# Complexité : solveur et approximation

M1 Informatique Luminy 2024-25


| | |
|---|---|
| Kévin Perrot | <kevin.perrot@univ-amu.fr> |
| Pablo Concha-Vega | <pablo.concha-vega@lis-lab.fr> |
| Antonio E. Porreca | <antonio.porreca@univ-amu.fr> |
| Mathieu Roget | <mathieu.roget@univ-amu.fr> |


9hCM 9hTD 9hTP

NF=MAX(ET ; 0.3*CC+0.7*ET)

# Dans la vie quotidienne...

Il existe des problèmes intéressants (que l'on aimerait bien résoudre) qui sont NP-complets (donc on ne peut pas a priori pas les résoudre).

## Alors, que faire ?

1. Si les instances sont petites, les résoudre en temps exponentiel.
2. Un cas particulier pourrait être résolu en temps polynomial.
3. Utiliser un solveur SAT ou CSP.
4. Utiliser un algorithme d'approximation.

# Montrer que mon problème $X$ est NP-complet

- Montrer que $X \in \text{NP}$ : algo poly pour vérifier une solution.
- Choisir un problème $A$ déjà connu pour être NP-complet, et le réduire à $X$ (montrer $A \leq_T^P X$).
  C'est-à-dire, expliquer comment résoudre efficacement $A$ en utilisant un hypothétique algorithme efficace pour $X$.
  $\Longrightarrow$ une seule réduction

Choisir un problème qui ressemble un peu à $X$ :

**3-SAT** : clauses de taille 3
**1in3-SAT** : exactement 1 littéral vrai par clause
**NAE-SAT** : pas tous les littéraux égaux
$\hookrightarrow$ théorème de dichotomie de Schaefer

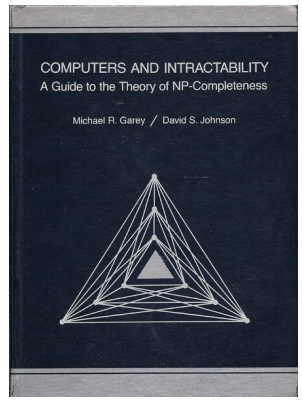
**Clique Stable Transversal Cycle-hamiltonien TSP**

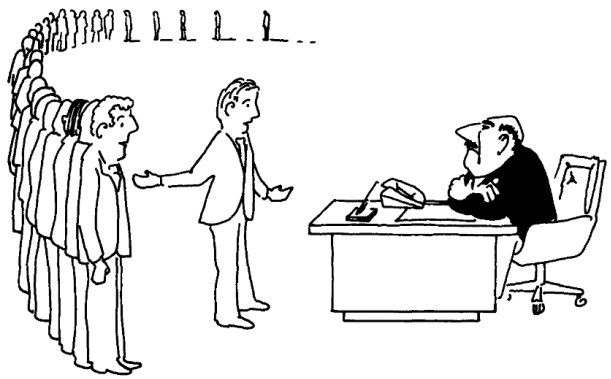
**3-Coloration** des graphes planaires 4-réguliers

**3D-Matching** : instance $M \subseteq X \times Y \times Z$ and $k$

**Subset-sum**, **Paritition**

**...**



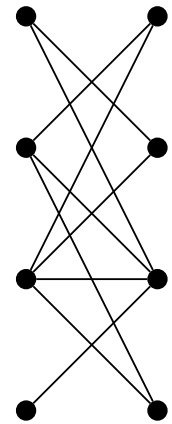



"I can't find an efficient algorithm, but neither can all these famous people."

# 2. Cas particuliers de problèmes NP-complets

**2-SAT**, **Horn-SAT** $\in$ P

Graphe biparti : qui ne contient pas de cycle impair (testable en temps linéaire).



**Clique** des graphes bipartis $\in$ P

$\forall k$ : **k-Coloration** des graphes bipartis $\in$ P

$\hookrightarrow$ car clique maximum $\underset{\text{graphe parfait}}{=}$ nombre chromatique $\leq 2$

**Transversal** (**Vertex-cover**) des graphes bipartis $\in$ P

$\hookrightarrow$ car transversal minimum $\underset{\text{théorème König}}{=}$ matching maximum (2D)

⚠ **Cycle-hamiltonien** des graphes bipartis reste NP-complet

Presque tous les problèmes sont plus faciles sur les arbres !

