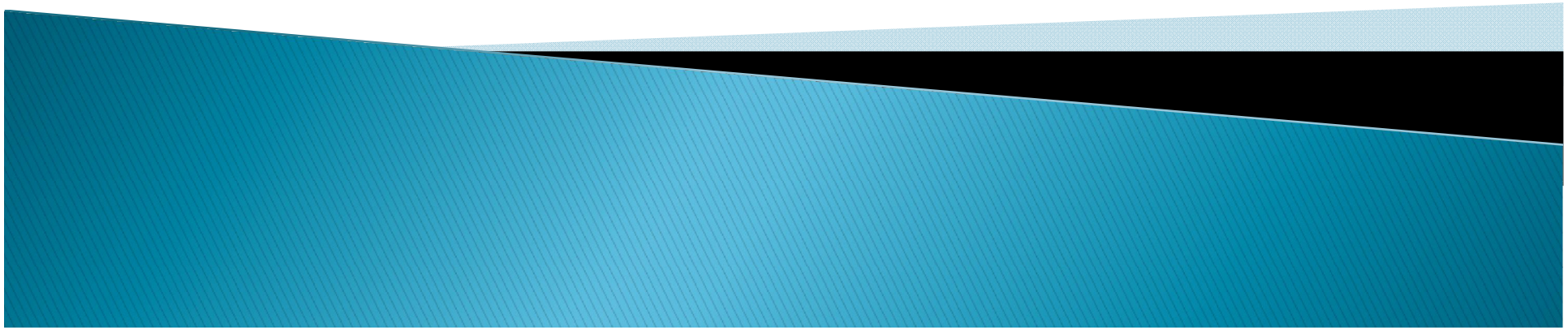
Notion de complexité

Dr H.BELLEILI
Laboratoire Labged
h.belleili@gmail.com



Plan

- Définition
- Complexité des algorithmes,
- Complexité du problème

définition

- Ce que l'on entend par complexité des algorithmes est une évaluation du coût d'exécution d'un algorithme en termes:
  - de temps (complexité temporelle)
  - ou d'espace mémoire (complexité spatiale).
- Dans ce qui suit nous traitons de la complexité temporelle.
- On s'intéresse au coût des actions résultant de l'exécution de l'algorithme, en fonction d'une "taille" *n* des données traitées.
- Ceci permet en particulier de comparer deux algorithmes résolvant le même problème.

Complexité temporelle

- La complexité temporelle d'un algorithme est le nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme.
- Ce nombre s'exprime en fonction de la taille n des données.
- On s'intéresse au coût exact quand c'est possible, mais également:
  - au coût moyen (que se passe-t-il si on moyenne sur toutes les exécutions du programme sur des données de taille n),
  - au cas le plus favorable,
  - ou bien au pire cas.

Exemple

- Recherche d'un élément dans un tableau de taille n:
  - La taille des données:
  - Le cas le plus favorable:
  - Le pire cas:

- Tri d'un tableau de taille n
  - Taille des donnée:
  - Le cas le plus favorable
  - Le pire cas

Travail à faire

- rechercher des algorithmes que vous avez déjà vu ou fais et préciser la taille des données, le cas le plus favorable et le pire cas.

Analyse asymptotique

- L'analyse asymptotique considère le comportement de l'algorithme lorsque n tend vers l'infini.
- Le temps d'exécution dépend de la nature des données:
  - un algorithme de recherche d'une valeur dans un tableau peut s'arrêter dès qu'il a trouvé une occurrence de cette valeur. Si la valeur se trouve toujours au début du tableau, le temps d'exécution est plus faible que si elle se trouve toujours à la fin.
- Nous nous intéresserons à la complexité au "pire des cas" , de de telle sorte à ignorer cette dépendance.
- Dans l'exemple de recherche d'une valeur dans un tableau, on évaluera le temps d'exécution en supposant qu'il faut parcourir tout le tableau pour trouver la valeur cherchée

Exemple

- Fonction SOMMATION (T: tableau): entier
- Debut :
- R := 0
- I := 1
- {1} t1
- Tant que I <=n {2} faire t2
- R:= R+ T[I]; {3} t3
- I:= I+1; {4} t4
- Fin tq
- Fin

- Le temps d'exécution t(n) de cet algorithme s'écrit;

$$T(n)=t1+\sum_{i=1}^{n}(t2+t3+t4)+t2$$

$$T(n)=t1+t2+n(t2+t3+t4)$$

$$tit=t2+t3+t4$$

$$T(n)=t1+t2+n*tit$$

ce qui signifie que le temps d'exécution dépend linéairement de n. On dit que le temps d'exécution est une fonction affine de la taille *n*.

Exemple: Analyse asymptotique

- Analyser le comportement asymptotique de l'algorithme SOMMATION revient à évaluer *t(n)* quand *n* tend vers l'infini. Ou aussi à évaluer $\lim_{n\to\infty} T(n)/tit*n = \lim_{n\to\infty} \frac{t1+t2+n*tit}{tit*n} = 1$
- Autrement dit T(n) est équivalent à l'infini à tit*n ce qui s'écrit: $T(n) \underset{\infty}{\approx} tit*n$
- L'algorithme est donc asymptotiquement linéaire en *n*.

Exemple 2

- Dans l'exemple précédent l'évaluation an "pire des cas" est immédiate puisque t(n) ne dépend pas de la nature des données, ce qui n'est pas le cas de l'exemple-2 suivant:

- Ici, le pire des cas (celui qui conduit au temps d'exécution le plus grand) est celui où la condition *{#3'}* est toujours Vraie.

