P versus NP

(et autres questions liées)

Arnaud Casteigts
Université de Genève

Retraite annuelle de la CRM
Champéry

11 / 09 / 2026.

Disclaimer...

Dana Angluin (1947 -)

Nombreuses contributions en informatique théorique.

Types de preuves :

Proof by example :
The author gives only the case n=2 and suggests that it contains most of the ideas of the general proof.

Proof by intimidation :
'Trivial.'

Proof by vigorous handwaving :
Works well in a classroom or seminar setting.

Proof by omission :
'The reader may easily supply the details.'
'The other 253 cases are analogous.'

Proof by obfuscation :
A long plotless sequence of true and/or meaningless syntactically related statements.

Proof by funding :
How could three different government agencies be wrong ?

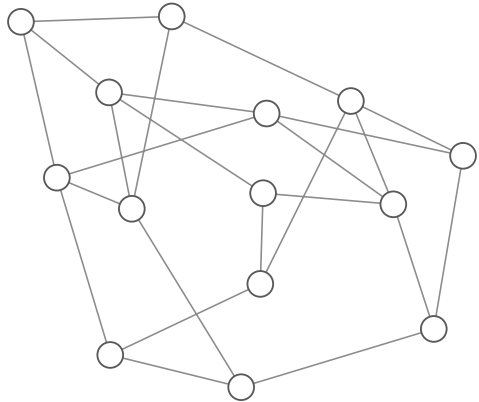
Proof by eminent authority :
'I saw Karp in the elevator and he said it was probably NP-complete.'

. . .

`https://users.cs.northwestern.edu/~riesbeck/proofs.html`

→ *Mon exposé utilisera plusieurs de ces techniques...*

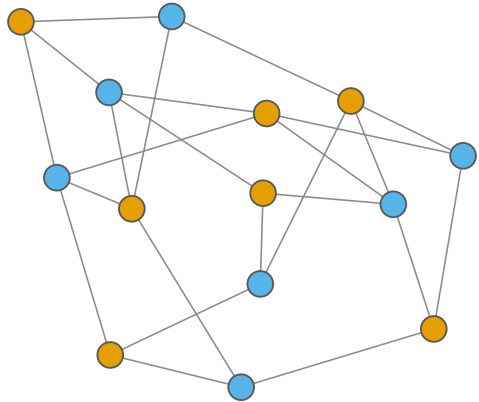
Commençons par un jeu



Objectif : Colorier les sommets de ce graphe de sorte que deux sommets voisins (reliés par une arête) ont toujours une couleur différente.

Est-ce faisable avec **2** couleurs ?

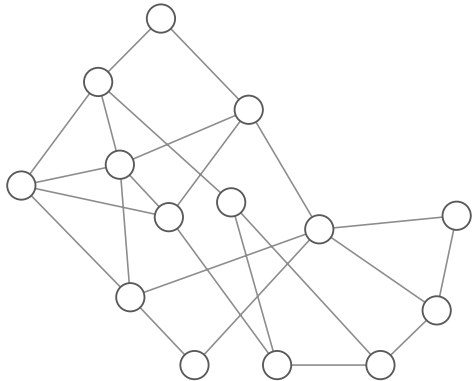
Commençons par un jeu



Objectif : Colorier les sommets de ce graphe de sorte que deux sommets voisins (reliés par une arête) ont toujours une couleur différente.

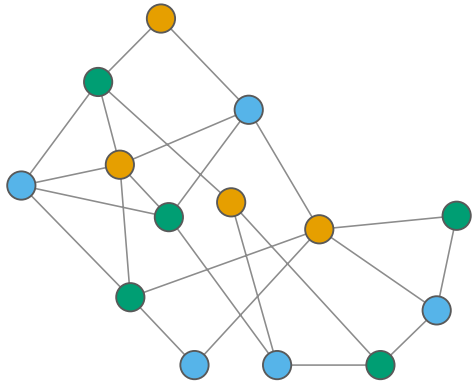
Est-ce faisable avec **2** couleurs ? Oui ! (facile).

Commençons par un jeu



Et celui là avec 3 couleurs ?

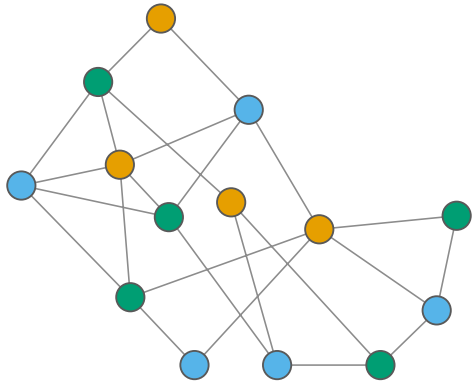
Commençons par un jeu



Et celui là avec 3 couleurs ?

