableau :

N nœud d'indice i alors

- les fils de N ont les indices  $2i$  et  $2i + 1$

- le père de N a l'indice  $E\left(\frac{i}{2}\right)$

### Construction d'un tas :

insertion d'une donnée dans un tas à p nœuds : placer la donnée à la première place libre. Puis tant que la nouvelle clé est supérieure à celle de son père on échange ceux-ci.

insertion en  $O(\log(p))$

construction en  $O(n \log(n))$

### principe du tri tas :

On répète n fois le procédé suivant :

- ranger la clé de la racine du tas courant à droite du tableau et placer la clé de ce qui était à droite à la racine du tas

- restaurer la structure de tas en échangeant la clé qui vient de monter avec le plus grand de ses fils jusqu'à que cette clé soit plus grande que ses deux fils

Complexité :  $O(n \log(n))$

## Hachage :

check  $\{x_i\} \in L$ ,  $\{f_j\} \in \Sigma$ ,  $L \in \Sigma$ ,  $f_j(x_i) = 1$

On veut vérifier l'appartenance de p mots  $(x_1, \dots, x_p)$  à une liste L de n mots dans un alphabet  $\Sigma$

### principe :

- tableau T de  $m \geq n$  mots

- fonction de hachage :

$h(\text{mots écrits dans } \Sigma) \rightarrow \{0, 1, \dots, m - 1\}$

- pour  $\mu \in L$  :  $T[h(\mu)] \leftarrow \mu$

collision :  $\exists \mu \neq \mu'$  dans  $L$  tq  $h(\mu) = h(\mu')$

- S'il n'y a pas collision on hache tous les mots de L pour les mettre dans T

- Pour i allant de 1 à p, pour savoir si  $x_i \in L$  on compare  $h(x_i)$  et  $x_i$ , on a :

$x_i \in L \Leftrightarrow T[h(x_i)] = x_i$

Complexité :  $O(n + p)$

### gestion des collisions :

On crée un tableau de listes chainées dont les maillons ont la même image par h.

Pour savoir si  $x \in L$  : recherche séquentielle dans la liste chainée correspondant à  $h(x) =$

### Complexité :

Dans le pire des cas (tout a été envoyé en i) :  
 $O(n + np) = O(np)$

Si on s'y prend bien (listes chainées de longueur majorée) :  $O(n + p)$

## Algorithme de Huffman

### longueur du message M :

$$L(M) = (\text{nb de caractères}) \times L(\text{caractère})$$

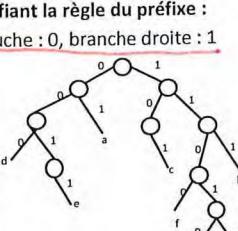
### Règle du préfixe :

le codage d'un caractère n'est jamais de début du codage d'un autre caractère

### construction d'un arbre binaire à partir d'un codage vérifiant la règle du préfixe :

branche gauche : 0, branche droite : 1

a	01
b	111
c	101
d	000
e	0011
f	1100
g	11010
h	11011



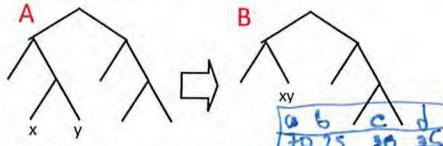
$$L(M) = \sum_{x \in \{ \text{caractères} \text{ du message } M \}} \text{occ}(x) \cdot \text{prof}(x)$$

**prop :** dans un arbre optimal toutes les feuilles ont des sœurs et tout noeud interne a deux fils.

**prop:** Il existe un arbre optimal dans lequel deux feuilles sœurs de profondeur max sont associées aux deux caractères les moins fréquents.

### Algorithme de Huffman :

on considère un arbre optimal A vérifiant les propriétés ci-dessus. Puis on transforme cet arbre en un arbre B comme cela :



avec  $\text{occ}(xy) = \text{occ}(x) + \text{occ}(y)$

**Prop :** A est optimal sur  $\Sigma$ ssi B est optimal sur  $\Sigma - \{x, y\}$

#### Principe :

- tableau d'arbres binaires : au début chaque case contient un arbre formé uniquement d'un caractère.