- En effet dans ce cas

R<-- 2*R *{#3''}* *est exécutée à chaque itération.*

- *{début}*
- R <-- 0
- I <-- 1
- {t1}
- Tant Que I ≤ N *{t2}* Faire
- R <-- R+T[I] *{t3}*
- Si R>1000 *{t3'}* Alors
  - R<-- 2*R *{t3''}*
- FinSi
- I <-- I+1 {t4}
- FinTantQue
- *{fin}*

$$T(n) = t1 + \sum_{i=1}^{n} (t2 + t3 + t3' + t3'' + t4) + t2$$

Evaluation de T(n) (séquence)

- Somme des coûts.
- Traitement1 T1
- Traitement2 T2

$$T = T1 + T2$$

Conditionnel

- Si Condition
  Alors I;

Cas défavorable: lorsque la condition est toujours vrai: $T = T_{condition} + T_I$;

Cas favorable: lorsque la condition est toujours fausse: $T = T_{condition}$

Evaluation de T(n) (embranchement)

- Max des coûts. Cas défavorable
  - si < condition > alors
  - Traitement1 T1
  - sinon
  - Traitement2 T2

  } max(T1, T2)

- Min des couts : cas favorable
  - min (T1,T2)

Boucle pour

Pour j=Début jusqu'à Fin (initialisation, comparaison, incrémentation)

Faire

I(j);

Finfaire

$$T(n) = 1 + \sum_{i=1}^{n} (2 + Tj(I))$$

Exemple: Tri par sélection

```
void tri_selection(const int n, int Tab[])
 {
 // n = Taille du tableau
1 int i,j, indice_min, echange;
 2 for(i = 0; i < (n-1); i++)
3 {
//Recherche de la valeur minimale entre i
    et n

4 indice_min=i;

5  for (j=i+1; j<n; j++)
6 { if (Tab[j] < Tab[indice_min])
    indice_min=j };
 // Echange entre i et indice_min
7 echange = Tab[i];
8 Tab[i]=Tab[indice_min];
9 Tab[indice_min]=echange;
}
```

- Notons T(n) le nombre d'instructions nécessaires pour trier un tableau d'entiers de taille n en utilisant notre fonction "Tri sélection":

- (1) op. elem. (i=0) + (2) op. elem. (comp , increm) + (4) op. elem. (lig 4 + Echange: lignes 7-9) → total 1+6
- Pour un entier i compris entre 0 et n-2, notons TBI(i) le nombre d'instructions réalisées par la boucle intérieure :

$$T(n) = 1 + [(6 + TBI(0)) + (6 + TBI(1)) + ::: + (6 + TBI(n-2))]$$
$$= 1 + 6 . (n - 1) + [TBI(0) + TBI(1) + ::: + TBI(n-2)]$$

Exemple: Tri par sélection (2)

- Boucle Interne BI (TBI)

10 for (j=i+1; j<n; j++)
11 { if (Tab[j] < Tab[indice_min]) indice_min=j };

Une instruction élémentaire d'initialisation : j=i+1. Cette étape coute une (1) unité.

Pour chaque itération j (de i + 1 jusqu' à (n-2)), nous réalisons :

-Deux (2) opérations élémentaires :
- une comparaison j < n; et une opération d'incrémentation j + +;.

- Une (1) opération élémentaire de comparaison : Tab[j] < Tab[indice min]

Si la condition est vraie, une (1) autre opération élémentaire d'affectation indice min=j; est réalisée.

- Dans le pire des cas, (cas défavorable) nous obtenons :

$$TBI(i) = 1 + 4 . (n - 1 - i).$$

Tri par sélection (suite)

$T(n) = 1 + 6 . (n - 1) + [TBI(0) + TBI(1) + ::: + TBI(n-2)]$

$= 1 + 6*(n-1)+[(1+4*(n-1-0)+1+4*(n-1-1)+...+1+4*(n-1-(n-2))]$

$= 1 + 6*(n-1)+(n-1)+[(4*(n-1-0)+4*(n-1-1)+...+4*(n-1-(n-2))]$

$= 7n-6+[(4*(n-1-0)+4*(n-1-1)+...+4*(n-1-(n-2))]$

$= 7n-6+4*[(n-1)+(n-2)+...+1]$

$= 7n-6+4* [(n-1).n]/2$

$T(n) = 2n^2 + 5n - 6$ (cas défavorable)

Tri par sélection (cas favorable)

- La boucle intérieure :
- 1 for (j=i+1; j<n; j++)
- 2 if (Tab[j] < Tab[indice_min]) indice_min=j;
- contient une instruction élémentaire d'initialisation : j=i+1. coût= 1 unité de temps
- Pour chaque itération j (de i + 1 jusqu'à (n-2)), Deux op élém:
  - une comparaison j < n; et une opération d'incrémentation j + +;.
  - Une (1) opération élém. de comparaison : Tab[j] < Tab[indice min]

- Dans le cas favorable, l'autre opération élém. d'affectation indice min=j; n'est pas réalisée.

Dans le cas favorable, nous obtenons :

- TBI(i) = 1 + 3 . (n − 1 − i).

Tri par sélection (cas favorable)

$T(n) = 1 + 6 . (n - 1) + [TBI(0) + TBI(1) + ::: + TBI(n-2)]$

$= 1 + 6*(n-1)+[(1+3*(n-1-0)+1+3*(n-1-1)+...+1+3*(n-1-(n-2))]$

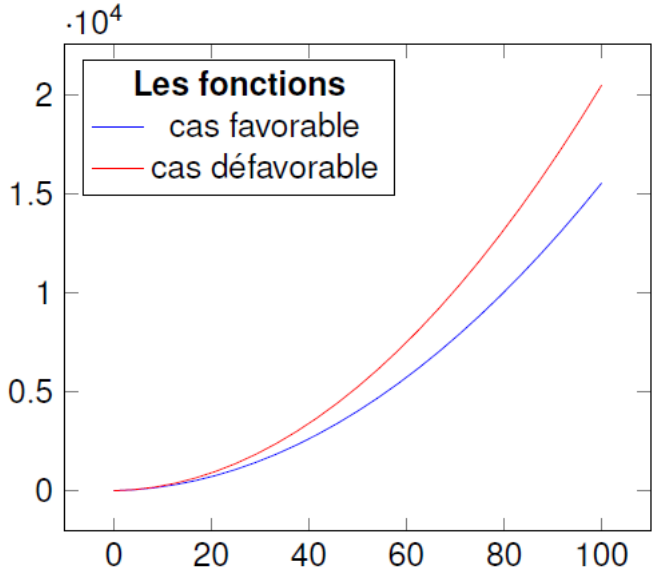
$= 1 + 6*(n-1)+(n-1)+[(3*(n-1-0)+3*(n-1-1)+...+3*(n-1-(n-2))]$