# 3. Solveur SAT

Quand on a besoin d'une solution... il faut bien résoudre un problème NP-complet !

⇓ ↪ donc en temps exponentiel dans le pire cas...

**Solveurs SAT** : puisque SAT est NP-complet, tout NP se réduit à lui !

Idée : on se concentre sur SAT, parce que les réductions vers SAT sont ≪ plutôt faciles ≫.

## Principe :

1. Pour chaque instance de $X$, construire une formule CNF équivalente ($\leq_T^P$). 🤔
2. Appeler le solveur pour trouver une solution (format DIMACS). 😴
3. Retraduire la solution pour l'instance de $X$. 😎

Techniques de résolution :

- DPLL : unit + pure + backtrack + heuristique de branchement (*eg.* Max Occur in Min Sized clauses)
- CDCL (conflict-driven clause learning) : DPLL + backjumping intelligent (éviter les recherches redondantes)
- Parallel SAT-solving : portfolio, diviser-pour-régnier, recherche locale

Minisat

```
apt intsall minisat
```


PySAT

```
pip install python-sat
```


glucose

```
make (C++)
```


Choco

```
javac -cp choco.jar ...
```

Solutions fermées payantes : Gurobi, Hexaly, CPLEX, ...

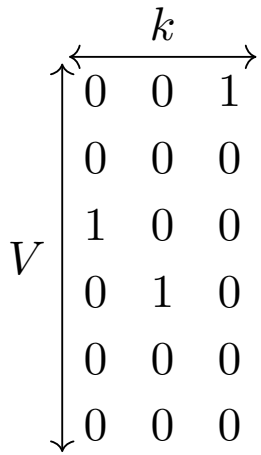
Attention à la consommation énergétique !



# 3. Modélisation SAT (= réduction vers SAT)

Par définition de NP-difficulté : $\forall X \in \mathrm{NP} : X \leq_T^{\mathrm{P}} \mathrm{SAT}$

1. Quelles variables ?
2. Quelle conjonction de contraintes (clauses) ?
3. Exprimer en CNF (il suffit de développer).




Exemple : Clique : instance $G = (V, E)$ et $k \in \mathbb{N}$.

- une variable $x_v$ pour chaque sommet $v \in V$
  $x_v = 1$ : sommet $v$ dans la clique
  $x_v = 0$ : sommet $v$ hors de la clique
- pour chaque arête $\{u, v\} \in E$ :
  pas les deux : $\neg(x_u \wedge x_v)$
  non l'un ou non l'autre : $\neg x_u \vee \neg x_v$
  si l'un alors pas l'autre : $x_u \Rightarrow \neg x_v$
  $\Big\} \equiv$
- comment imposer la taille $\geq k$ ?

- $k$ variables $x_v^1, \ldots, x_v^k$ pour chaque sommet $v \in V$
  $\bigvee_i x_v^i = 1$ : sommet $v$ dans la clique
  au moins une à vrai pour chaque $i$ : $\bigvee_{v \in V} x_v^i$
  au plus une à vrai par sommet : $\bigwedge_{i \neq j} (\neg x_v^i \vee \neg x_v^j)$
- pour chaque arête $\{u, v\} \in E$ : $\bigwedge_i \bigwedge_j (\neg x_u^i \vee \neg x_v^j)$

$$\varphi_{G,k} = \left[\bigwedge_i \bigvee_{v \in V} x_v^i\right] \wedge \left[\bigwedge_{v \in V} \bigwedge_i \bigwedge_{j \neq i} (\neg x_v^i \vee \neg x_v^j)\right] \wedge \left[\bigwedge_{\{u,v\} \in E} \bigwedge_i \bigwedge_j (\neg x_u^i \vee \neg x_v^j)\right]$$

En FNC ! nombre de variables : $n\,k$ nombre de clauses : $k + n\,k(k-1) + m\,k^2 \implies$ poly

---

Modélisation CSP : variables sur des domaines variés, avec contraintes (relations).
(constraint satisfaction problem) $\rightsquigarrow$ démo Choco !

---

Attention à la sensibilité au codage (à votre réduction).

# 4. Algorithmes d'approximation

Pour certaines applications de problèmes d'optimisation,
on pourrait se satisfaire d'une solution proche de l'optimale.

Pour une instance $w \in \Sigma^*$, soit $Q^*(w)$ la qualité d'une solution optimale,
et $Q_A(w)$ la qualité de la solution produite par l'algorithme $A$.

