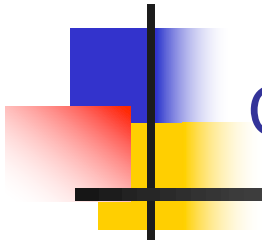
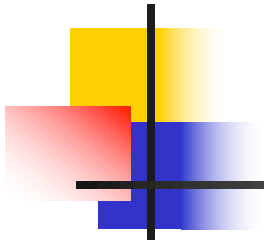


Chapitre 10

Les limites de la puissance algorithmique



Introduction

- Notre sujet d'étude jusqu'ici fut **l'algorithmique** :
  - la conception et l'analyse d'algorithmes spécifiques permettant de résoudre un problème donné.

- Nous allons maintenant aborder **la complexité du calcul** :
  - l'étude des limites de ce que les algorithmes peuvent accomplir.

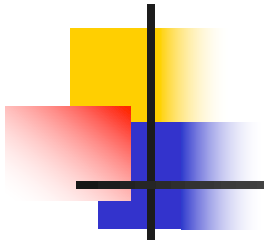
- Nous allons considérer globalement tous les algorithmes permettant de résoudre un problème donné et nous poser la question suivante:
  - **Combien d'opérations élémentaires en pire cas un algorithme doit-il effectuer pour résoudre notre problème?**
    - Exemple: combien de comparaisons, en pire cas, un (n'importe quel) algorithme de tri doit-il effectuer pour trier un tableau?

Introduction (suite)

- Nous cherchons alors une **borne asymptotique inférieure (minorant)** sur le temps d'exécution que doit avoir, en pire cas, n'importe quel algorithme pour résoudre un problème donné
  - On cherche donc la fonction g(n) qui croît le plus rapidement telle que $C_{worst}(n) \in \Omega(g(n))$ pour tous les algorithmes possibles (ou ceux appartenant à une large classe)
  - Exemple: nous allons démontrer qu'un algorithme de tri par comparaisons, quel qu'il soit, doit effectuer $\Omega(n \log n)$ comparaisons, en pire cas, pour trier un tableau de n valeurs

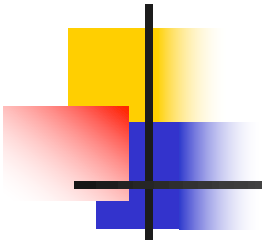
- Un algorithme est donc asymptotiquement **optimal** lorsque son temps d'exécution en pire cas $\in O(g(n))$ (avec le même g(n) que ci-haut)
  - Le minorant g(n) est, dans ce cas, également optimal

- **La complexité du calcul peut donc nous informer sur le degré d'optimalité d'un algorithme**



Introduction (suite)

- De plus…
- Nous allons voir qu'il existe une large classe de problèmes (dits NP-complets) pour lesquels nous n'avons pas d'algorithme efficace et que l'existence d'un algorithme efficace pour un de ces problèmes impliquerait l'existence d'un algorithme efficace pour presque tous les problèmes de décision « bien posés » (ceux de la classe NP)!
- Les problèmes NP-complets sont donc, dans un certain sens, les plus difficiles à résoudre (de la classe NP) mais ils sont omniprésents et ont parfois une importance pratique considérable.
- Nous verrons alors comment démontrer qu'un problème est NP-complet (et donc probablement intraitable).



Minorants triviaux

- Un **minorant trivial** est une borne inférieure (sur le temps d'exécution de n'importe quel algorithme) obtenue en examinant uniquement la taille de la réponse que doit fournir l'algorithme et la quantité d'éléments d'entrée que l'algorithme doit examiner.
  - Ex1: tout algorithme devant générer toutes les permutations de n objets s'exécute en temps $\in \Omega(n!)$ car les n! permutations possibles doivent être produites.
  - Ex2: tout algorithme devant calculer le produit de 2 matrices $n \times n$ s'exécute en un temps $\in \Omega(n^2)$ car il doit examiner $2n^2$ éléments et produire une matrice de n^2 éléments
    - L'algorithme classique prend un temps $\in \Theta(n^3)$ mais l'algorithme de Strassen prend un temps $\in \Theta(n^{\lg(7)})$
- Les minorants triviaux sont souvent trop petits pour être utiles
  - Ex: il est de $\Omega(n^2)$ pour le problème du commis voyageur (voir plus loin) car nous devons examiner les n(n-1)/2 distances entre les n villes. Mais nous ne connaissons présentement aucun algorithme à temps polynomial en n.