$= 7n-6+[(3*(n-1-0)+3*(n-1-1)+...+3*(n-1-(n-2))]$

$= 7n-6+3*[(n-1)+(n-2)+...+1]$

$= 7n-6+3* [(n-1).n]/2$

$T(n)= 3/2*n^2+11/2*n-6$


·10^4
Les fonctions
cas favorable
cas défavorable
2
1.5
1
0.5
0
0
20
40
60
80
100

Notation O et Θ

Df1 :

On dit que f est asymptotiquement dominée par g noté f=O(g) s'il existe n0, et c>0 tel que |f(n0)| <=c.g(n0)

Exemple :
f(n)=3n+1, g(n)=n
f(n) est en O(n) pour n0=2 et c=4 on a l'inégalité 3n+1<=4n

la notion grand O est appelé symbole de Landau

Notation O et Θ

Df2 :

f est de même ordre de grandeur que g noté $f=\Theta(g)$ lorsque $f=O(g)$ et $g=O(f)$.

Exemple :

$f(n)=3n+1$, $g(n)=n$

$3n+1$ et en $\Theta(n)$.

Nous avons déjà $f=O(g)$, d'autre part pour $n0=2$ et $c=2$ $n<=2(3n+1)$ et donc $g=O(f)$

En général f est une quantité à étudier (temps, nombre d'opérations) et g fait partie d'une échelle de fonction simples(n, nlogn, …) destinée à informer sur le comportement asymptotique de f

Définition 3

Petit o

- f est négligeable devant g : f=o(g) qd $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- f est équivalente à g qd $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$
- Relation entre O et o

f est négligeable devant g → f est dominé par g……(1)

f est équivalente à g → f est de même ordre que g……..(2)

Application (1)

- Soient les fonctions
- $f1(n)=n$, $f2(n)=2^n$,
- $f3(n)=n^2$, $f4(n)=2n$,
- $f5(n)=n^n$, $f6(n)=\log n$, $f7(n)=n!$, $f8(n)= n\log n$
- Pour chaque couple (i, j) dire si on a
- $fi=o(fj)$,
- $fi=O(fj)$
- $fi=\Theta(fj)$.

Application (2)

- *f1 = o(f2) f1(n)=n et f2(n)=2^n*
- on démontre que

$$\lim_{n\to\infty} \frac{f1}{f2} = 0$$

$$\lim_{n\to\infty} \frac{f1}{f2} = \lim_{n\to\infty} \frac{n}{2^n}$$

- Changement de variable: X=2^n n→infini donc X→infini et n=logX donc *f1=o(f2)*

$$\lim_{X\to\infty} \frac{\log X}{X} = 0$$

Application (3)

- Montrer que pour tout entier k, on a $\sum_{i=0}^{n} i^{k+1} = \Theta(n^{k+1})$
- Pour démontrer que *f* est de même ordre de grandeur que *g*, il faut démontrer que f est négligeable devant g. (propriété 2)
- Autrement dit $\lim_{n \to \infty} \frac{\sum_{i=0}^{n} i^{k+1}}{n^{k+1}} = 1 \ ?$
- Donc $\lim_{n\to\infty} \frac{\sum_{i=0}^{n} i^{k+1}}{n^{k+1}} = \lim_{n\to\infty} \frac{0+1+2^k+\ldots n^{k+1}}{n^{k+1}} = 1$

$$\sum_{i=0}^{n} i^{k+1} = \Theta(n^{k+1})$$

Analyse des algorithmes

- L'analyse des algorithmes et la notation O( ) permettent de parler de l'efficacité d'un algorithme spécifique. Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité.

- Les algorithmes de complexité constante O(k) : k premiers éléments d'une liste
- Les algorithmes sous-linéaires, dont la complexité est en général en $O(\log n)$. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal n.
- Les algorithmes linéaires en complexité $O(n)$ ou en $O(n \log n)$ sont considérés comme rapides, comme l'évaluation de la valeur d'une expression composée de n symboles ou les algorithmes optimaux de tri.
- Plus lents sont les algorithmes de complexité située entre $O(n^2)$ et $O(n^3)$, c'est le cas de la multiplication des matrices et du parcours dans les graphes.
- Au delà, les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme lents, sans parler des algorithmes exponentiels (dont la complexité est supérieure à tout polynôme en n dits impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

Méthode de comptage

- principe: on compare C(n+1) avec C(n) on a différents cas possibles:

1. C(n+1)=C(n) la complexité est O(1)
2. C(n+1) = C(n) +1 → O(n)
3. C(n+1) = C(n) + epsilon → O(logn) ou bien C(2n)= C(n) +1
4. C(n+1) = C(n) + n $O(n^2)$
5. C(n+1) = 2* C(n) $O(2^n)$

Algorithme linéaire vs algorithme quadratique

- Comparons ainsi deux algorithmes dont les temps d'exécution *ta* et *tb* seraient les suivants:

- $ta(n) = 100xn$ //*linéaire*
- $tb(n) = 2 * n^2$ //*quadratique*

- pour n = 50 les deux temps sont identiques, mais pour
- n< 50 tb est meilleur que ta
- n=100, ta = 10.000, tb =20.000 ta est meilleur que tb
- n=1000, ta = 100.000, tb = 2.000.000 ta est meilleur que tb
- n = 10.000, ta =1.000.000, tb = 200.000.000.