(plus difficile)

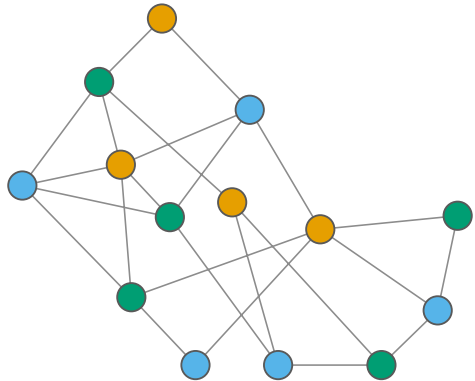
Commençons par un jeu



Et celui là avec 3 couleurs ? (plus difficile)

Qu'en est-il de **vérifier** une solution donnée ?

Commençons par un jeu



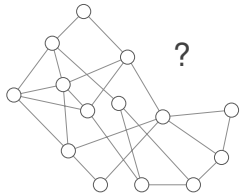
Et celui là avec 3 couleurs ? (plus difficile)

Qu'en est-il de **vérifier** une solution donnée ?

$\rightarrow$ Il n'y a qu'à vérifier que chaque arête a ses extremités de couleur différente. (facile)

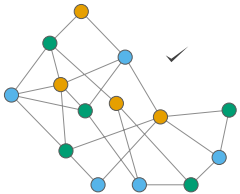
Intuition

Chercher une solution



Lent

Vérifier une solution



Rapide

Ces deux tâches ont-elles des coûts vraiments différents ?

Le problème P vs. NP donne un sens précis à cette question.

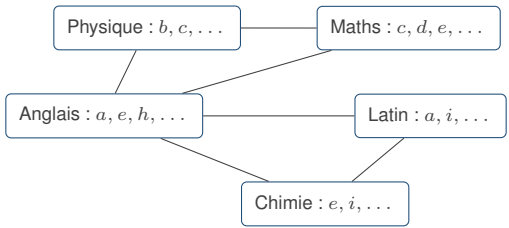
(avec beaucoup d'implications)

Parenthèse : créer un emploi du temps d'examen

Chaque matière est suivie par un ensemble d'élèves $\{a, b, c, \dots\}$.

Deux matières qui **partagent un élève** ne peuvent pas avoir lieu en même temps.

Graphe de conflit :

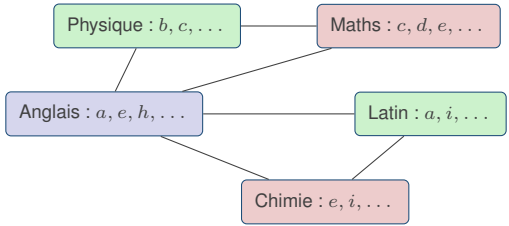


Parenthèse : créer un emploi du temps d'examen

Chaque matière est suivie par un ensemble d'élèves $\{a, b, c, \dots\}$.

Deux matières qui **partagent un élève** ne peuvent pas avoir lieu en même temps.

Graphe de conflit :

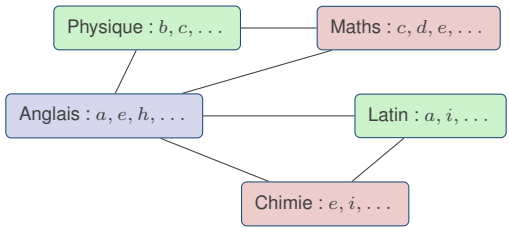


Parenthèse : créer un emploi du temps d'examen

Chaque matière est suivie par un ensemble d'élèves $\{a, b, c, \dots\}$.

Deux matières qui **partagent un élève** ne peuvent pas avoir lieu en même temps.

Graphe de conflit :



k couleurs $\implies$ k créneaux suffisent pour organiser les examens.

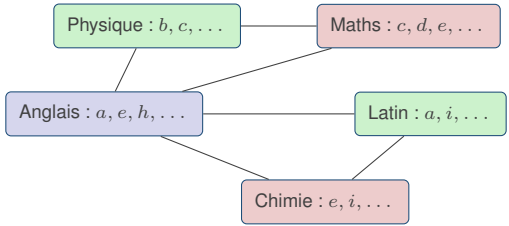
Les matières ayant la même couleur peuvent avoir lieu en même temps.

Parenthèse : créer un emploi du temps d'examen

Chaque matière est suivie par un ensemble d'élèves $\{a, b, c, \dots\}$.