- trier les caractères par occurrence décroissante.

- Pour i allant de 1 à n-1

- On fusionne les 2 arbres les plus à droite du tableau courant en un arbre binaire

- On considère un nouveau caractère xy obtenu par concaténation des caractères associés aux racines des arbres fusionnés (x et y) avec  $\text{occ}(xy) = \text{occ}(x) + \text{occ}(y)$  et on insère le nouvel arbre pour respecter l'ordre des occurrences



### Graphes

**Graph fini simple orienté :**  $G = (X, A)$  avec X l'ensemble des sommets (finis) et A l'ensemble des arc,  $A \subseteq X^2 \setminus \{(x, x), x \in X\}$

**Notation et prop :**

$n = \text{Card}(X), m = \text{Card}(A)$

$0 \leq m \leq n^2 - n$

**prop :**  $\sum_{x \in X} d^+(x) = \sum_{x \in X} d^-(x) = m$

**chemin :**  $(x_1, \dots, x_k)$  tel que  $\forall i \in X \text{ et } \forall 1 \leq i < k \quad (x_i, x_{i+1}) \in A$

**chemin élémentaire :** ne passe pas deux fois par le même sommet  $i \neq j \Rightarrow x_i \neq x_j$

**chemin simple :** ne passe pas deux fois pas un même arc

**prop :** élémentaire  $\Rightarrow$  simple, mais réciproque fausse

**circuit :**  $(x_1, \dots, x_k)$  tel que  $\forall i \in X \text{ et } \forall 1 \leq i < k \quad (x_i, x_{i+1}) \in A \text{ et } (x_k, x_1) \in A$

**sous graphe induit :**  $Y \subseteq X$  le sous graphe de G induit par Y est  $(Y, B)$  avec  $B = A \cap (Y \times Y)$

**graphe partiel :**  $B \subseteq A$  le graphe partiel de G induit par B est  $(X, B)$

**forte connexité :**  $\forall x \neq y \in X, \exists$  un chemin allant de x vers y et un chemin allant de y vers x.

**prop :** un graphe est fortement connexessi il existe un chemin passant au moins une fois par chaque sommet.

**composante fortement connexe :**  $C \subseteq X$  tq le sous graphe induit par C est fortement connexe et l'ajout d'un ou plusieurs sommets fait perdre la forte connexité.

**partition d'un graphe :**  $C_1, \dots, C_k$  les cfc de G, on a :

- (i)  $\bigcup_{l=1}^k C_l = X$
- (ii)  $i \neq j \Rightarrow C_i \cap C_j = \emptyset$
- (iii)  $\forall i \quad C_i \neq \emptyset$
- (iv)  $\forall i \quad C_i$  induit un sous graphe fortement connexe maximal (pour l'inclusion) pour la forte connexité.

**représentation d'un graphe orienté :**

Matrice d'adjacente :  $M = (m_{x,y})_{x,y \in X}$  avec

$$m_{x,y} = \begin{cases} 1 & \text{si } (x, y) \in A \\ 0 & \text{sinon} \end{cases}$$

Liste d'adjacence des successeurs (ou prédécesseurs) : Tableau à n cases, chaque case correspond à un sommet et contient la liste chainée de ses successeurs (ou prédécesseurs)

### plus court et plus long chemin :

circuit absorbant pour la recherche d'un pcch :  $p(C) < 0$

**Algorithme de Dijkstra :** Comme quels courts de un nodo al todo el resto graph oriente etman oriente.

Il existe au moins un chemin optimal élémentaire.

**principe :** A l'itération courante : on a déjà trouvé un pcch de s vers certains sommets constituant un ensemble S.

On définit pour  $x \in X$  :

$\Pi(x) =$  poids d'un pcch de s à x ne passant en plus de x que par des sommets de S (on suppose qu'il existe)

$\text{père}(x) =$  prédécesseur immédiat de x sur un tel chemin ( $\text{père}(x) \in S$ )