Temps de calcul

$Taille$	$\log_2 n$	n	$n \log_2 n$	n^2	2^n
10	$0.003\,ms$	$0.01\,ms$	$0.03\,ms$	$0.1\,ms$	$1\,ms$
100	$0.006\,ms$	$0.1\,ms$	$0.6\,ms$	$10\,ms$	$10^{14}\,siecles$
1000	$0.01\,ms$	$1\,ms$	$10\,ms$	$1\,s$	
10^4	$0.013\,ms$	$10\,ms$	$0.1\,s$	$100\,s$	
10^5	$0.016\,ms$	$100\,ms$	$1.6\,s$	$3\,heures$	
10^6	$0.02\,ms$	$1\,s$	$20\,s$	$10\,jours$	

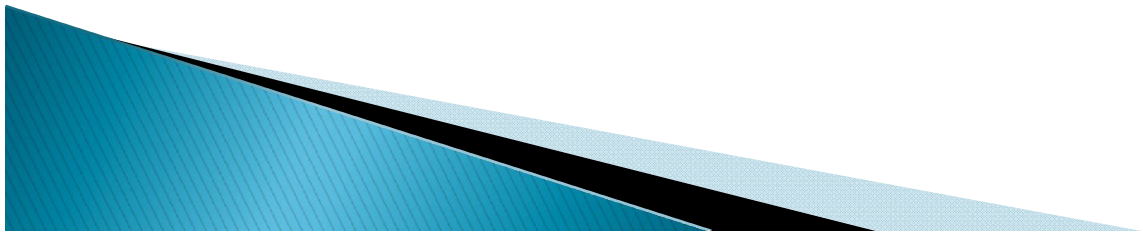
Temps d'exécution

$nTs =$ nombre d'instructions par seconde

nTs	2^n	n^2	$n \log_2 n$	n	$\log_2 n$
10^6	20	1000	63000	10^6	10^{300000}
10^7	23	3162	600000	10^7	$10^{3000000}$
10^9	30	31000	4.10^7	10^9	
10^{12}	40	10^6	3.10^{10}		

Analyse de la complexité

- L'analyse de la complexité traite des problèmes plutôt que des algorithmes. Indépendamment de l'algorithme utilisé, on peut diviser les problèmes entre ceux qui peuvent être résolus en un temps polynomial et ceux qui ne peuvent pas l'être.



Classification des problèmes

- Le but de la théorie de la complexité est la classification des problèmes de décision suivant leur degré de difficulté de résolution.
- Dans la littérature, il existe plusieurs classes de complexité, mais les plus connues sont les suivantes.
- La classe P. c'est la classe des problème pouvant être résolus en un temps polynomial. C'est la classe des problèmes dits facile.
- La classe NP : C'est l'abréviation pour "non deterministic polynomial time". Cette classe renferme tous les problèmes de décision dont on peut associer à chacun d'eux un ensemble de solutions potentielles (au pire exponentiel) tel qu'on puisse vérifier en un temps polynomial si une solution potentielle satisfait la question posée.
- Le terme non déterministe désigne la capacité d'un algorithme à trouver la bonne solution.

Classification des problèmes (suite)

- La classe PSPACE. Problèmes pouvant être résolus en utilisant une quantité d'espace mémoire raisonnable (définie comme une expression polynomiale en fonction de la taille des données) sans considération du temps d'exécution que cela prendra. .
- La Classe EXPTIME. Problèmes ne pouvant être résolus que par des algorithmes de complexité temporelle exponentielle.
- La Classe Indécidable Pour certains problèmes, il est possible de montrer qu'il ne peut exister du tout d'algorithmes pour les résoudre, peu importe le temps ou l'espace qui leur est alloué.

Classification de problèmes

- La théorie de la complexité se limite juste à l'étude des problèmes de décision.
- Définition 1. Un problème de décision est un problème dont la solution est formulée en termes oui/non
- Exemple 1: Étant donné un graphe G = (X,E), existe t-il un chemin de longueur <= L ?
- Exemple 2: Étant donné un graphe G=(X,E), les sommets de X peuvent-il être colorés par au plus m couleurs de telle manière que les sommets adjacents soient de couleurs différentes?

Les problèmes P

- La classe des problèmes polynomiaux déterministes (qui peuvent être résolus en $O(n^k)$ pour un k donné) est appelée classe P. Ces problèmes sont dits faciles

Exemple 1 : Problème DIVCOM.

Instance : un couple d'entiers (*a*, *b*).

Problème : *a* et *b* possèdent-ils un diviseur commun (en dehors de 1) ?

DIVCOM appartient à P ; en d'autres termes, il existe une procédure *divcom*, de complexité polynomiale, telle que:

$$divcom(a,b) = \begin{cases} 1 \ si \ a,b \ possedent \ un \ diviseur \ commn > 1 \\ 0 \ \sin on \end{cases}$$

- Procédure naïve: incrémente un diviseur *d* à partir de 2, jusqu'à ce que *d* divise à la fois *a* et *b*, ou jusqu'à atteindre le plus petit d'entre eux,
  - complexité au pire (lorsqu'il n'existe pas de diviseur commun) est *exponentielle* par rapport à la *taille* des données (si *a* et *b* sont représentés sur *n* bits, le nombre d'itérations peut être de l'ordre de 2^n).

Algorithme d'euclide

- Par contre l'*algorithme d'Euclide* permet de répondre à la question posée en temps polynomial : une caractéristique fondamentale de cet algorithme est que :
  - le nombre de divisions nécessaires pour calculer le pgcd est linéaire O(n) (ie proportionnel au nombre *n* de chiffres utilisés pour représenter *a* ou *b*),
  - et chaque division est de complexité quadratique $O(n^2)$,
  - d'où une complexité $O(n^3)$. Une fois le pgcd calculé, résoudre DIVCOM est immédiat

Exemple 2:

- Le problème de la connexité dans un graphe:
- Étant donné un graphe à s sommets, il s'agit de savoir si toutes les paires de sommets sont reliées par un chemin.
- Pour le résoudre, on dispose de l'algorithme de parcours en profondeur qui va construire un arbre couvrant du graphe à partir d'un sommet.
- Si cet arbre contient tous les sommets du graphe, alors le graphe est connexe.
- Le temps nécessaire pour construire cet arbre est au plus s^2, donc le problème est bien dans la classe P.