Deux matières qui **partagent un élève** ne peuvent pas avoir lieu en même temps.

Graphe de conflit :



k couleurs $\implies$ k créneaux suffisent pour organiser les examens.

Les matières ayant la même couleur peuvent avoir lieu en même temps.

Aussi utilisé pour :

- Attribuer les altitudes de vol (conflits = croisement de trajectoire)
- Attribuer les fréquences radios (conflits = interférences)
- Placer les invités à un mariage (conflits = mésentente)
- ...

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU		
FACTORISER UN NOMBRE		
ECHECS*		

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU		
FACTORISER UN NOMBRE		
ECHECS*		

			9			1		7
3		5						
	2							
			5					
				3	9			
	1				2	4		6
	4					9		
	7			2	8		3	
5			7					

4	6	8	9	5	3	1	2	7
3	9	5	2	1	7	8	6	4
7	2	1	6	8	4	5	9	3
2	8	7	5	4	6	3	1	9
6	5	4	1	3	9	7	8	2
9	1	3	8	7	2	4	5	6
8	4	2	3	6	5	9	7	1
1	7	9	4	2	8	6	3	5
5	3	6	7	9	1	2	4	8

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	
FACTORISER UN NOMBRE		
ECHECS*		

			9			1		7
3		5						
	2							
			5					
				3	9			
	1				2	4		6
	4					9		
	7			2	8		3	
5			7					

4	6	8	9	5	3	1	2	7
3	9	5	2	1	7	8	6	4
7	2	1	6	8	4	5	9	3
2	8	7	5	4	6	3	1	9
6	5	4	1	3	9	7	8	2
9	1	3	8	7	2	4	5	6
8	4	2	3	6	5	9	7	1
1	7	9	4	2	8	6	3	5
5	3	6	7	9	1	2	4	8

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	Rapide
FACTORISER UN NOMBRE		
ECHECS*		

			9			1		7
3		5						
	2							
			5					
				3	9			
	1				2	4		6
	4					9		
	7			2	8		3	
5			7					

4	6	8	9	5	3	1	2	7
3	9	5	2	1	7	8	6	4
7	2	1	6	8	4	5	9	3
2	8	7	5	4	6	3	1	9
6	5	4	1	3	9	7	8	2
9	1	3	8	7	2	4	5	6
8	4	2	3	6	5	9	7	1
1	7	9	4	2	8	6	3	5
5	3	6	7	9	1	2	4	8

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	Rapide
FACTORISER UN NOMBRE		
ECHECS*		

Exemple : factoriser 6315265397

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	Rapide
FACTORISER UN NOMBRE	Lent	
ECHECS*		

Exemple : factoriser 6315265397 $= 73291 \times 86167$

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	Rapide
FACTORISER UN NOMBRE	Lent	Rapide
ECHECS*		

Exemple : factoriser 6315265397 $= 73291 \times 86167$

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	Rapide
FACTORISER UN NOMBRE	Lent	Rapide
ECHECS*		

* : Étant donné un plateau partiellement joué, existe-t-il une stratégie gagnante à coup sûr pour le joueur 1 ?

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	Rapide
FACTORISER UN NOMBRE	Lent	Rapide
ECHECS*	Lent	

* : Étant donné un plateau partiellement joué, existe-t-il une stratégie gagnante à coup sûr pour le joueur 1 ?

Classer les problèmes (intuitivement)

	Chercher une solution	**Vérifier** une solution
$2-$ COLORATION	Rapide	Rapide
$3-$ COLORATION	Lent	Rapide
SUDOKU	Lent	Rapide
FACTORISER UN NOMBRE	Lent	Rapide
ECHECS*	Lent	Lent

* : Étant donné un plateau partiellement joué, existe-t-il une stratégie gagnante à coup sûr pour le joueur 1 ?

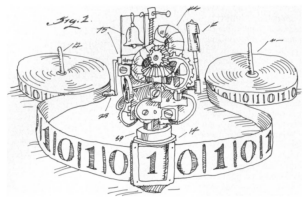
Plan de la matinée

Première partie : P vs NP

1. Mesurer le temps de calcul
2. Les classes P et NP
3. La question P versus NP
4. Les réductions : comparer la difficulté des problèmes
5. La NP-complétude : un seul problème pour les gouverner tous

Deuxième partie : Élargissements

1. Formalisation de l'informatique
2. Calculabilité
3. Complexité algorithmique
4. Autres sujets liés à P vs NP



Mesurer le temps de calcul

Mesurer le temps de calcul

Comment mesure-t-on le temps d'exécution ? **Nombre d'opérations**.

