Fiche de révision INF101 *

Antonin GODARD

23 décembre 2018

Ce document est à lire après avoir suivi le cours. Le document fait donc l'hypothèse que le lecteur connaît au préalable les notions dont le document parle.

Complexité : mesure du temps nécessaire de calcul pour finir l'algorithme. Soit (P) un problème. Soit I une instance de (P). Soit A un algorithme pour (P). Soit $f_A(I)$ le nombre d'opérations élémentaires effectuées pour compléter I. On définit alors c_A par :

$$c_A : \left| \begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & \max(f_A(I)) \end{array} \right.$$

Structures de Données

Variable : emplacement précis en mémoire. Possède une adresse et une taille variable.

Tableau : collection de variables. Contient des cases numérotées. Caractérisé par :

- son adresse
- sa taille
- nature des entités conservées (de même type pour un tableau).

On accède à l'adresse via :
$T[i] = T[0] + i \times t$. (t : taille).

Liste chaînée : suite de données homogènes (même type) auxquelles on accède de proche en proche, chaque donnée contenant l'adresse de l'élément suivant. Avantage : pas de problème de dimensionnement. Inconvénient : plus d'accès direct.

Pile : collection ordonnée de données respectant la stratégie « Last in, first out » (LIFO).

File : De même, « First in, first out » (FIFO).

Arbre : Un arbre A est de la forme $A = (R, \underbrace{A_1, \ldots, A_k}_{\text{Arbres}})$. R est la racine l'arbre A.

Terminologie : Soit N un nœud de A.

- Les fils de N sont les racines des sous-arbres de N ;
- Un nœud interne est un nœud admettant au moins 1 fils ;
- une feuille est un nœud n'ayant pas de fils ;
- tout nœud autre que la racine admet un père unique.

Profondeur de N dans A :

$$\text{prof}(N) = \left| \begin{array}{l} 0 \textit{ si } N = R \\ \text{prof(père de } N) + 1 \textit{ sinon.} \end{array} \right.$$

Hauteur de A : $h = \max_{N \in A}(\text{prof}(N))$.

Arbre complet : Chaque nœud a deux fils.

Arbre parfait : $\forall i$, $(0 \leqslant i < h)$, le niveau i est saturé.

Tri

Problème : On dispose d'une collection de n éléments, on souhaite les ordonner sous forme croissante.

Tri insertion : À la i^{e} étape, on considère les i premiers éléments triés, l'élément $i+1$ est alors inséré dans les éléments déjà triés par comparaison.

Algorithm 1: Tri Insertion $O(n^2)$

```
Data: T tableau d'indices 1 à n
for i variant de 2 à n do
    j ⟵ i ;
    cle ⟵ T[j] ;
    while j ⩾ 2 et T[j] > cle do
        T[j] ⟵ T[j − 1] ;
        j ⟵ j − 1 ;
    T[j] ⟵ cle ;
```

Tri sélection : On cherche le plus petit élément, on le met à la 1^{re} position ; on cherche le 2^{e} plus petit, etc.

Algorithm 2: Tri Sélection $O(n^2)$

```
Data: T tableau d'indices 1 à n
for i variant de 1 à n − 1 do
    indicePetit ⟵ i ;
    min ⟵ T[i] ;
    for j variant de 1 à n + 1 do
        if T[j] < min then
            indicePetit ⟵ j ;
            min ⟵ T[j] ;
        échanger(T, i, indicePetit) ;
```

Tri rapide : On utilise une fonction nommée *partition* sur une liste qui sélectionne une donnée *cle*, et qui range les données plus petites que *cle* à gauche de *cle*, puis range à droite de *cle* les éléments plus grands que *cle*. Il est défini récursivement la plupart du temps.