Minorants informationnels

- Ces minorants sont obtenus en considérant la **quantité d'information** qu'un algorithme doit extraire pour résoudre un problème.

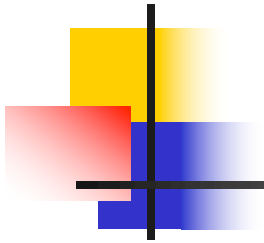
- Ex: un « adversaire » choisit un nombre compris entre 1 et n et vous devez le trouver en posant le minimum de questions (se répondant par oui ou non).

- La meilleure stratégie consiste à poser une question qui divise en deux sous-ensembles, de mêmes tailles, le nombre de solutions possibles.
  - Ex1: ton nombre est-il plus petit que $\lceil n/2 \rceil$?
  - Ex2: ton nombre est-il impair?

- C'est la meilleure stratégie en pire cas, car en pire cas, le nombre recherché sera dans le plus grand des deux sous-ensembles.
  - Il faut alors minimiser la taille du plus grand des deux sous-ensembles et cela est réalisé en divisant par deux.

Minorants informationnels (suite)

- Au début il existe n solutions possibles et, après avoir obtenu la réponse à une question « optimale », il reste $\lceil n/2 \rceil$ solutions possibles.
- Le pire cas est alors obtenu lorsque le nombre de solutions restantes est toujours impair (tant que ce nombre est supérieur à 2).
  - C'est le cas lorsque $n = 2^k + 1$
  - Après 1 question, il reste $\lceil (2^k + 1)/2 \rceil = 2^{k-1} + 1$ solutions possibles
  - Après k questions, il ne reste que $2^0 + 1$ solutions possibles
  - Il faut alors $k + 1 = \lg(n-1) + 1$ questions, en pire cas, pour trouver la solution (pour $n > 1$).
  - Pour $n - 1 = 2^k$, on a $\lg(n-1) + 1 = \lfloor \lg(n-1) \rfloor + 1 = \lceil \lg(n) \rceil$ (voir chap1 ou annexe A du manuel)
- Nous devons donc poser $\lceil \lg(n) \rceil$ questions en pire cas pour obtenir la solution



Minorants informationnels (suite)

- Considérons maintenant le problème de trier un tableau de n éléments en effectuant uniquement des comparaisons

- Pour trier un tableau d'éléments distincts, nous devons effectuer une seule permutation parmi les n! permutations possibles

- Lorsque nous comparons deux éléments entre eux, cela élimine la moitié des permutations possibles.
  - En effet : si x < y alors cela élimine toutes les permutations où x succède à y

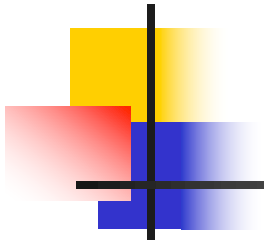
- Similaire à l'exemple précédent : chaque comparaison divise par deux le nombre de solutions possibles. Mais, cette fois-ci, le nombre de solutions possibles est n!
  - Il faut alors $\lceil \log(n!) \rceil \in \Theta(n \log n)$ comparaisons en pire cas

Minorants informationnels (suite)

- **Tout algorithme de tri par comparaisons nécessite donc, en pire cas, $\Omega(n \log n)$ comparaisons.**
  - Les algorithmes de tri en $O(n \log n)$ sont donc asymptotiquement optimaux.
  - Le minorant en $\Omega(n \log n)$ est donc également optimal.

- De tels succès sont (présentement) rares.
  - Pour la majorité des problèmes, le meilleur minorant croît vraiment beaucoup plus lentement que le temps d'exécution, en pire cas, du meilleur algorithme.
  - En fait, pour plusieurs problèmes, le meilleur algorithme dont nous disposons possède un temps d'exécution à croissance exponentielle en pire cas.