Mesurer le temps de calcul

Comment mesure-t-on le temps d'exécution ? **Nombre d'opérations**.

Exemple : chercher si une liste de n éléments contient un élément e donné.

Mesurer le temps de calcul

Comment mesure-t-on le temps d'exécution ? **Nombre d'opérations**.

Exemple : chercher si une liste de n éléments contient un élément e donné.

Un algorithme possible : parcourir tous les éléments de la liste et les comparer à `e` :

```
def contient(L, e):
    for x in L:
        if x == e:
            return True
    return False
```

Combien d'opérations **dans le pire des cas** ?

Mesurer le temps de calcul

Comment mesure-t-on le temps d'exécution ? **Nombre d'opérations**.

Exemple : chercher si une liste de n éléments contient un élément e donné.

Un algorithme possible : parcourir tous les éléments de la liste et les comparer à `e` :

```
def contient(L, e):
    for x in L:
        if x == e:
            return True
    return False
```

Combien d'opérations **dans le pire des cas** ?

Cela dépend de la **taille de l'entrée** n !

Mesurer le temps de calcul

Comment mesure-t-on le temps d'exécution ? **Nombre d'opérations**.

Exemple : chercher si une liste de n éléments contient un élément e donné.

Un algorithme possible : parcourir tous les éléments de la liste et les comparer à `e` :

```
def contient(L, e):
    for x in L:
        if x == e:
            return True
    return False
```

Combien d'opérations **dans le pire des cas** ?

Cela dépend de la **taille de l'entrée** n !

Ici, le temps est **proportionnel** à n, ce que l'on note $\mathcal{O}(n)$ (temps **linéaire**).

(p.ex. doubler la taille de L revient à doubler le temps d'exécution)

Mesurer le temps de calcul

Autre exemple : détecter si une liste a des doublons (deux élements identiques).

Mesurer le temps de calcul

Autre exemple : détecter si une liste a des doublons (deux élements identiques).

Un algorithme possible : comparer toutes les paires d'éléments de L.

```
def contient(L, e):
    for i in range(len(L)):
        for j in range(len(L)):
            if i != j and L[i] == L[j]:
                return True
    return False
```

Mesurer le temps de calcul

Autre exemple : détecter si une liste a des doublons (deux élements identiques).

Un algorithme possible : comparer toutes les paires d'éléments de L.

```
def contient(L, e):
    for i in range(len(L)):
        for j in range(len(L)):
            if i != j and L[i] == L[j]:
                return True
    return False
```

Ici, le nombre d'opérations est de $\mathcal{O}(n^2)$ (doubler la taille de L revient à **quadrupler** le temps)

Mesurer le temps de calcul

Autre exemple : détecter si une liste a des doublons (deux élements identiques).

Un algorithme possible : comparer toutes les paires d'éléments de L.

```
def contient(L, e):
    for i in range(len(L)):
        for j in range(len(L)):
            if i != j and L[i] == L[j]:
                return True
    return False
```

Ici, le nombre d'opérations est de $\mathcal{O}(n^2)$ (doubler la taille de L revient à **quadrupler** le temps)

Un peu plus malin :

```
def contient(L, e):
    for i in range(0,len(L)-1):
        for j in range(i+1,len(L)):
            if L[i] == L[j]:
                return True
    return False
```

Mesurer le temps de calcul

Autre exemple : détecter si une liste a des doublons (deux élements identiques).

Un algorithme possible : comparer toutes les paires d'éléments de L.

```
def contient(L, e):
    for i in range(len(L)):
        for j in range(len(L)):
            if i != j and L[i] == L[j]:
                return True
    return False
```

Ici, le nombre d'opérations est de $\mathcal{O}(n^2)$ (doubler la taille de L revient à **quadrupler** le temps)

Un peu plus malin :

```
def contient(L, e):
    for i in range(0,len(L)-1):
        for j in range(i+1,len(L)):
            if L[i] == L[j]:
                return True
    return False
```

$\rightarrow$ Deux fois plus rapide, mais toujours $\mathcal{O}(n^2)$. (La notation $\mathcal{O}$ néglige les facteurs constants)

Mesurer le temps de calcul

Autre exemple : détecter si une liste a des doublons (deux élements identiques).