Algorithm 3: partition(g,d)

```
Data: Données du tableau T entre
      les indices g et d
cle ⟵ T[g] ;
i ⟵ g + 1 ;
j ⟵ d ;
while j ⩾ i do
    while i ⩽ j et T[i] ⩽ cle do
        i ⟵ i + 1;
    while T[j] > cle do
        j ⟵ j − 1;
    if i < j then
        échanger(T,i,j) ;
        i ⟵ i + 1;
        j ⟵ j − 1;
échanger(T,g,j);
return j
```

Algorithm 4: tri_rapide(g,d) $O(n \log n)$

```
Data: j
if g ⩽ d then
    j ⟵ partition(g,d);
    tri_rapide(g,j − 1);
    tri_rapide(j + 1,d);
```

Tri par arbre binaire de recherche (ABR) : arbre binaire vérifiant :

$\forall N \in A$, la clé de N est $\geqslant$ aux clés du sous-arbre gauche de N et $\leqslant$ aux clés du sous-arbre droit de N.

Le tri ABR se décompose en deux étapes :

1. Construction de l'ABR
2. Parcours de l'ABR

Un parcours infixe trie les clés de l'ABR, le tout en $O(n^2)$.

Tri tas : on définit un tas comme un arbre dont les clés vérifient :

$\forall N \in A$, la clé de N est $\geqslant$ à la clé des fils de N.

Le tri tas se décompose en deux étapes :

1. construction du tas en $O(n \log(n))$
2. exploitation du tas en $O(n \log(n))$

D'où la complexité en $O(n \log(n))$.

Algorithm 5: montée(p)

```
Data: i est un entier et cle est du
      même type que les données
      du tas
i ⟵ p ;
cle ⟵ T[p] ;
while i ⩾ 2 et cle ⩾ T[⌊i/2⌋] do
    T[i] ⟵ T[⌊i/2⌋] ;
    i ⟵ ⌊i/2⌋ ;
T[i] ⟵ cle ;
```

Principe du tri : on échange la racine avec la dernière feuille du tas. L'ex-racine est à la fin du tableau, on n'y touche plus. On va alors reconstruire le tas avec les données restantes en faisant descendre la donnée échangée.

*Cours suivi avec Olivier HUDRY

Algorithm 6: descente(q, p)

Data: p est le nombre de données et la donnée d'indice q est diminuée

$trouve \longleftarrow FAUX$;
$i \longleftarrow q$;
$cle \longleftarrow T[q]$;
while *not trouve et* $2i \leqslant p$ **do**
  if $2i == p$ **then**
    $indice\_grand \longleftarrow p$
  else
    if $T[2i] \geqslant T[2i+1]$ **then**
      $indice\_grand \longleftarrow 2i{+}1$;
  if $cle < T[indice\_grand]$ **then**
    $T[i] \longleftarrow T[indice\_grand]$;
    $i \longleftarrow indice\_grand$;
  else
    $trouve \longleftarrow VRAI$;
$T[i] \longleftarrow cle$;

Algorithm 7: Tri tas $O(n \log(n))$

Data: p est un entier
for $2 \leqslant p \leqslant n$, $p \leftarrow p+1$ **do**
  montée(p) ;
for $n \geqslant p \geqslant 2$, $p \leftarrow p-1$ **do**
  échanger$(T, 1, p)$;
  descente$(1, p-1)$;

Fonctions de hachage

Pour gérer une collision en hachage, on peut faire appel à un « hachage linéaire ». Si, lors de la construction de la table, on veut ranger un mot mais que la place est déjà occupée, alors on se déplace cycliquement vers la droite (càd qu'on repart du début du tableau à chaque fois), puis on range le mot dans une case libre.

Codage de Huffman

Problème : on souhaite envoyer un texte sous forme binaire afin d'optimiser la longueur du texte codé.
On définit :

- $\text{occ}(x)$ = nombre d'occurence de x
- $\text{l}(x)$ = longueur de x

Le codage respecte la **règle du préfixe** : tout codage de caractère ne peut être le préfixe du codage d'un autre caractère. On associe un arbre binaire à ce codage dont les feuilles correspondent au caractère et chaque branche gauche représente un 0 et chaque branche droite représente un 1.
Long. du texte : $L = \sum_x \text{occ}(x) \times p(x)$
Propositions :