- Examinons alors la complexité (de résolution) des problèmes



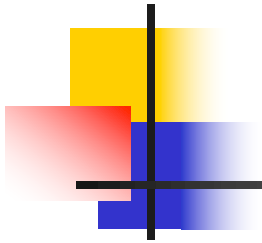
Algorithmes à temps polynomial

- Nous disons qu'un algorithme résout un problème **en temps polynomial** lorsque son temps d'exécution, en pire cas pour toute instance de taille n, est $\in O(p(n))$ où p(n) désigne une fonction polynomiale de n.
  - Note: $t(n) \in O(p(n))$ ssi il existe un entier fini k, indépendant de n, tel que $t(n) \in O(n^k)$
  - Note2: $\log(n) \in O(n)$. Les algorithmes dont le temps $\in O(\log n)$ sont des algorithmes à temps polynomial. Même remarque pour les algorithmes dont le temps d'exécution $\in O(n \log n)$ …

- Les problèmes solubles par un algorithme à temps polynomial sont dits **traitables** (faciles).
  - C'est la majorité des problèmes que nous avons traité jusqu'ici: trier un tableau, trouver le k-ième plus petit élément, trouver l'arbre de recouvrement minimal, trouver pgcd(n,m) …
- Les problèmes non solubles par un algorithme à temps polynomial sont dits **intraitables** (difficiles).

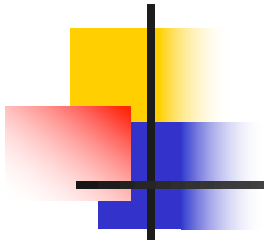
Algorithmes à temps polynomial et « traitabilité »

- Pourquoi disons-nous qu'un problème est traitable ssi il est soluble par un algorithme à temps polynomial?
  - Nous observons que lorsqu'il existe un algorithme à temps polynomial, le degré du polynôme est souvent petit (ex: $k \leq 3$) et qu'il donne souvent naissance à un algorithme que nous pouvons utiliser en pratique.
  - Lorsqu'il existe uniquement un algorithme supra polynomial (ex: exponentiel) pour résoudre un problème, cet algorithme ne peut pas traiter, en un temps raisonnable, les instances de grande taille.
  - Les polynômes possèdent de bonnes propriétés de fermeture:
    - Fermeture sous la composition: si p(n) et q(n) sont deux polynômes, alors p(q(n)) est également un polynôme.
      - Un algorithme qui utilise un nombre polynomial de fois un autre algorithme à temps polynomial, sur des instances de tailles polynomiales, sera également un algorithme à temps polynomial



Problèmes de décision

- Un problème peut être intraitable simplement parce que la solution qu'il doit produire est de trop grande taille.
  - Ex: Pour le problème de trouver toutes les permutations de n entiers distincts, l'algorithme doit, au minimum, produire à sa sortie toutes les n! permutations possibles. Alors tout algorithme pour ce problème doit s'exécuter en temps $\Omega(n!)$.
- Pour éliminer ces cas (triviaux) d' « intraitabilité », considérons d'abord les **problèmes de décision**: les problèmes dont la solution est oui ou non.
  - Exemple: Ayant un graphe connexe G = ‹V, E› et un nombre k, existe-t-il un arbre de recouvrement dont le poids est au plus k?
    - L'algorithme solutionnant ce problème doit uniquement répondre oui ou non (pour n'importe quelle instance).
- Beaucoup de problèmes d'optimisation (comme celui de trouver l'arbre de recouvrement minimal d'un graphe connexe) possèdent un problème de décision qui lui est très relié.



La classe P

- **La classe P est l'ensemble des problèmes de décision qui sont solubles par un algorithme à temps polynomial**.
  - Exemple: Le problème de décision précédent est dans P.
    - En effet, pour le solutionner, il suffit d'exécuter un algorithme à temps polynomial (comme celui de Kruskal) pour trouver un arbre de recouvrement minimal du graphe G. Soit w la somme des poids de cet arbre. Si $w \leq k$, on retourne « oui ».
      Autrement, on retourne « non ».

- Est-ce que tous les problèmes de décision sont dans P?
- Réponse: non!

- En fait, il existe des problèmes de décision pour lesquels il n'existe **aucun** algorithme qui puisse les résoudre! (peu importe leur temps d'exécution)

Certificats succincts et systèmes de preuves

- Une solution proposée pour une instance x « oui » est en fait un **certificat**, i.e., une preuve que x est une instance « oui ».
- Le certificat est **succinct** lorsque sa taille est polynomiale en la taille |x| de l'instance x.
- **Notation**: un **problème de décision X** est un ensemble d'instances « oui ». Nous écrivons alors $x \in X$ lorsque x est une instance « oui » et $x \notin X$ lorsque x est une instance « non ».
- Pour tout $x \in X$, il existe forcément un certificat q qui prouve que $x \in X$.
- Pour tout $x \notin X$, il n'existe pas de q prouvant que $x \in X$.
- Soit Q l'**ensemble des certificats** possibles pour un problème X
- Un **système de preuves** pour X est un ensemble $F \subseteq X \times Q$ de paires (x,q) tel que nous avons:
  - $\forall x \in X, \exists q \in Q : (x,q) \in F$
- i.e., $(x,q) \in F$ ssi x est une instance « oui » et q est une preuve de ce fait
- Notez que $(x,q) \notin F$ lorsque $x \notin X$ (peu importe q) car $F \subseteq X \times Q$