Exemple 3:

- Tester le coloriage d'un graphe
- Donnée: Un graphe G = (V;E)
- avec pour chaque sommet $v \in V$ la donnée d'une couleur parmi un nombre fini
- Question : Décider si G est colorié avec ces couleurs : c'est-à-dire si il n'y a pas d'arête de G avec deux extrémités de la même couleur.
- Ce problème est dans la classe P. En effet, il suffit de parcourir les arêtes du graphe et de tester (vérifier) pour chacune si la couleur de ses extrémités est la même.

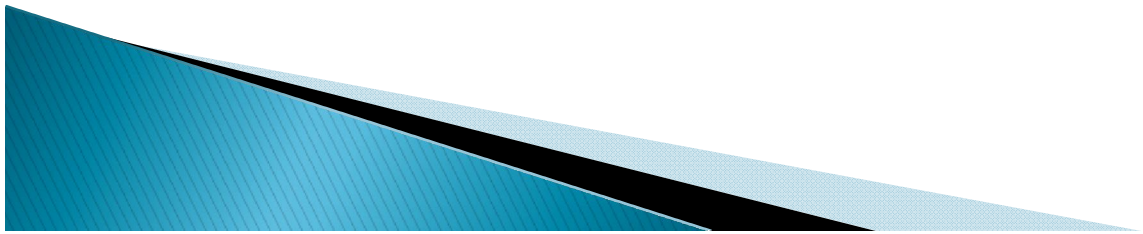
Exemple 4

- Evaluation en calcul propositionnel
- Donnée: Une formule F(x1, x2…, xn) du calcul propositionnel, des valeurs $x1,..,xn \in \{0,1\}$ pour chacune des variables de la formule.
- Question: Décider si la formule F est vraie pour ces valeurs des variables.
- Ce problème est dans la classe P. En effet, étant donnée une formule du calcul propositionnel F(x1,… , xn) et des valeurs pour x1, x2,…, xn appartenant à {0,1}, il est facile de calculer la valeur de vérité de F(x1,…, xn).
- Cela se fait en un temps que l'on vérifie facilement comme polynomial en la taille de l'entrée.
- Beaucoup d'autres problèmes sont dans P.

Conclusion : Les problèmes dans P correspondent en fait à tous les problèmes facilement solubles.

Les problèmes NP

- Un problème NP Non-déterministe Polynomial (et non pas Non polynomial, erreur très courante).
- La classe NP réunit les problèmes de décision pour lesquels la réponse 'OUI' peut être décidée par un algorithme non-déterministe et l'on peut vérifier en un temps polynomial si une solution proposée convient.



Les problèmes NP (2)

- Intuitivement, les problèmes dans NP sont tous les problèmes qui peuvent être résolus en énumérant l'ensemble des solutions possibles et en les testant avec un algorithme polynomial.

Exemple 1

- k-COLORABILITE
- Donnée: Un graphe G = (V;E) et un entier k.
- Décider s'il existe un coloriage du graphe utilisant au plus k couleurs;
- c'est-à-dire décider s'il existe une façon de colorier les sommets de G avec au plus k couleurs pour obtenir un coloriage de G.

Exemple 2

- SAT, Satisfaction en calcul propositionnel
  Donnée: Une formule F = (x1,…, xn) du calcul propositionnel

- Décider si F est satisfiable : c'est-à-dire décider s'il existe $\{x1, x2, \ldots, xn\} \in \{0,1\}^n$ tel que F s'évalue à vraie.

Exemple 3

- La Recherche de cycle hamiltonien dans un graphe peut se faire avec deux algorithmes :
  - le premier génère l'ensemble des cycles (ce qui est exponentiel) ;
  - le second teste les solutions (en temps polynomial).

Analyse (suite)

- Comme nous allons le voir, on sait toutefois montrer que ces trois problèmes sont équivalents au niveau de leur difficulté, et cela nous amène à la notion de réduction,
- c'est-à-dire à l'idée de comparer la difficulté des problèmes.

Analyse

- Pour les trois problèmes on connaît des algorithmes exponentiels :
- tester tous les coloriages, pour le premier,
- ou toutes les valeurs de $\{0,1\}^n$ pour le second,
- ou tous les chemins pour le dernier.

- Pour les trois problèmes on ne connaît pas d'algorithme efficace (polynomial), et on n'arrive pas à prouver qu'il n'y en a pas.
- Mais on sais vérifier en un temps polynomial une solution

Les problèmes les plus durs

- Si on considère une classe de problèmes (P ou NP…), on peut introduire la notion de problème le plus difficile pour la classe. C'est la notion de complétude
- Définition (C-complétude) Soit C une classe de problèmes de décisions.
- Un problème A est dit C-complet, si
  - 1. A est dans C ;
  - 2. tout autre problème B de C est tel que B <= A. (A est au moins aussi difficile que B)

- On dit qu'un problème A est C-dur s'il vérifie la condition 2 de la définition

- Un problème A est donc C-complet s'il est C-dur et dans la classe C.
- Un problème C-complet est donc le plus difficile, ou un des plus difficiles, de la classe C.
- Clairement, s'il y en a plusieurs, ils sont équivalents :
- Tous les problèmes C-complets sont équivalents.