1. tout arbre optimal est *complet*
2. $\exists$ un arbre optimal dans lequel les 2 caractères les moins fréquents (α & β) sont associés aux feuilles de profondeur maximum
3. Idem, avec α et β de même père, donc frères

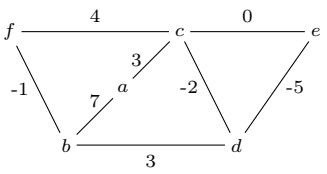
Théorie des graphes

Graphes Non-Orientés

Un graphe non-orienté $G = (X, A)$ est un couple d'ensembles finis avec :

- $X = \{\text{sommets}\}$
- $A = \{\textbf{arêtes}\}$, les arêtes étant de la forme $\{x, y\}$ (le sens compte).
- $m = \Omega(X)$ (cardinal)
- $n = \Omega(Y)$
- Pour une arête $a = \{x, y\}$:
  - a est adjacente à x et y
  - x et y sont les extrémités de a
  - x et y sont voisins
- Voisinage de x
  $V(x) = \{y \in X / \{x, y\} \in A$
- $d(x) = \Omega(V(x))$ et $\sum_{x \in X} d(x) = 2m$
- Chaîne : suite de sommets $x_1, x_2, \ldots, x_k$ avec :
  1. $\forall 1 \leqslant i \leqslant k \quad x_i \in X$
  2. $\forall 1 \leqslant i \leqslant k-1 \quad \{x_i, x_{i+1}\} \in A$
  3. $\forall 1 < i < k \quad x_{i-1} \neq x_{i+1}$
- Chaîne simple (resp. élémentaire) : ne passe pas $2\times$ par une même arête (resp. sommet)
- $x_1, x_2, \ldots, x_k$ élémentaire $\iff$ $\forall 1 \leqslant i, j \leqslant k \quad x_i \neq x_j$
- Cycle : suite de sommets $x_1, x_2, \ldots, x_k, (x_1)$ avec :
  1. $\forall 1 \leqslant i \leqslant k \quad x_i \in X$
  2. $\forall 1 \leqslant i \leqslant k-1 \quad \{x_i, x_{i+1}\} \in A \, et \, \{x_k, x_1\} \in A$
  3. $\forall 1 < i < k \quad x_{i-1} \neq x_{i+1} \, et \, x_2 \neq x_k \, et \, x_1 \neq x_{k-1}$
- G est dit connexe si $\forall x, y \in X \quad \exists$ chaîne entre x et y. On définit de même les composantes connexes qui sont les sous-graphes connexes de G.
- Une arête de G est un *isthme* si la suppression de cette arête fait croître le nombre de composantes connexes de G.
- Un arbre est un graphe connexe sans cycle.

Exemple :



Codage d'un graphe :

- Matrice d'adjacence symétrique où a est la 1re colonne ou 1re ligne. Ici :

$$M = \begin{pmatrix} +\infty & 7 & 3 & +\infty & +\infty & +\infty \\ & +\infty & +\infty & 3 & +\infty & -1 \\ & & +\infty & -2 & 0 & 4 \\ & & & +\infty & -5 & +\infty \\ & (sym) & & & +\infty & +\infty \\ & & & & & +\infty \end{pmatrix}$$

- Liste d'adjacence, où chaque ligne décrit les extrémités triées d'un élément avec leurs poids. Ici :

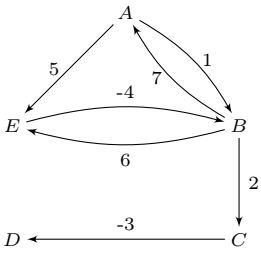
$a \longrightarrow b,7 \longrightarrow c,3$
$b \longrightarrow a,7 \longrightarrow d,3 \longrightarrow f,-1$
$c \longrightarrow a,3 \longrightarrow d,-2 \longrightarrow e,0 \longrightarrow f,4$
$d \longrightarrow b,3 \longrightarrow c,-2 \longrightarrow e,-5$
$e \longrightarrow c,0 \longrightarrow d,-5$
$f \longrightarrow b,-1 \longrightarrow c,4$