Algorithmes de vérification

- Exemple: si X est l'ensemble des graphes hamiltoniens, le système de preuve F pour X sera alors simplement l'ensemble des paires (g, σ) de graphe $g \in X$ et de séquences de nœuds $\sigma \in Q$ tels que σ est un cycle hamiltonien pour g.

- En plus d'un système de preuve F pour X, il faut pouvoir vérifier q'une paire $(x,q) \in F$. Il faut donc un algorithme de vérification.

- Un **algorithme de vérification** pour un système de preuves $F \subseteq X \times Q$ d'un problème de décision X est un algorithme A tel que:
  - $A(x,q) = 1$ lorsque $(x,q) \in F$
  - Lorsque $(x,q) \notin F$, A peut retourner 0 ou exécuter indéfiniment.

- Ainsi A doit **accepter** F. Il n'est pas nécessaire pour A de décider F.

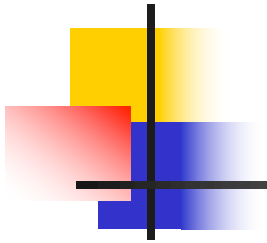
- Un système de preuves muni d'un algorithme de vérification est appelé un **système de preuves vérifiables**.

Algorithmes de vérification à temps polynomial

- **Un algorithme de vérification A(x,q) est à temps polynomial** ssi son temps d'exécution est $\in O((|x|+|q|)^k)$ pour une constante k.

- Si A(x,q) termine toujours au bout d'un temps polynomial, il doit alors forcément retourner 0 au bout de ce temps lorsque $(x,q) \notin F$.
- Un algorithme de vérification à temps polynomial doit donc **décider** F.

- Ainsi, un algorithme de vérification à temps polynomial pour un système de preuves $F \subseteq X \times Q$ d'un problème de décision X est un algorithme A tel que:
  - A(x,q) termine en un temps $\in O((|x|+|q|)^k)$ pour une constante k.
  - A(x,q) = 1 si $(x,q) \in F$
  - A(x,q) = 0 si $(x,q) \notin F$



La classe NP

- La classe NP est l'ensemble des problèmes de décision qui admettent un **système de preuves succinctes vérifiables en temps polynomial**.
- **Définition de la classe NP**. Un problème de décision X est dans NP si et seulement si il existe un système de preuves $F \subseteq X \times Q$ et un algorithme de vérification A tels que:
  - $\forall\, x \in X, \exists\, q \in Q : (x,q) \in F$ et $|q| \in O(|x|^k)$ pour une constante k
    - (i.e., le certificat q doit être succinct)
  - $\forall$ (x,q): A(x,q) termine en temps polynomial et:
    - A(x,q) = 1 si $(x,q) \in F$
    - A(x,q) = 0 si $(x,q) \notin F$
- NP signifie « Nondeterministic Polynomial-time » car si, pour une instance $x \in X$, nous obtenons, de manière « non déterministe », une preuve q (que $x \in X$), nous pouvons vérifier ce fait en temps polynomial (à l'aide d'un algorithme de vérification).
- Cette définition de la classe NP est légèrement plus précise que celle présentée dans votre manuel.

P ⊆ NP

- **Théorème: P ⊆ NP**.
- **Preuve:** Considérons un problème de décision X soluble par un algorithme L à temps polynomial.
  - L'idée: dans ce cas L(x) décide en temps polynomial si, oui ou non, $x \in X$. Nous avons donc pas besoins de preuve que $x \in X$ car L(x) nous indiquera si $x \in X$ au bout d'un temps polynomial.
  - Nous pouvons donc utiliser un ensemble trivial Q = {0} de certificats et définir un système de preuves $F = \{(x,0) : x \in X\}$.
  - Donc, pour tout $x \in X$, il existe un certificat succinct q = 0 tel que $(x,q) \in F$. Et pour tout $x \notin X$, il n'existe pas de certificat q tel que $(x,q) \in F$.
  - L'algorithme de vérification A(x,q) ignore q et exécute L(x) qui terminera en temps polynomial. Après la terminaison de L(x), A(x,q) retourne 1 si L(x) = « oui » et retourne 0 si L(x) = « non ».
  - Tout problème X dans P possède donc un système de preuves vérifiables en temps polynomial. Donc $X \in P \Rightarrow X \in NP$. **CQFD**.