**pseudo code :**

- ❖  $S \leftarrow \{s\}$
- ❖  $\Pi(s) \leftarrow 0$
- ❖ Pour  $x \neq s, \Pi(x) \leftarrow +\infty$
- ❖ pivot  $\leftarrow s$
- ❖ pour j allant de 1 à  $n-1$ 
  - pour  $x \in (X \setminus S) \cap \Gamma^+(pivot)$ 
    - Si  $\Pi(x) > \Pi(pivot) + p(pivot, x)$ 
      - $\Pi(x) \leftarrow \Pi(pivot) + p(pivot, x)$
      - $\text{père}(x) \leftarrow pivot$
  - déterminer pivot  $\in X \setminus S$  tel que  $\Pi(pivot) = \min_{x \in X \setminus S} \Pi(x)$
  - $S \leftarrow S \cup \{pivot\}$

**prop :**

- pour tout sommet y non dans S, s'il existe au moins un arc dont l'origine est dans S et l'extrémité est y alors  $\Pi(y) = \min_{x \in S, (x,y) \in A} [\Pi(x) + p(x, y)]$

-  $\Pi(pivot)$  est la longueur d'un pcch de s à pivot et  $\text{père}(pivot)$  son prédécesseur dans ce pcch.

**Complexité :** (matrice des poids)  $O(n^2)$

**Utilisation :**

	$\Pi(x), \text{père}(x)$					
S	s	a	b	c	d	e
init	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$s, c$	0	3s	$+\infty$	1s	$+\infty$	$+\infty$
$s, c, d$	0	3s	5c	1s	3d	$+\infty$
$s, c, d, a$	0	3s	4a	1s	3d	16d
$s, c, d, a, b$	0	3s	4a	1s	3d	5b
$s, c, d, a, b, e$	0	3s	4a	1s	3d	5b

pcch de s à e : voir la dernière ligne

$s \rightarrow a \rightarrow b \rightarrow e$

$\rightarrow$  min  $\Pi(x)$  lettres

nombre de pas

definis S

lettres

fère

partida

punto

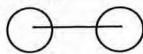
definis S

Lemme 2 : si  $G$  sans cycle  $m \leq n - 1$

Corollaire : Un arbre a  $n-1$  arêtes, et le nombre

d'arbres est inférieur ou égal à  $\binom{n(n-1)}{n-1}$

isthme : arête de  $G$  dont la suppression fait croître le nombre de cc de  $G$



Lemme 3 :  $a \in A$  est un isthme ssi il n'appartient à aucun cycle de  $G$

THM : les 6 proposition suivantes sont équivalentes :

- $G$  est connexe sans cycle (ie un arbre)
- $G$  est connexe et  $m=n-1$
- $G$  est sans cycle et la suppression d'une arête le déconnecte
- $G$  est sans cycle et  $m=n-1$
- $G$  est sans cycle et l'ajout d'une arête crée un cycle
- $\forall x \neq y \in X$  il existe une chaîne unique entre  $x$  et  $y$

### Algorithme de Kuskal (recherche d'un arbre couvrant de poids min) :

principe :

- trier les arêtes selon les poids croissants

- examiner les arêtes selon les poids croissants :

l'arête examinée est sélectionnée si elle ne crée pas de cycle avec les arêtes déjà sélectionnées

On s'arrête quand on a  $n-1$  arêtes.

Pour savoir si on crée un cycle : on sélectionne  $\{x,y\}$  si  $x$  et  $y$  sont dans des composantes connexes distinctes

On construit un tableau CC des cc de  $G$  : au début les cases sont numérotées de 1 à  $n$  et représentent les sommets. On regarde chaque arête et si les numéros des sommets sont différents on leur met le même numéro. On s'arrête lorsque tous les numéros sont les mêmes.