Problématiques de La théorie de la complexité

- La problématique centrale, la plus connue, dans cette théorie est la fameuse question : P = NP? .
- En d'autres termes, s'il est toujours facile de vérifier une solution, est-il aussi facile de trouver une solution ?
- la réponse à cette question résout plusieurs autre grandes questions de cette discipline, mais bien entendu pas toutes les questions.
- Il n'y a pas de raison de croire que l'égalité (P = NP) soit vraie. L'attente de la plupart des informaticiens et mathématiciens est que l'égalité soit fausse.
- Malheureusement, il n'existe pas de preuve pour cette assertion.
- Par conséquent, toute une théorie est construite sur les classes P et NP sans que l'on puisse affirmer s'il existe un problème dans NP qui ne soit pas dans P.
- Par conséquent, on peut se poser la question suivante: quelle est l'intérêt d'une théorie si elle ne peut répondre à la question de connaître le degré de difficulté d'un problème donné.
- Une réponse à cette question vient de la notion de NP-complétude.

La NP-complétude

- La théorie de la NP-complétude concerne la reconnaissance des problèmes les plus durs de la classe NP.
- La notion de la difficulté d'un problème qui est introduite dans cette classe est celle d'une classe de problème qui sont équivalents :
  - si l'un d'eux est prouvé être facile alors tous les problèmes de NP le sont.
- Inversement, si l'un d'eux est prouvé être difficile, alors la classe NP est distincte de la classe P.
- Définition : Un problème de décision est dit NP-complet si tout problème de la classe NP lui est polynomialement réductible.

Les problèmes NP-Complets

- Les problèmes NP-complets sont les problèmes les plus difficiles de la classe NP
- En effet, bien qu'on puisse *vérifier* rapidement (en un temps polynomial) toute solution proposée d'un problème NP-complet, on ne sait pas en *trouver* efficacement,
- Tous les algorithmes connus pour résoudre des problèmes NP-complets ont un temps d'exécution exponentiel en la taille de l'entrée dans le pire cas,
- et sont donc inexploitables en pratique même pour des instances de taille modérée.
- On cherche plutôt des solutions approchées en utilisant des algorithmes d'approximation

Algoritmes d'approximation

- Des algorithmes d'approximation permettent de trouver des solutions approchées de l'optimum en un temps raisonnable pour un certain nombre de programmes. Dans le cas d'un problème d'optimisation on trouve généralement une réponse correcte, sans savoir s'il s'agit de la meilleure solution ;
- Des heuristiques permettent d'obtenir des solutions généralement bonnes mais non exactes en un temps de calcul modéré ;
- Des algorithmes par séparation et évaluation permettent de trouver la ou les solutions exactes. Le temps de calcul n'est bien sûr pas borné polynomialement mais, pour certaines classes de problèmes, il peut rester modéré pour des instances relativement grandes ;
- On peut restreindre la classe des problèmes d'entrée à une sous-classe suffisante, mais plus facile à résoudre.

Exemples de problèmes NP-complets

- problème du stable: trouver dans un graphe fini un ensemble de m sommets non reliés (m étant donné)
- problème du sac à dos : soient un ensemble S de nombre entiers positifs et un entier M. Existe t-il une partie A de S telle que $M = \sum_{i \in A} a_i$
- problème du cycle hamiltonien : soit un graphe fini. Existe t-il un cycle de longueur *n* passant une et une seule fois par tous les *n* sommets.
- problème du 3-coloriage : soit un graphe fini. Peut-on colorier les sommets du graphe à l'aide de 3 couleurs de telle manière que deux sommets adjacents n'aient pas la même couleur.
- problème des machine parallèles. Soient *m* machines identiques et *n* jobs. Chaque job *i* doit être exécuté sans interruption par une des *m* machines pendant un temps pi. Existe t-il une partition des *n* jobs sur les *m* machines de telle manière que le temps de fin de tous les jobs ne dépasse pas le temps T.
- …….

La réduction polynomiale

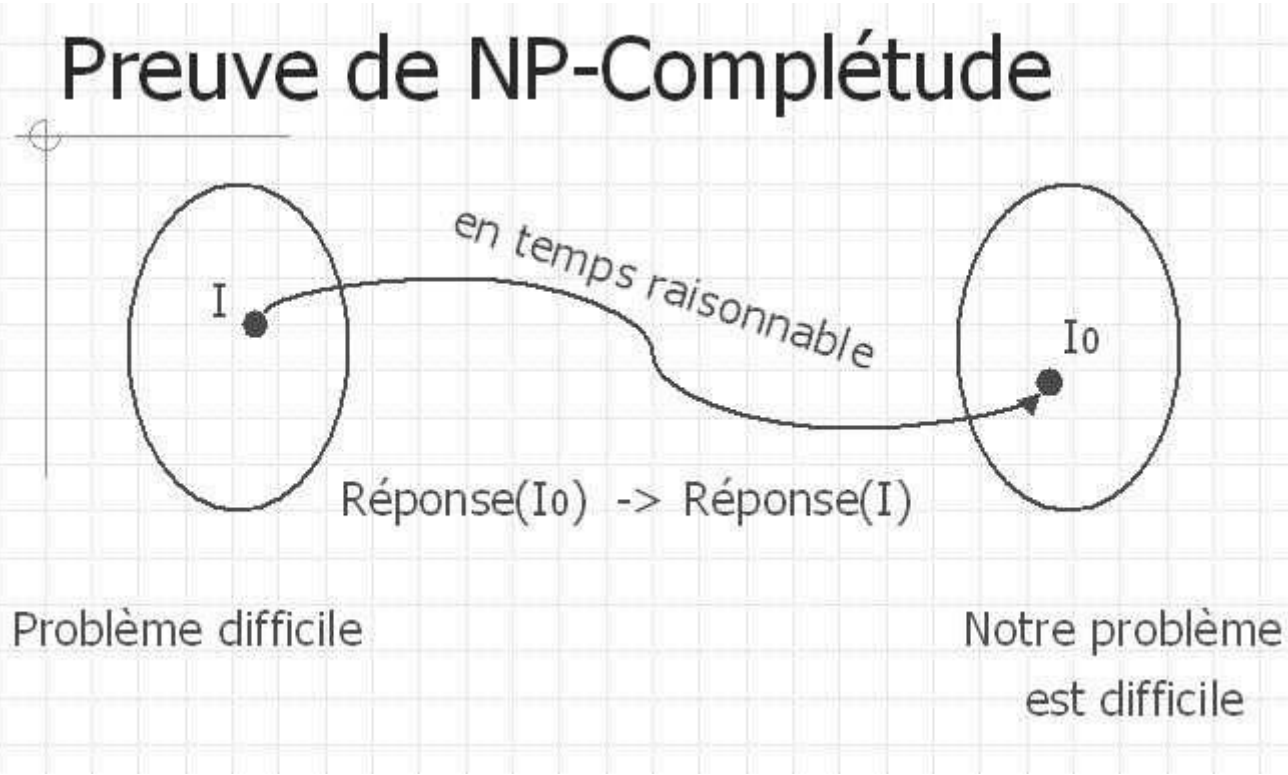
- les techniques de réduction nous permettent de relier deux problèmes entre eux.
- En d'autres termes, la réduction peut être vue comme un moyen pour affirmer qu'un problème est aussi facile ou difficile qu'un autre problème.
- On utilise la réduction pour montrer la difficulté de résolution d'un problème en transformant un problème déjà connu comme étant difficile à résoudre vers notre problème de départ.