Graphe Orientés

Un graphe orienté $G = (X, A)$ est un couple d'ensembles finis avec :

- $X = \{\text{sommets}\}$
- $A = \{\textbf{arcs}\}$, les arcs étant de la forme (x, y) (le sens compte).
- $m = \Omega(X)$ (cardinal)
- $n = \Omega(Y)$
- Pour un arc (x, y) :
  - x est prédécesseur de y
  - y est successeur de x
  - $V^+(x)$ = voisinage extérieur de x
  - $V^-(x)$ = voisinage intérieur de x

Exemple :



On définit alors de manière analogue la matrice d'adjacence de G. La liste d'adjacence distingue la liste des successeurs et la liste des prédécesseurs.

Arbre couvrant de poids minimum

Étant donné un graphe G non-orienté et pondéré, on cherche un arbre couvrant de poids minimum sur ce graphe.

On utilise l'algorithme de Kruskal qui consiste, pour un graphe *connexe* de n sommets, à sélectionner à chaque itération une nouvelle arête du graphe qui ne crée pas de cycle. Il se déroule en deux étapes :

1. trier les arêtes par ordre croissant
2. tant que l'on a pas retenu $n-1$ arêtes, on procède de la sorte : considérer, dans l'ordre du tri, la première arête non examinée, puis appliquer la condition du paragraphe ci-dessus

Algorithm 8: Algo. de Kruskal $O(n^2 + m \log m)$

Data: T l'arbre couvrant de poids minimum

Trier les arêtes $\longrightarrow L$;
for *i de 1 à n* **do**
  $CC(i) \longleftarrow i$;
$ComptL \longleftarrow 1$;
$ComptT \longleftarrow 0$;
while $CompteurT < n-1$ **do**
  Soit $\{x, y\}$ l'arête $L[ComptL]$;
  $ComptL \longleftarrow ComptL + 1$;
  if $CC(x) \neq CC(y)$ **then**
    $ComptT \longleftarrow ComptT + 1$;
    $T(ComptT) \longleftarrow \{x, y\}$;
    $aux \longleftarrow CC(y)$;
    for *i de 1 à n* **do**
      if $CC(i) = aux$ **then**
        $CC(i) \leftarrow CC(x)$;

Plus courts chemins

Cas des graphes pondérés positivement : algorithme de Dijkstra

On cherche une arborescence des plus courts chemins de la racine r vers tous les autres sommets du graphe.
$p(a, b)$ est le poids de l'arc (a, b).

Algorithm 9: Algorithme de Dijkstra $O(n^2)$

$S \longleftarrow \{r\}$;
pivot $\longleftarrow r$;
$\pi(r) \longleftarrow 0$;
$\forall x \neq r, \pi(x) \longleftarrow \infty$;
for *i variant de 1 à* $n-1$ **do**
  for $x \in \overline{S} \cap V^+(pivot)$ **do**
    if $\pi(x) > \pi(pivot) + p(pivot, x)$ **then**
      $\pi(x) \leftarrow \pi(\text{pivot}) + p(\text{pivot}, x)$;
      père$(x) \leftarrow$ pivot;
  déterminer pivot $\in \overline{S}$ $(\pi(\text{pivot}) = \min_{x \in \overline{S}} \pi(x))$;
  $S \longleftarrow S \cup \{\text{pivot}\}$;

Cas des graphes sans circuit : algorithme de Bellman

Soit $G = (X, A)$ orienté pondéré par $p : A \to \mathbb{R}$ *sans-circuit*, et un sommet $r \in X$.
Problème : déterminer, pour $x \neq r$, un *pcch.* de r à x.
plch.
On définit deux attributs :
- $\pi(x)$ = poids d'un pcch. de r à x
- père(x) = prédécesseur de x sur un tel chemin

$$\pi(x) = \min_{y \in V^-(x)} [\pi(y) + p(y, x)]$$

Une **numération topologique** de G orienté est une bijection $\varphi : X \to \{1, 2, \ldots, n\}$ tq. $\forall (x, y) \in A, \varphi(x) < \varphi(y)$.