Tableau associé à l'arbre (au final sera de poids minimum)

pseudo code :

- ❖ pour  $i$  allant de 1 à  $n$   $CC[i] \leftarrow i$
- ❖ trier les arêtes par poids croissant :
- ❖  $T \leftarrow \emptyset$
- ❖ Tant que  $|T| \leq n-1$ 
  - soit  $a = \{x,y\}$  la première arête de  $L$
  - $L \leftarrow L \setminus \{a\}$
  - si  $CC[x] \neq CC[y]$ 
    - $T \leftarrow T \cup \{x,y\}$
    - aux  $\leftarrow CC[y]$
    - pour  $i$  allant de 1 à  $n$ 
      - si  $CC[i] = \text{aux}$ 
        - ♦  $CC[i] \leftarrow CC[x]$

Complexité :  $O(m \log(m) + n^2)$

ab	-1
cf	0
cb	2
ac	3
bf	4
dc	5
de	6
ea	7
df	8

Pour avoir un arbre couvrant de poids max on trie par poids décroissant

### Algorithme de Prim :

principe : On fait croître un arbre jusqu'à ce qu'il devienne couvrant. Pour cela on répète  $n-1$  fois l'itération suivante :

sélectionner l'arête de plus petit poids parmi les arêtes ayant une extrémité dans l'arbre en construction et l'autre à l'extérieur.

Complexité :  $O(n^2)$

Rque : l'algo de Prim prend en compte l'ordre des poids

Pour avoir un arbre couvrant de poids max on prend à chaque fois l'arête de poids max

### Parcours d'un graphe

arborescence de racine  $r$  : graphe orienté tq

- si on oublie l'orientation des arcs on obtient un arbre
- il existe un chemin unique depuis  $r$  vers tout sommet de l'arborescence.

### parcours d'un graphe orienté :

- ❖ au départ aucun sommet marqué, aucun arc traversé
- ❖ marquer  $r$
- ❖ tant qu'il existe un sommet  $x$  marqué et un arc  $(x,y)$  non traversé
  - choisir un tel arc  $(x,y)$
  - traverser  $(x,y)$
  - si  $y$  n'est pas marqué
    - marquer  $y$
    - père( $y$ )  $\leftarrow x$

### parcours d'un graphe non orienté :

- ❖ au départ aucun sommet marqué aucune arête traversée
- ❖ marquer  $r$
- ❖ tant qu'il existe un sommet  $x$  marqué et une arête  $\{x,y\}$  non traversée
  - choisir une telle arête  $\{x,y\}$
  - traverser  $\{x,y\}$
  - si  $y$  n'est pas marqué
    - marquer  $y$
    - père( $y$ )  $\leftarrow x$

Complexité d'un parcours :  $O(n+m)$  si dans

la boucle on considère les arcs (ou arêtes) et les sommets  $O(1)$  fois

### parcours "marquer-examiner" :

examiner :

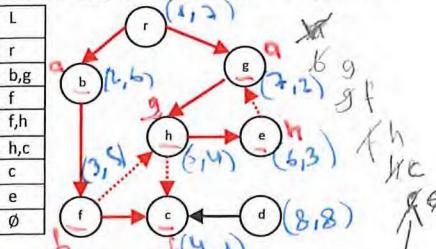
- ❖ pour tout arc  $(x,y)$ 
  - traverser  $(x,y)$
  - si  $y$  n'est pas marqué
    - marquer  $y$
    - père( $y$ )  $\leftarrow x$
    - mettre  $y$  dans une structure d'attente  $L \leftarrow L \cup \{y\}$

### parcours à partir de $r$ :

- ❖ marquer  $r$
- ❖  $L \leftarrow \{r\}$
- ❖ tant que  $L$  est non vide
  - soit  $x$  le premier élément de  $L$
  - retirer  $x$  de  $L$
  - examiner  $x$

lorsque  $L$  est une file : On obtient l'arborescence des pcch issus de  $r$  par rapport au nombre d'arcs

→ alg de pcch pour des poids de 1 en  $O(m)$  si on code le graphe avec les listes d'adjacence.



### Parcours en profondeur :

#### DFS(x) : depth first search

- ❖ marquer  $x$
- ❖ pour tout arc de la forme  $(x,y)$ 
  - traverser  $(x,y)$
  - si  $y$  n'est pas marqué
    - père( $y$ )  $\leftarrow x$
    - appliquer DFS( $y$ )

Complexité :  $O(n+m)$

### Numérotations préfixe et postfixe :

On attribut le n° préfixe courant lorsqu'on rencontre pour la première fois le sommet dans DFS et le n° postfixe courant lorsqu'on le rencontre pour la dernière fois.

pseudo code :