Comment prouver la NP-complétude d'un problème en pratique

- Il est utile de signaler que montrer qu'un problème est NP-complet signifie qu'il existe au moins une instance pour laquelle le seul algorithme connu pour résoudre cette instance est exponentiel.
- Réduire un problème P2 à un autre problème P1 revient à montrer que P2 est au moins plus facile à résoudre que P1, et
- P1 est au moins plus difficile à résoudre que P2.
- Cette réduction signifie que P2 est un cas particulier de P1.

- Pour montrer qu'un nouveau problème P1 est NP-complet, on procède comme suit :
  - 1. Montrer que P1 est dans NP
  - 2. Choisir un problème, P2, NP-complet approprié
  - 3. Construire une réduction f, qui transforme P2 vers P1

Preuve de NP-Complétude



Exemple1

- Soient les deux problèmes de décision suivants:
  - Problème 1:
    - Données: – N, un nombre de participants,
      – une liste de paires de participants: les paires d'ennemis
    - Question: Puis-je faire « p » équipes de telle sorte qu'aucune équipe ne contienne une paire d'ennemis.
  - Problème 2:
    - Données: – Un graphe G,
      – un entier k un nombre de couleurs.
    - Question: G est-il k-coloriable?
- Quel lien entre les deux problèmes?

Réduction

- Sachant que le problème k-colorable (noté Pb2)est NP-complet,
- Prouver que le problème 1 (noté Pb1) est NP-complet.

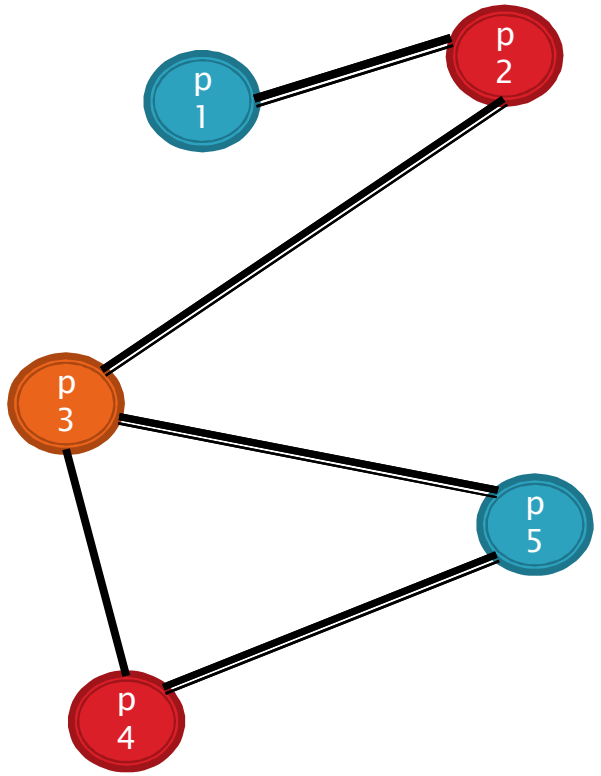
- Cela se fait en cherchant une réduction *red* tq
- *red(Pb2) = Pb1*.
- Je transforme une instance du Pb2 en une instance du Pb1:
  - Les sommets du graphe sont les personnes.
  - Il y a un arc entre deux sommets si les personnes sont ennemies.
  - k = p
- Alors: G est k-coloriable si et seulement si je peux faire «p = k »équipes.

Exemple

- On prend l'exemple de 5 personnes p1, p2,p3,p4,p5,
- Les couples (p1,p2); (p4,p5); (p3,p2);(p3,p5) ; (p3,p4) déterminent les personnes ennemies
- Combien d'équipes je peux faire de manière à ce que les personnes ennemies ne soient pas dans la même équipe?→ Contrainte C
- La réduction revient à trouver la transformation du problème Pb2 (k-colorable) en un problème Pb1
- Ensuite déterminer le nombre de couleurs k de Pb2 revient à trouver le nombre d'équipes que l'on peut réaliser (vérifiant la contrainte C).

Exemple (suite)

- Trouver k le nombre de couleurs revient à trouver le nombre d'équipes,
- Le graphe est 3-colorable
- Donc on peut faire 3 équipes
- Eq1: p1,p5;
- Eq2 : p2,p4;
- Eq3: p3



La réduction est polynomiale

- la construction de G se fait polynomialement en fonction de l'instance du problème 1 (Pb1).

Le problème marche aussi dans l'autre sens

- Si je peux trouver p équipes vérifiant la contrainte alors le graphe correspondant est p-colorable.
- La réduction du Pb1 à Pb2 est aussi polynomiale
- En effet, étant donné une instance de Pb1 je peux la transformer avec un algorithme polynomial au problème Pb2 (k-coloriable).
- Une réponse positive à Pb1 me donne une réponse positive à Pb2;
- Une réponse négative à Pb1 me donne une réponse négative à Pb2

- Remarque: on peut avoir une réduction polynomiale dans un sens et pas dans l'autre.