Algorithm 10: Algorithme de Bellman

Déterminer une numérotation topologique φ et φ^{-1} ;
Pour $x \neq r$, $\pi(x) \longleftarrow +\infty$; $-\infty$
$\pi(r) \longleftarrow 0$;
for *i variant de 2 à n* **do**
  $x \longleftarrow \varphi^{-1}(i)$;
  $\pi(x) \longleftarrow$
  $\min_{y \in V^-(x)} [\pi(y) + p(y, x)]$;
  $\pi(x) \leftarrow \max_{y \in V^-(x)} [\pi(y) + p(y, x)]$;
  père$(x) \longleftarrow y^* \in V^-(x)$ avec
  $\pi(x) = \pi(y^*) + p(y^*, x)$;

Problème de l'ordonnancement

Soit T un ensemble de p tâches t_i $(1 \leqslant i \leqslant p)$. Chaque tâche t_i possède une durée d_i. On doit respecter des contraintes de la forme : avant de commencer x, on doit avoir fini y. On peut réaliser les tâches en parallèle.
Comment réaliser chaque tâche de façon à :
- respecter les contraintes
- minimiser la durée totale de réalisation, $D_{\min}$.

Modélisation : soit $G = (X, A)$ orienté, pondéré, défini par :
- $X = T \cup \{\text{début}, \text{fin}\}$
- $(x, y) \in A \iff x$ doit être réalisée avant y et on pondère (x, y) par d_x
- (début, x) pondéré par 0
- (x, fin) pondéré par d_x

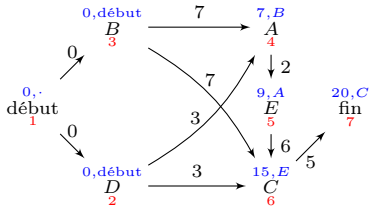
Exemple :

	Durée	Contraintes
A	2	B,D avant A
B	7	/
C	5	B,D,E avant C
D	3	/
E	6	A avant E

$\forall \mathcal{C}$ chemin de début à fin , $D_{\min} \geqslant p(\mathcal{C})$.
G est sans-circuit ssi le problème d'ordonnancement admet une solution. Dans ce cas, considérons un plch. de début à $x \neq$ début : le poids de ce chemin donne la date au plus tôt de début de réalisation de x. D'où :

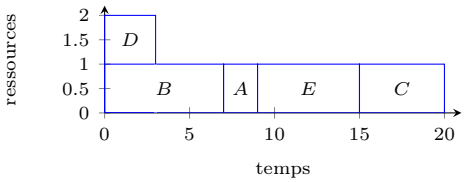
$D_{\min}$ = poids d'un plch. de début à fin

$\Rightarrow$ Algorithme de Bellman :



Numérotation topologique
Marquage de Bellman

Diagramme de Gantt



Parcours de graphes

Parcours « marquer-examiner »

Soit $G = (X, A)$ et $r \in X$.

Algorithm 11: « Moule » d'un parcours de graphe orienté en largeur

Aucun sommet n'est examiné;
Aucun sommet n'est marqué;
Marquer r;
Aucun arc n'est traversé;
while $\exists$ *un sommet* x *marqué et un arc* (x, y) *non traversé* **do**
  if x *non examiné* **then**
    Examiner x
  ;
  traverser (x, y);
  if y *non marqué* **then**
    marquer y;
    père$(y) \longleftarrow x$;

Gestion des objets *marqués* selon une file :
- On ajoute les nouveaux sommets marqués à la fin de la file
- On retire le sommet du début de la file une fois qu'il est examiné

Parcours en profondeur DFS

Soit $G = (X, A)$ et $r \in X$.

cas préfixe
cas postfixe

Algorithm 12: DFS(x)

marquer x;
attribuer à x le numéro préfixe courant;
for $(x, y) \in A$ **do**
  traverser (x, y);
  if y *non marqué* **then**
    père$(y) \longleftarrow x$;
    appliquer DFS à x;
attribuer à x le numéro postfixe courant;