On dit que $A$ possède un **facteur d'approximation** $\rho(n)$ lorsque :

$$\forall n : \quad \max_{w \in \Sigma^n} \left\{ \frac{Q_A(w)}{Q^*(w)} , \frac{Q^*(w)}{Q_A(w)} \right\} \leq \rho(n)$$

Remarques. $\rho(n) \geq 1$ et $\rho(n) = 1 \iff$ optimal.

Exemple : Algo de 2-approximation pour **Transversal** (**Vertex-cover**)

```
Entrée: G
C = ensemble vide
E = arêtes de G
tant que E != vide faire:
    prendre (u,v) dans E
    ajouter u et v à C
    retirer de E toutes les arêtes
            incidentes à u ou à v
retourner C
```

Temps : $\mathcal{O}(|V| + |E|)$ pour listes d'adjacence.

Facteur d'approximation 2 :

À la fin `C` est bien un transversal,
car on boucle pour couvrir toutes les arêtes.

L'ensemble $E'$ des arêtes prises dans `E`
doivent être couverte, et elles sont disjointes,
donc $Q^*(G) \geq |E'|$.
On ajoute à `C` les deux extrémités des arêtes de $E'$,
donc $Q_A(G) = 2|E'|$.

# 4. Algorithmes d'approximation

Deux paramètres à prendre en compte :

- temps d'exécution, et
- facteur d'approximation.

Compromis :
Un schéma d'approximation en temps polynomial (PTAS) est
un algorithme prenant en entrée une instance $w$ et un réel $\epsilon > 0$,
et qui produit en temps polynomial en $n$ (peu importe la dépendance en $\epsilon$, *eg.* $\mathcal{O}(n^{1/\epsilon})$)
une solution optimale à un facteur $1 + \epsilon$ près.

# 4. Algorithmes probabilistes

Exemple : **Max-3-SAT**
Algo : choisir 0 ou 1 avec proba $\frac{1}{2}$ pour chaque variable (indépendamment).
$\Longrightarrow$ Espérance de satisfaire $\frac{7}{8}$ des clauses (87.5%).

Chaque littéral peut satisfaire sa clause avec proba $\frac{1}{2}$,
donc la proba qu'une 3-clause $C_i$ soit satisfaite est $E(C_i) = 1 - (\frac{1}{2})^3 = \frac{7}{8}$.
Proba indépendante pour chaque clause : $E(C_1 + \ldots + C_m) = \sum_{i=1}^{m} E(C_i) = m\frac{7}{8}$.

C'est une $\frac{8}{7}$-approximation probabiliste (au maximum on statisfait $m$ clauses).

# Take-home message

Si savoir résoudre efficacement mon problème A
peut permettre de résoudre efficacement SAT
ou n'importe quel autre problème NP-complet, alors :

1. soit instances très petites
2. soit cas-particulier
3. soit solveur SAT ou CSP
4. soit approximation
5. soit 

# Bonus

**Théorème.** La fonction $f(k) = \begin{cases} 1 & \text{si } \mathbf{SAT} \in \mathrm{DTIME}(n^k) \\ 0 & \text{sinon} \end{cases}$
est calculable en temps polynomial.

$\hookrightarrow$ Car c'est une fonction à seuil (si P = NP) ou constante (si P $\neq$ NP).

$\Longrightarrow$ Mais on ne connaît pas d'algorithme pour la calculer !

# Bonus

**Théorème.** L'algorithme suivant résoud* SAT en temps polynomial si et seulement si P = NP.

* à partir d'une certaine taille de formule

Entrée : $\varphi$ de taille $n$.

- $i \leftarrow 1$ ;
- `pour` $i$ de 1 à $n$ `faire`
  - simuler $M_i(\psi)$ pour toute formule $\psi$ de taille $\leqslant \log n$,
  - si pour tout $\psi$, $M_i$ donne toujours une affectation valide lorsque $\psi \in$ `SAT`, sortir de la boucle ;
- simuler $M_i(\varphi)$, produisant un résultat $a$ (une affectation des variables de $\varphi$) ;
- si $\varphi(a) = 1$ `accepter`, sinon `rejeter`.

[Perifel pages 93–95]

$\Longrightarrow$ Donc on ne sait pas analyser la complexité de ce programme !