Définition formelle

- Soient L et L' deux langages de Σ^* correspondant à deux propriétés.
- Une réduction polynomiale de L dans L' est une application red calculable polynomiale de Σ^* dans Σ^* telle que :
- $u \in L$ SSi $red(u) \in L'$.
- On note alors $L \leq_p L'$.

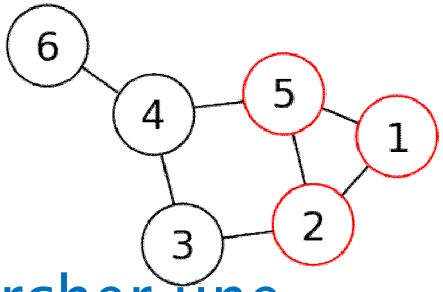
Réduction=transformation

- Par exemple, une réduction d'un problème Pb1 d'équipes en un problème Pb2 de coloriage de graphes sera une transformation:
  - qui associera à toute instance i du Pb1 d'équipes, une instance red(i) du pb2 de coloriage de graphes
  - telle que red(i) a une solution Ssi l'instance i avait une solution.
- cette transformation est polynomiale, i.e. calculable par un algorithme polynomial.
- Remarque: on en déduit donc que la taille de red(i) est bornée polynomialement par rapport à celle de l'instance i.

Exemple 2

- Soient les deux problèmes suivants:
- Clique:
  - Entrée: G=(S,A) --un graphe non orienté
  - k un entier
  - Sortie: Oui, Ssi G contient une clique de cardinal k.
- Indépendant
  - Entrée: G=(S,A) --un graphe non orienté
  - k -- un entier
  - Sortie: Oui, Ssi G contient un ens. indépendant de cardinal k.

Clique maximale et ensemble maximal indépendant



- Le problème de clique maximale:
  - Étant donné un graphe non orienté chercher une clique maximale revient à chercher un sous ensemble maximal (le plus grand) de sommets où chaque pair de sommets est reliée par un arc. Ici la clique maximale est (5,1,2,).
- Le problème de l'ensemble maximal indépendant:
  - Étant donné un graphe non orienté chercher le plus grand sous ensemble de sommets tels que aucune paire de sommet n'est reliée par un arc.

Réduction de clique dans indépendant

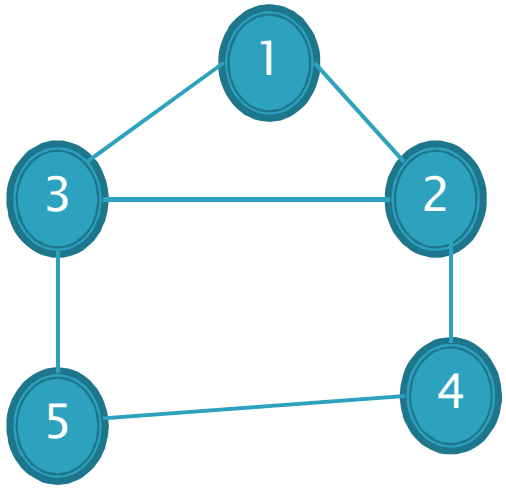
- On peut choisir comme réduction de clique dans indépendant:
- L'application *red* qui à l'instance de clique (G=(S,A),k) associe (G'=(S,A'),k) avec A'={(x,y)/(x,y) $\notin$A et (y,x) $\notin$A}
- Monter que c'est bien une réduction polynomiale de clique dans indépendant consiste à:
  - Monter que c'est correct : pour tout (G,k) clique(G,k) $\Leftrightarrow$ indépendant(G',k)
  - Montrer que la transformation est polynomiale.

Réduction de clique dans indépendant (2)

- Réduction *red* à l'instance (G=(S,A),k) associe (G'=(S,A'),k) avec A'={(x,y)/(x,y) $\notin$A et (y,x) $\notin$A}
- C'est correct,

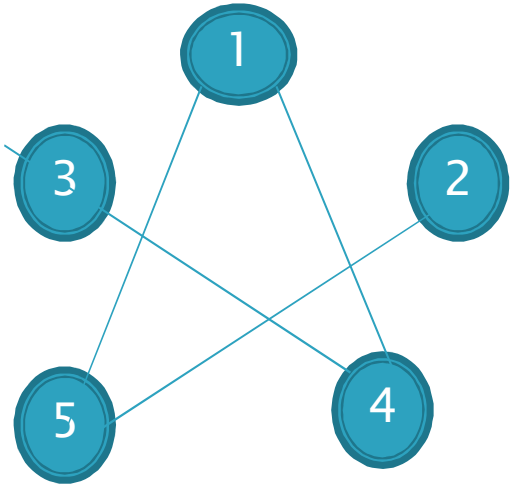
  $\forall$(G,k), clique(G,k) $\Leftrightarrow$ indépendant(G',k);
- (G=(S,A),k) a une clique de cardinal k: C est un ensemble indépendant de cardinal k de G'=red(G).
- Réciproquement: (G'=(S,A'),k) a un ensemble indépendant C de cardinal k: C est une clique de cardinal k de G.
- La transformation est polynomiale: l'algorithme qui calcule *red(G)* est bien polynomial (en $O(card(S)^2)$).

Exemple



Le graphe G clique maximale
C={1,2,3} k=3

Transformation en un problème indépendant



Le graphe G' ensemble maximal indépendant: tous les arcs qui ne sont pas dans G (le complément de G C={1,2,3} k=3

Exemple suite

- L'algorithme qui transforme l'instance du problème de clique maximale en une instance du problème d'ensemble indépendant maximal fonctionne comme suit:
- À chaque arc possible entre deux sommets ne grade que l'arc qui ne se trouve pas dans G.
- Cette transformation est polynomiale.

Merci