Exemples de problèmes dans NP

- Les problèmes suivants sont tous dans NP.
  - Cycle hamiltonien.
  - Commis voyageur décision.
  - Sac à dos décision
- Pour prouver cela il suffit de démonter qu'ils possèdent tous un système de preuves succinctes vérifiables en temps polynomial.
- **Cycle hamiltonien**: Existe-t-il un cycle parcourant exactement une fois chaque nœud du graphe g?
  - le système de preuves est l'ensemble de paires (g, σ) où g est un graphe hamiltonien et σ est une liste de nœuds. L'algorithme de vérification A(g,σ) n'a qu'à parcourir σ et vérifier:
    - Si chaque paire de nœuds consécutifs dans cette liste est une arête appartenant à g.
    - Si cette liste comprend n+1 nœuds avec n = nombre de nœuds de g.
    - Si le premier nœud de cette liste est le même que le dernier nœud.
    - Si les n premiers nœuds sont distincts.
  - Cette vérification nécessite uniquement un temps polynomial en |g|+|σ|.

Exemples de problèmes dans NP (suite)

- **Commis voyageur décision**: Soit n villes (et la matrice des n(n-1)/2 distances) et un entier d. Existe-t-il un circuit de longueur au plus d qui parcourt les n villes?
  - Le système de preuves est l'ensemble des triplets (M,d,σ) où M est la matrice symétrique des distances de dimensions n x n, d est la longueur maximale du parcours désiré. Le paramètre σ est une liste de nœuds. L'algorithme de vérification doit vérifier que:
    - σ contient n+1 nœuds
    - Les n premiers nœuds de σ sont distincts
    - Le dernier nœud de σ est le même que le premier nœud
    - La longueur du circuit (obtenu en sommant la distance entre chaque paire de nœuds consécutifs dans σ) est $\leq$ d.
  - Cette vérification nécessite uniquement un temps polynomial en n + $|\sigma|$.

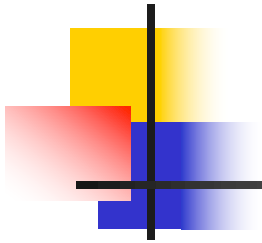
Exemples de problèmes dans NP (suite)

- **Sac à dos décision**. Soit n objets avec poids w_1, ..., w_n, et valeurs v_1, …, v_n, et un sac à dos de capacité W et un nombre V. Existe-t-il un sous-ensemble de ces objets tel que le poids total est au plus W et tel que la valeur totale est au moins V ?
  - Le système de preuves est l'ensemble de pairs (x,S) où x est constitué des vecteurs (w_1, ..., w_n) et (v_1, …, v_n) et des deux nombres W et V. Pour tout $x \in X$, S est un sous-ensemble de {1,…,n}. L'algorithme de vérification A(x,S) retourne 1 ssi:
    - $\sum_{i \in s} w_i \leq W$ et $\sum_{i \in S} v_i \geq V$
  - Ce qui s'effectue en un temps polynomial en |x| et |S|.

- Ces 3 problèmes sont donc dans NP mais nous ne savons pas s'ils sont dans P car, en dépit de nombreuses années d'effort, nous n'avons toujours pas trouvé d'algorithmes à temps polynomiaux pour les résoudre et nous n'avons pas encore pu démontrer que ces problèmes ne peuvent pas être résolus en temps polynomial.

Polynomialement réductible (par association)

- Pour comparer le degré de difficulté entre deux problèmes, il est souvent utile de procéder par réduction. Ainsi, si résoudre un problème résout également le deuxième, on conclut que le premier problème n'est pas plus facile que le deuxième.
- Un problème de décision X_1 est **polynomialement réductible (PR)** à un problème de décision X_2 ssi il existe une **fonction de réduction f calculable en temps polynomial** et qui a la propriété de satisfaire:
  - **Si $x \in X_1$ alors $f(x) \in X_2$**
  - **Si $f(x) \in X_2$ alors $x \in X_1$**
- L'existence d'une telle fonction de réduction f implique que si X_2 est résoluble en temps polynomial (par un algorithme A_2), alors X_1 est également résoluble en temps polynomial.
  - En effet, pour déterminer si $x \in X_1$ on exécute f(x) en un temps polynomial en |x| et, ensuite, on détermine si $f(x) \in X_2$ à l'aide de A_2 en un temps polynomial en |f(x)|. On a alors déterminé si $x \in X_1$ (en un temps polynomial en |x|) car $\mathbf{x \in X_1 \Leftrightarrow f(x) \in X_2}$.