Un algorithme possible : comparer toutes les paires d'éléments de L.

```
def contient(L, e):
    for i in range(len(L)):
        for j in range(len(L)):
            if i != j and L[i] == L[j]:
                return True
    return False
```

Ici, le nombre d'opérations est de $\mathcal{O}(n^2)$ (doubler la taille de L revient à **quadrupler** le temps)

Un peu plus malin :

```
def contient(L, e):
    for i in range(0,len(L)-1):
        for j in range(i+1,len(L)):
            if L[i] == L[j]:
                return True
    return False
```

$\rightarrow$ Deux fois plus rapide, mais toujours $\mathcal{O}(n^2)$. (La notation $\mathcal{O}$ néglige les facteurs constants)

Encore plus malin ?

Mesurer le temps de calcul

Autre exemple : détecter si une liste a des doublons (deux élements identiques).

Un algorithme possible : comparer toutes les paires d'éléments de L.

```
def contient(L, e):
    for i in range(len(L)):
        for j in range(len(L)):
            if i != j and L[i] == L[j]:
                return True
    return False
```

Ici, le nombre d'opérations est de $\mathcal{O}(n^2)$ (doubler la taille de L revient à **quadrupler** le temps)

Un peu plus malin :

```
def contient(L, e):
    for i in range(0,len(L)-1):
        for j in range(i+1,len(L)):
            if L[i] == L[j]:
                return True
    return False
```

$\rightarrow$ Deux fois plus rapide, mais toujours $\mathcal{O}(n^2)$. (La notation $\mathcal{O}$ néglige les facteurs constants)

Encore plus malin ? Trier la liste (temps $O(n \log n)$), puis vérifier les paires consécutives ($O(n)$). Beaucoup mieux !

Complexité d'un problème

Nombre d'opérations requises (dans le pire des cas) par un algorithme qui résoud ce problème, en fonction de la taille de l'entrée n.

Complexité d'un problème

Nombre d'opérations requises (dans le pire des cas) par un algorithme qui résoud ce problème, en fonction de la taille de l'entrée n.

Taille de l'entrée ?

Exemples classiques

une liste L	$n = \|L\|$ en supposant des éléments de taille fixe
un graphe	$n =$ nombre de sommets, $m =$ nombre d'arêtes (convention)
un entier N	$n = \log_2 N$ (représentation binaire)

Complexité d'un problème

Nombre d'opérations requises (dans le pire des cas) par un algorithme qui résoud ce problème, en fonction de la taille de l'entrée n.

Taille de l'entrée ?

Exemples classiques

une liste L	$n = \|L\|$ en supposant des éléments de taille fixe
un graphe	$n =$ nombre de sommets, $m =$ nombre d'arêtes (convention)
un entier N	$n = \log_2 N$ (représentation binaire)

Erreur fréquente : La taille d'un entier N est le nombre de bits qu'il faut pour l'écrire, **pas** sa valeur.

Tester tous les diviseurs de N jusqu'à $\sqrt{N}$ semble rapide, mais ça ne l'est pas en fonction de la taille de l'entrée, car $\sqrt{N} = 2^{n/2}$.

Complexité d'un problème

Nombre d'opérations requises (dans le pire des cas) par un algorithme qui résoud ce problème, en fonction de la taille de l'entrée n.

Taille de l'entrée ?

Exemples classiques

une liste L	$n = \|L\|$ en supposant des éléments de taille fixe
un graphe	$n =$ nombre de sommets, $m =$ nombre d'arêtes (convention)
un entier N	$n = \log_2 N$ (représentation binaire)

Erreur fréquente : La taille d'un entier N est le nombre de bits qu'il faut pour l'écrire, **pas** sa valeur.

Tester tous les diviseurs de N jusqu'à $\sqrt{N}$ semble rapide, mais ça ne l'est pas en fonction de la taille de l'entrée, car $\sqrt{N} = 2^{n/2}$.

Par exemple, factoriser un nombre donné en **unaire** est un problème "facile", mais factoriser ce nombre donné en **binaire** (ou décimal) est un problème "difficile".

Une frontière naturelle : le temps polynomial

Convention : un algorithme est **rapide** si le nombre d'opérations est en $\mathcal{O}(n^c)$ pour une constante c fixée.

Pourquoi cette frontière-là ?

- **Robuste** : ne dépend pas du modèle de calcul ni du langage de programmation.
- **Stable** : la somme et le produit de polynômes sont des polynômes. Un algorithme polynomial qui en appelle un autre reste polynomial.
- **Réaliste** : dans la pratique, les algos polynomiaux connus ont souvent des exposants ≤ 3.

Une frontière naturelle : le temps polynomial

Convention : un algorithme est **rapide** si le nombre d'opérations est en $\mathcal{O}(n^c)$ pour une constante c fixée.

Pourquoi cette frontière-là ?