Flux et flots

Soit $G = (X, A)$ un graphe orienté. On dote chaque arc a d'une capacité $c(a)$:

$$c : \left| \begin{array}{ccc} A & \to & \mathbb{R}_+^* \\ a & \mapsto & c(a) \end{array} \right.$$

Soient s et p resp. source et puit.
Un flot f est défini par :

1. $f : \left| \begin{array}{ccc} A & \to & \mathbb{R}_+ \\ a & \mapsto & f(a) \end{array} \right.$

2. $\forall a \in A \quad 0 \leqslant f(a) \leqslant c(a)$

3. $\forall x \in X \backslash \{s, p\}$
$$\sum_{y \in V^-(x)} f(y,x) = \sum_{z \in V^+(x)} f(x,z)$$

On définit ainsi

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

Problème de la coupe minimum

Soit G orienté, $c : A \to \mathbb{R}_+^*$, s et p.

Une coupe $(S, \overline{S})$ est une bipartition de X avec :

1. $S \cap \overline{S} = \varnothing$
2. $S \cup \overline{S} = X$
3. $s \in S$ et $p \in \overline{S}$

On définit $c(S, \overline{S}) = \sum_{(x,y) \in S \times \overline{S}} c(x,y)$.

Soit f un flot de s à p. Soit $(S, \overline{S})$ une coupe dans G. Alors :

$$v(f) = \sum_{(x,y) \in S \times \overline{S}} f(x,y) - \sum_{(z,t) \in \overline{S} \times S} f(z,t)$$

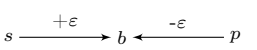
Corollaire 1 : $\forall$ flot f, $\forall$ coupe $(S, \overline{S})$ $v(f) \leqslant c(S, \overline{S})$.

Corollaire 2 : Soit $v_{\max}$ la valeur maximale d'un flot et soit $c_{\min}$ la capacité minimum d'une coupe. Alors $v_{\max} \leqslant c_{\min}$. De plus, s'il existe f et $(S, \overline{S})$ avec $v(f) = c(S, \overline{S})$ alors f est un flot maximum et $(S, \overline{S})$ une coupe de capacité maximum.

Une *chaîne augmentante* (cf. algorithme de marquage suivant) pour f est une suite de sommets de la forme $x_1 x_2 \dots x_k$ avec :

1. $\forall\, 1 \leqslant i \leqslant k \quad x_i \in X$
2. $\forall\, 1 \leqslant i \leqslant k-1$
   - $(x_i, x_{i+1}) \in A$ avec $f(x_i, x_{i+1}) < c(x_i, x_{i+1})$
   - $(x_{i+1}, x_i) \in A$ avec $f(x_{i+1}, x_i) > 0$

On les représente de la sorte :



Algorithm 13: Algorithme de Ford et Fulkerson

```
s est marqué par (Δ, +∞) ;
Les autres sommets sont non marqués ;
Aucun sommet n'est examiné ;
while p non marqué et ∃ un sommet marqué non-examiné do
    Soit x un tel sommet et α la val. absolue de la seconde composante de la marque de x ;
    for y ∈ V⁺(x) non-marqué do
        if f(x,y) < c(x,y) then
            marquer y par (x, min[α, c(x,y) − f(x,y)]) ;
    for z ∈ V⁻(x) non-marqué do
        if f(x,y) > 0 then
            marquer z par (x, −min[α, f(z,x)]) ;
    x est examiné ;
```