- ❖ marquer  $x$
- ❖ attribuer à  $x$  le n° préfixe courant
- ❖ pour tout arc de la forme  $(x,y)$ 
  - traverser  $(x,y)$
  - si  $y$  n'est pas marqué
    - père( $y$ )  $\leftarrow x$
    - appliquer DFS( $y$ )
- ❖ attribuer à  $x$  le n° postfixe courant

CFC

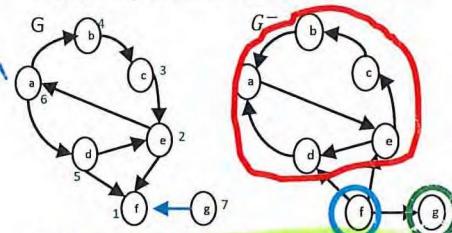
### Détermination des composantes fortement connexes d'un graphe orienté G=(X,A) :

principe :

1. Appliquer DFS autant de fois que nécessaire pour obtenir les n° postfixes de tous les sommets

2. Dans  $G^- = (X, B)$  avec  $B = \{(x, y) \text{ tq } (y, x) \in A\}$  appliquer DFS jusqu'à atteindre tous les sommets en considérant à chaque application le sommet de plus grand n° postfixe parmi les sommets non encore atteints

3. Les sommets atteints à chaque application de DFS dans la phase 2. constituent une cfc de  $G$



### Flot maximum et coupe minimum :

$G = (X, A)$  orienté valué par  $c : A \rightarrow \mathbb{R}_+$ ,  $s$  (source) et  $t$  (puits) deux sommets de  $G$

flot :  $f$  def sur  $A$  vérifiant

(i)  $\forall a \in A \quad 0 \leq f(a) \leq c(a)$

(ii)  $\forall x \in X \setminus \{s, t\} \quad \sum_{y \in \Gamma^-(x)} f(x, y) = \sum_{y \in \Gamma^+(x)} f(x, y)$

valeur d'un flot :

$$v(f) = \sum_{x \in \Gamma^+(s)} f(s, x) - \sum_{y \in \Gamma^-(s)} f(y, s)$$

$$\text{Rque : } v(f) \leq \sum_{x \in \Gamma^+(s)} c(x, s)$$

coupe : bipartition  $(S, \bar{S})$  de  $X$  tq

(i)  $S \cap \bar{S} = \emptyset$  et  $S \cup \bar{S} = X$

(ii)  $s \in S$  et  $t \in \bar{S}$

$$\text{capacité : } c(S, \bar{S}) = \sum_{\substack{(x,y) \in A \\ x \in S \\ y \in \bar{S}}} c(x, y)$$

Thm : si  $f$  est un flot et  $(S, \bar{S})$  une coupe alors

$$v(f) = \sum_{\substack{(x,y) \in A \\ x \in S \\ y \in \bar{S}}} c(x, y) - \sum_{\substack{(z,t) \in A \\ z \in \bar{S} \\ t \in S}} c(z, t)$$

Corollaire :

- si  $f$  est un flot et  $(S, \bar{S})$  une coupe alors

$$v(f) \leq c(S, \bar{S})$$

- si  $f_{max}$  est un flot de valeur maximum et  $(S_{max}, \bar{S}_{max})$  une coupe de capacité minimum alors il y a égalité dans l'inégalité ci-dessus ssi les arcs  $(x, y)$  tq  $x \in S_{max}$  et  $y \in \bar{S}_{max}$  vérifient  $f_{max}(x, y) = c(x, y)$  et les arcs  $(z, t)$  tq  $z \in \bar{S}_{max}$  et  $t \in S_{max}$  vérifient  $f_{max}(z, t) = 0$

Thm : Soit  $v_{max}$  la valeur max d'un flot et  $c_{min}$  la capacité min d'une coupe.

Alors  $v_{max} = c_{min}$

de plus si  $v(f) = c(S, \bar{S})$  alors  $f$  est un flot de valeur max et  $(S, \bar{S})$  une coupe de capacité min

### Algorithme de Ford et Fulkerson :

Complexité des algo récursifs :  $E(k) = k + \sum_{y \in \Gamma^+(k)} c(y)$ .