- **Robuste** : ne dépend pas du modèle de calcul ni du langage de programmation.
- **Stable** : la somme et le produit de polynômes sont des polynômes. Un algorithme polynomial qui en appelle un autre reste polynomial.
- **Réaliste** : dans la pratique, les algos polynomiaux connus ont souvent des exposants ≤ 3.

Un temps qui n'est pas polynomial ?

- $\mathcal{O}(2^n)$ (temps exponentiel)
- $\mathcal{O}(n!)$ (temps factoriel)

Pour avoir une idée, à 10^9 opérations par seconde :

	$n = 20$	$n = 50$	$n = 100$
n^3	$8\ \mu$s	$0{,}1$ ms	1 ms
2^n	1 ms	13 jours	4×10^{13} ans
$n!$	77 ans	inimaginable	

Une frontière naturelle : le temps polynomial

Convention : un algorithme est **rapide** si le nombre d'opérations est en $\mathcal{O}(n^c)$ pour une constante c fixée.

Pourquoi cette frontière-là ?

- **Robuste** : ne dépend pas du modèle de calcul ni du langage de programmation.
- **Stable** : la somme et le produit de polynômes sont des polynômes. Un algorithme polynomial qui en appelle un autre reste polynomial.
- **Réaliste** : dans la pratique, les algos polynomiaux connus ont souvent des exposants ≤ 3.

Un temps qui n'est pas polynomial ?

- $\mathcal{O}(2^n)$ (temps exponentiel)
- $\mathcal{O}(n!)$ (temps factoriel)

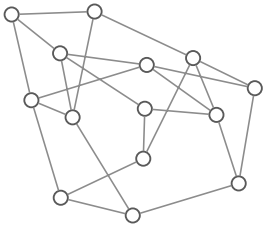
Pour avoir une idée, à 10^9 opérations par seconde :

	$n = 20$	$n = 50$	$n = 100$
n^3	$8\ \mu$s	$0{,}1$ ms	1 ms
2^n	1 ms	13 jours	4×10^{13} ans
$n!$	77 ans	inimaginable	

← 3000 fois l'âge de l'univers...

Retour sur la coloration de graphes

2-Coloration



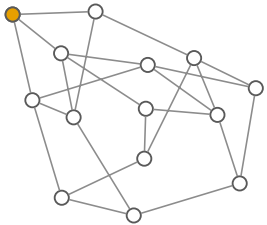
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration



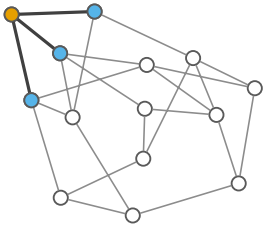
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration



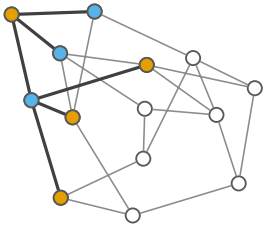
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration



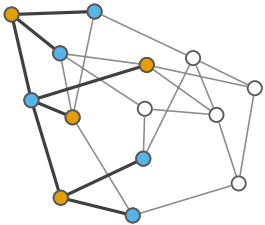
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration



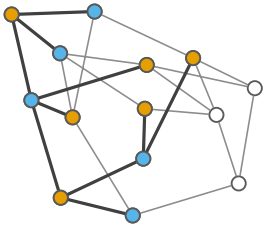
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration



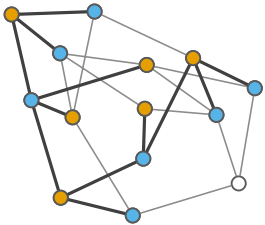
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration



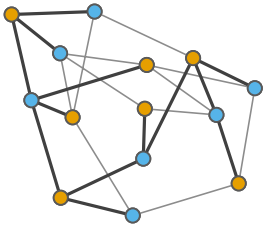
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration



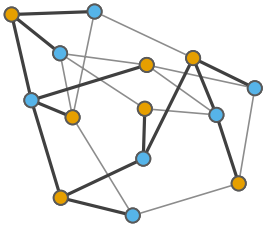
Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n + m)$, **linéaire** en la taille de l'entrée.

Retour sur la coloration de graphes

2-Coloration

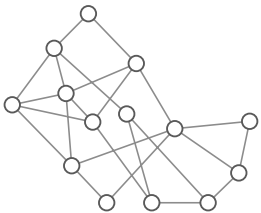


Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n+m)$, **linéaire** en la taille de l'entrée.