Théorème de Mayer

Soit $G = (X, A)$ un graphe orienté.
Problème 1 : déterminer le nombre minimum d'*arcs* à retirer de G pour laisser un graphe fortement connexe = forte arc-connectivité.
Problème 2 : de même mais en enlevant des *sommets* = forte sommet-connectivité.
Problème 3 & 4 : mêmes problèmes avec G non-orienté.
On résout le **problème 1** à l'aide des flots.
On définit C_1 et C_2 comme étant *arcs-disjoints* s'il n'ont pas d'arête en commun.
Sous-problème : Soit a et b 2 sommets de G. Déterminer le minimum $N(a,b)$ d'arcs à retirer afin de plus avoir de chemin de a vers b.
Posons $P(a,b)$ le nombre maximum de chemin arc-disjoint de a vers b.
Théorème : En attribuant des capacités unitaires aux arcs de G, en considérant a et b resp. source et puit, et en considérant f_{ab}^* le flot maximum de a vers b, alors :

$$N(a,b) = P(a,b) = v(f_{ab}^*)$$

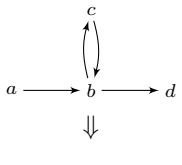
Preuve par transitivité de l'inégalité :

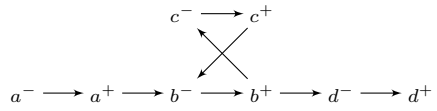
$$N(a,b) \overset{(1)}{\geqslant} P(a,b) \overset{(2)}{\geqslant} v(f_{ab}^*) \overset{(3)}{\geqslant} N(a,b)$$

La solution du **problème 1** est alors donnée par :

$$\min_{a,b \in X} (N(a,b))$$

On résout alors le **problème 2** en remplaçant les sommets de la sorte :





Puis on résout le problème 1 avec le graphe obtenu.
Le **problème 3** est résolu en remplaçant $\{x, y\}$ par (x, y) et (y, x). Attention, après la recherche d'arcs-disjoints il faut ré-ajuster les arcs afin de le transformer en graphe non-orienté.
Le **problème 4** en résolu grâce au **problème 2** puis **1**.

Complexité des algorithmes

Soit π un problème. Soit I une instance de π. La taille de I, notée $|I|$, est le nombre de bits nécessaires pour décrire I.
Pour $I = k \in \mathbb{N}$, $|I| = \lceil \log_2(k+1) \rceil \simeq \log_2 k$. Pour $I = G$ un graphe, avec M la matrice d'adjacence de, $|I| = |M| = n^2$.
Un algorithme A est dit polynomial si sa complexité c_A peut être majorée par un polynôme en la taille de ses données :

$$c_A(|I|) \leqslant Q(|I|)$$

Exemple : tri par ABR en $O(n^2)$ avec $I = (a_1, \dots, a_n)$. $|I| = \sum_{i=1}^n |a_i| = \sum_{i=1}^n \lceil \underbrace{\log_2(a_i + 1)}_{\geqslant 1} \rceil \geqslant n$.
$\Rightarrow$ complexité en $O(|I|^2)$.

Un problème est dit polynomial s'il existe un algo. polynomial le résolvant. Comme « trier des entiers » ou le « pcch. dans G pondéré positivement ».
La recherche d'un pcch. dans G pondéré positivement *ou* négativement n'est pas connu pour être polynomial (mais il pourrait l'être un jour !).
On y voit plusieurs types de problèmes :

- les problèmes d'optimisation, tel que la recherche d'un max. ou d'un min.
- les problèmes de décision, exemples :
  - Nom : cycle eulérien
    Instance : $G = (X, A)$ non-orienté
    $\mathcal{Q}$: existe-t-il un cycle passant exactement 1 fois par toutes les arêtes de G ?
  - Nom : cycle hamiltonien
    Instance : $G = (X, A)$ non-orienté
    $\mathcal{Q}$: existe-t-il un cycle passant exactement 1 fois par tout les sommets de G ?

$P = \{$pb. de décision polynomiaux$\}$
$NP = \{$pb. de décision pour lesquels on peut vérifier en temps polynomial qu'une solution proposée par un devin donne bien la réponse « oui »$\}$

- cycle eulérien $\in P$

— cycle eulérien $\notin P$ mais $\in NP$. Effectivement, sur certains graphes, on peut repérer un cycle hamiltonien à vue d'œil.

Un problème de décision est dit NP-complet si :

1. $\pi \in NP$
2. polynomialité de π entraînerait $P = NP$

Il est dit NP-difficile si sa polynomialité entrainerait $P = NP$.