PR

- **Théorème: cycle hamiltonien est PR à commis voyageur décision.**
- **Preuve:**
  - Soit g un graphe pour le problème cycle hamiltonien.
  - Assignons une distance 1 à toutes les arêtes de g
  - À chaque paire de noeuds non reliés par une arête de g, ajoutons une arête de distance 2. Le graphe résultant f(g) est maintenant totalement connecté (avec des arêtes de distances 1 ou 2).
  - f(g) est donc une instance du problème du commis voyageur.
  - Mais pour obtenir une instance de commis voyageur **décision,** il faut, en plus, se choisir un entier d. Choisissons d = n (où n est le nombre de nœuds du graphe g)
  - La paire (f(g), n) constitue alors une instance pour commis voyageur décision. Cette instance est obtenue en un temps polynomial en |g|.

Exemple de réduction entre problèmes de décision

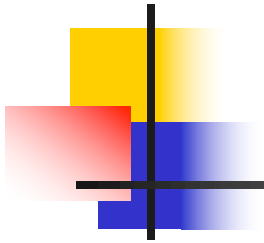
- … Preuve (suite) …
  - Si g est une instance « oui » de cycle hamiltonien, alors f(g) possède un circuit dont la somme des distances = n. Alors (f(g),n) est une instance « oui » de commis voyageur décision.
  - Si g est une instance « non » de cycle hamiltonien, alors f(g) ne possède pas de circuit de longueur n (sinon il serait un cycle hamiltonien pour g). Alors tous les circuits de f(g) ont une longueur supérieure à n. Donc (f(g),n) est une instance « non » de commis voyageur.
  - Alors g est une instance « oui » de cycle hamiltonien si et seulement si (f(g),n) est une instance « oui » de commis voyageur décision.
  - Puisque chaque instance (f(g),n) est obtenue en un temps polynomial en |g|, cycle hamiltonien est PR à commis voyageur décision. **CQFD**

Problèmes NP-complets

- **Un problème de décision X est NP-complet si est seulement si:**
  - **X est dans NP et**
  - **Tout problème dans NP est PR à X**
- Il n'est pas évident de prouver que tout problème dans NP (connu et inconnu) se réduit à un problème X. C'est pourquoi Stephen Cook obtint le prix Turing après avoir découvert en 1971 le premier problème NP-Complet qu'il appela CNF-SAT (pour Conjonctive Normal Form Satisfiability).
- Une instance CNF-SAT est une conjonction de disjonctions de variables ou de leurs négations. Ex:

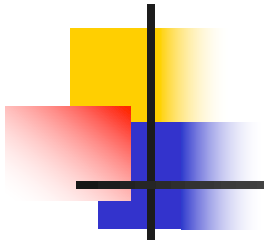
$$(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3 \vee x_4)$$

- Le problème de décision associé à une instance SAT est: existe-t-il une affectation des variables à des valeurs booléennes rendant cette clause vraie?
- Dans cet exemple, la réponse est oui. Le certificat est x_1=faux, x_2=faux, x_3=vrai, x_4=vrai.
- La preuve que tous les problèmes de NP sont réductibles en temps polynomial à CNF-SAT est un chef-d'œuvre (et dépasse largement les objectifs du cours).
- Tout problème dans NP peut donc être exprimé sous forme d'une formule SAT.



Problèmes NP-complets

- Pour prouver qu'un problème X est NP-complet, il « suffit » de prendre un problème NP-complet connu Y et de le réduire au problème X.
- Ainsi, tout problème dans NP se réduit à Y et, par transitivité, se réduit à X.
- En procédant de cette façon, plus d'une centaine de problèmes NP-complets ont été identifiés dont cycle hamiltonien, commis voyageur décision, sac à dos décision, k-colorabilité, programmation linéaire entière …
- Si vous n'arrivez pas, en dépit de nombreux efforts, à trouver un algorithme à temps polynomial pour un problème de NP, il y a de fortes chances qu'il existe un problème NP-complet qui soit réductible en temps polynomial à votre problème.

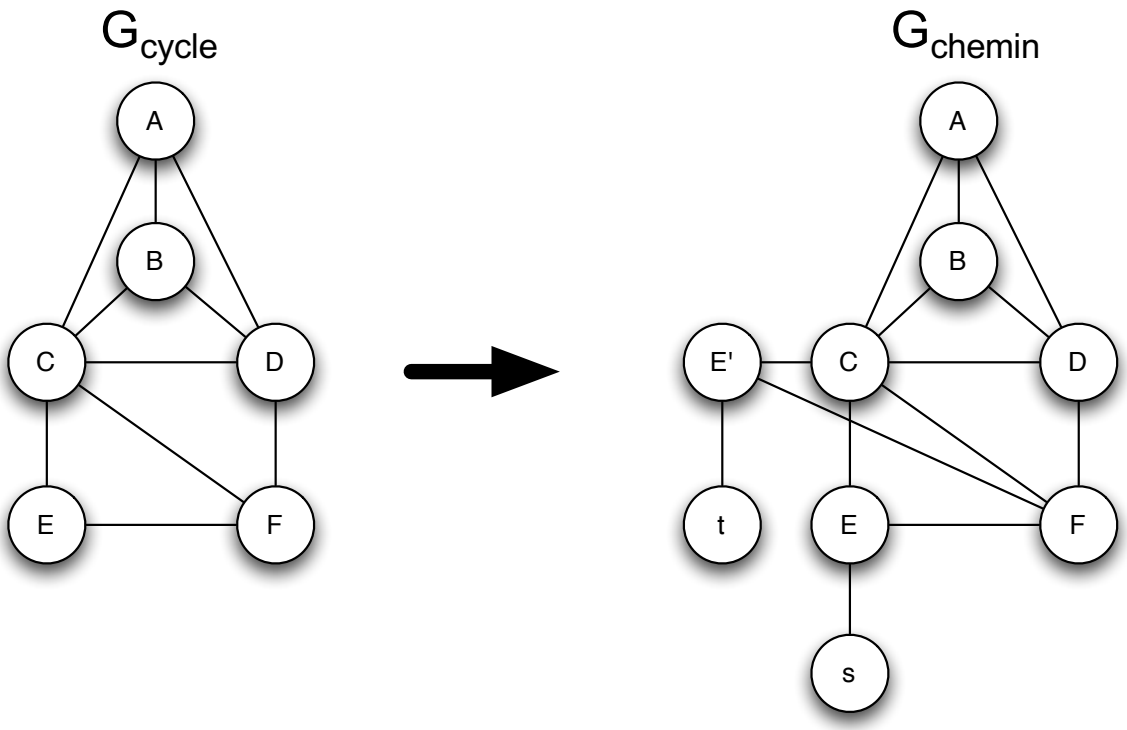


Preuve qu'un problème est NP-complet

- Nous savons que le problème de déterminer l'existence d'un cycle hamiltonien dans un graphe est NP-complet.
- Nous voulons démontrer que vérifier si un graphe contient un chemin hamiltonien est aussi NP-complet.
  - Un chemin hamiltonien est un chemin qui visite exactement une fois tous les nœuds du graphe.
- L'instance des deux problèmes est donnée par un graphe non orienté.
- Il faut donc modifier, en temps polynomial, l'instance d'un cycle hamiltonien G_{cycle} en une instance de chemin hamiltonien G_{chemin} de sorte qu'il existe un cycle hamiltonien dans G_{cycle} si et seulement s'il existe un chemin hamiltonien dans G_{chemin}.

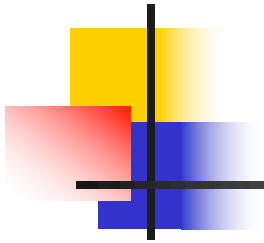
Réduction de Cycle Hamiltonien à Chemin Hamiltonien

- On procède ainsi pour transformer l'instance
  - On prend un nœud quelconque u dans le graphe G_{cycle}.
  - On crée une copie u' du nœud u.
  - Pour chaque arête (u, v), on crée une arête (u', v).
  - On crée ensuite des nœuds s et t et deux arêtes (s, u) et (t, u').



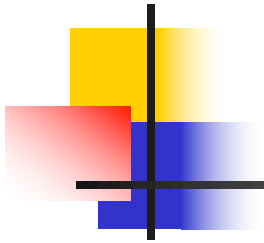
Réduction de Cycle Hamiltonien à Chemin Hamiltonien

- **Théorème**: Il existe un cycle hamiltonien dans G_{cycle} si et seulement s'il existe un chemin hamiltonien dans G_{chemin}.
- **Démonstration**: (=>) Soit u_1 le nœud dans G_{cycle} qui a été dupliqué pour construire G_{chemin}. Soit $u_1, u_2, u_3, \ldots, u_n, u_1$ la séquence de nœud formant un cycle hamiltonien dans G_{cycle}. Alors le chemin s, $u_1, u_2, u_3, \ldots, u_n, u'_1$, t est un chemin hamiltonien dans G_{chemin}.
- (<=) Soit C un chemin hamiltonien dans G_{chemin}. Les nœuds s et t sont les deux extrémités du chemin puisque ces nœuds n'ont qu'un seul voisin. Ils ne peuvent donc pas se retrouver au milieu du chemin.
- Sans perte de généralité, disons que le chemin quitte s pour aller vers t. Alors le deuxième nœud du chemin est u_1 et l'avant-dernier nœud du chemin est u'_1. Le chemin C a donc la forme s, $u_1, u_2, u_3, \ldots, u_n, u'_1$, t.
- La séquence $u_1, u_2, u_3, \ldots$ un u_1 est un cycle dans G_{cycle} puisque l'arête (u_i, u_{i+1}) est une arête à la fois dans G_{cycle} et dans G_{chemin}. **CQFD**



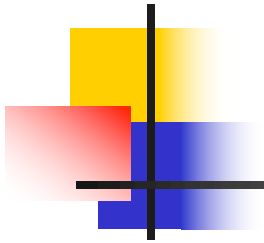
P ≠ NP ou P = NP ?

- Le fait de démontrer qu'un problème A est NP-complet donne un fort indice de son intraitabilité, car un algorithme à temps polynomial pour résoudre A pourrait résoudre, en temps polynomial, tous les problèmes de NP.
- En raison de l'existence des problèmes NP-complets, une majorité d'informaticiens **croient** que P ≠ NP.



Réductions et similarités

- Le fait qu'il existe un problème NP-complet qui soit similaire à votre problème n'assure pas la NP-complétude de votre problème.
- En effet considérez le problème de déterminer si oui ou non un graphe possède un cylcle eulérien.
  - Un cycle eulérien est un cycle qui parcourt toutes les arêtes d'un graphe une seule fois.
- En dépit de la similarité entre cycle eulérien et cycle hamiltonien, cycle eulérien est résoluble en temps polynomial, alors que cycle hamiltonien est NP-complet!
  - En effet, selon un des théorèmes d'Euler, un graphe possède un cycle eulérien ssi chaque nœud est de degré pair (le degré d'un nœud est le nombre d'arêtes que possèdent ce nœud)
  - Il suffit donc de visiter chaque nœud une seule fois pour déterminer si ce graphe possède un cycle eulérien



Que faire lorsque c'est NP-complet?

- De nombreux problèmes d'intérêt pratique sont NP-complets et sont trop importants pour les ignorer.

- Le prochain chapitre traite des approches algorithmiques nous permettant de trouver des solutions non optimales, mais qui peuvent parfois être très satisfaisantes.

En résumé (définitions)

- **Définition 1**: Un problème de décision est un problème dont la solution est « oui » ou « non ».
- **Définition 2**: Un certificat est une donnée qui prouve que la réponse à l'instance du problème est « oui ».
- **Définition 3**: Un algorithme de vérification est un algorithme qui prend une instance *x* et une donnée *c* et qui retourne « oui » si *c* est bien un certificat pour *x*. Il est à temps polynomial s'il s'exécute en temps $O((|x| + |c|)^k)$ pour un entier k.
- **Définition 4**: La classe P est l'ensemble des problèmes de décision solubles en temps polynomial.
- **Définition 5**: La classe NP est l'ensemble des problèmes de décision qui admettent un algorithme de vérification à temps polynomial.

En résumé (suite)

- **Définition 6**: Le problème de décision X_1 est polynomialement réductible (PR) à un problème de décision X_2 si et seulement si il existe une fonction *f* calculable en temps polyomial et qui a la propriété de satisfaire:

  x est une instance « oui » de $X_1 \Leftrightarrow$ f(x) est une instance « oui » de X_2.

- **Définition 7**: Y est un problème NP-Complet si et seulement si tout problème $X \in NP$ est polynomialement réductible à Y.