3-Coloration

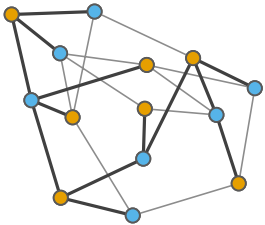


Algorithme :

Meilleur algo connu : $\mathcal{O}^*(1.3289^n)$ **exponentiel** :-(

Retour sur la coloration de graphes

2-Coloration

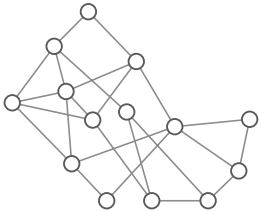


Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n+m)$, **linéaire** en la taille de l'entrée.

3-Coloration



Algorithme :

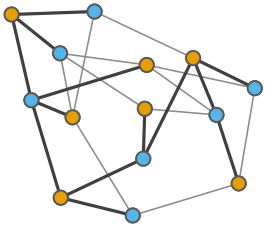
Meilleur algo connu : $\mathcal{O}^*(1.3289^n)$ **exponentiel** :-(

Toujours mieux que brute-force en $\mathcal{O}(3^n)$...

On ne connaît pas d'algorithme **polynomial**.

Retour sur la coloration de graphes

2-Coloration

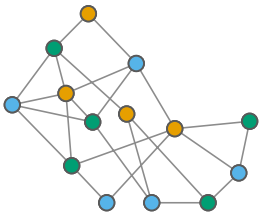


Algorithme :

- Choisir un sommet de départ ; lui donner la couleur 1.
- Faire un parcours en profondeur, colorier en alternant.
- En cas de problème détecté, arrêter et renvoyer NON.
- Si on termine sans problème, renvoyer OUI.

- **Aucun choix** n'est jamais fait : ça passe ou ça casse. Cet algo fonctionne ssi c'est faisable (graphe biparti)
- Coût : peut être réalisé en temps $\mathcal{O}(n+m)$, **linéaire** en la taille de l'entrée.

3-Coloration



Algorithme :

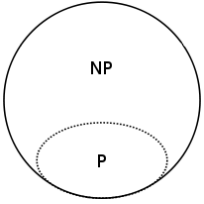
Meilleur algo connu : $\mathcal{O}^*(1.3289^n)$ **exponentiel** :-(

Toujours mieux que brute-force en $\mathcal{O}(3^n)$...

On ne connaît pas d'algorithme **polynomial**.

Malgré tout, une solution est **vérifiable** en temps $O(m)$.

Les classes P et NP

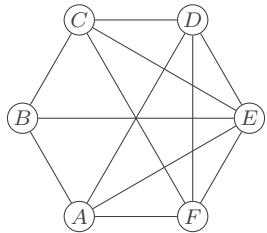


Avant de commencer : quelques protagonistes

- CLIQUE

Étant donné un graphe G et un entier k, est-ce que G contient un sous-graphe complet de taille k ?

Exemple : ce graphe et $k = 4$:

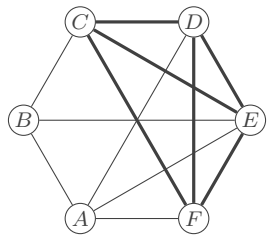


Avant de commencer : quelques protagonistes

- CLIQUE

Étant donné un graphe G et un entier k, est-ce que G contient un sous-graphe complet de taille k ?

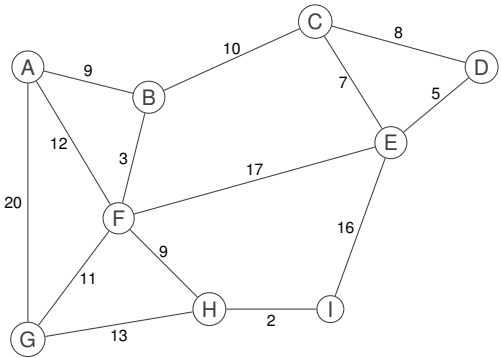
Exemple : ce graphe et $k = 4$:



Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant

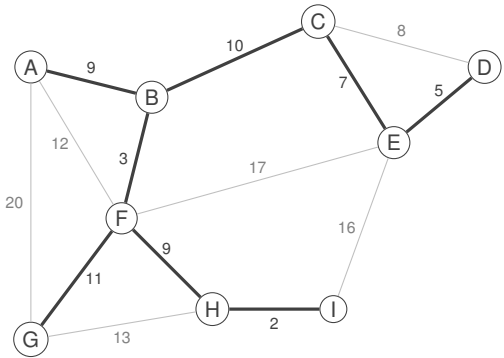
Étant donné un graphe G dont les arêtes ont des poids, trouver un arbre de poids minimum qui relie tous les sommets. (arbre = graphe sans cycle)



Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant

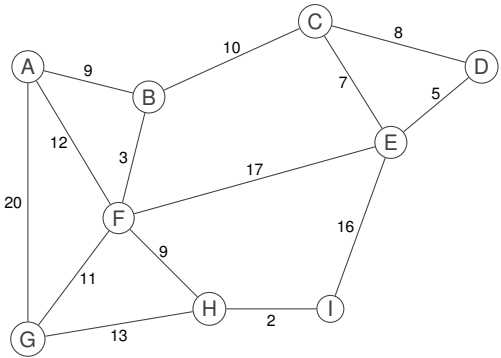
Étant donné un graphe G dont les arêtes ont des poids, trouver un arbre de poids minimum qui relie tous les sommets. (arbre = graphe sans cycle)



Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant
- Voyageur de Commerce

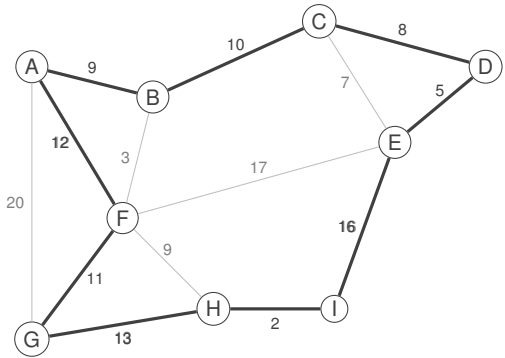
Voyageur de Commerce : Étant donné un graphe G dont les arêtes ont des poids, trouver un cycle de poids minimum qui visite exactement une fois chaque sommet.



Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant
- Voyageur de Commerce

Voyageur de Commerce : Étant donné un graphe G dont les arêtes ont des poids, trouver un cycle de poids minimum qui visite exactement une fois chaque sommet.



Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant
- Voyageur de Commerce
- SAT

SAT : Étant donné une formule booléenne ϕ sous forme « normale conjonctive », existe-t-il des valeurs vrai/faux pour les variables qui satisfont ϕ ?

Exemple :

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_5) \wedge (x_5 \vee \neg x_6) \wedge (x_6 \vee \neg x_3 \vee \neg x_4)$$

Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant
- Voyageur de Commerce
- SAT

SAT : Étant donné une formule booléenne ϕ sous forme « normale conjonctive », existe-t-il des valeurs vrai/faux pour les variables qui satisfont ϕ ?

Exemple :

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_5) \wedge (x_5 \vee \neg x_6) \wedge (x_6 \vee \neg x_3 \vee \neg x_4)$$

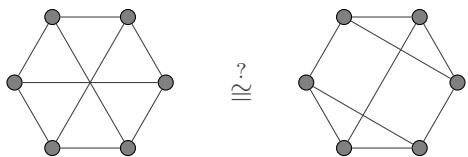
Une solution (parmi d'autres) :

x_1	x_2	x_3	x_4	x_5	x_6
V	V	F	V	F	F.

Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant
- Voyageur de Commerce
- SAT
- Isomorphisme de Graphes

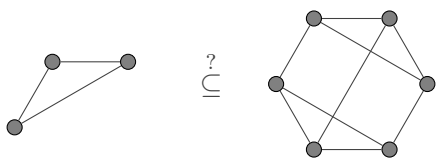
Isomorphisme de Graphes : Étant donné deux graphes G_1 et G_2, sont-ils structurellement identiques ?



Avant de commencer : quelques protagonistes

- Clique
- Arbre Couvrant
- Voyageur de Commerce
- SAT
- Isomorphisme de Graphes
- Isomorphisme de Sous-graphes

Isomorphisme de Sous-graphes : Étant donné deux graphes G_1 et G_2, est-ce que G_1 est un sous-graphe de G_2 ?



Autre détail important : Problèmes de décision

Focus sur les problèmes de **décision** : problèmes dont la réponse est **oui** ou **non**.

Très peu de perte de généralité.

Autre détail important : Problèmes de décision

Focus sur les problèmes de **décision** : problèmes dont la réponse est **oui** ou **non**.

Très peu de perte de généralité.

Exemple : Trouver une clique *vs* savoir si elle existe.

Étant donné un graphe G et un entier k, comment trouver une clique de taille k en utilisant un algorithme pour la version oui/non (un **oracle**) ?

Autre détail important : Problèmes de décision

Focus sur les problèmes de **décision** : problèmes dont la réponse est **oui** ou **non**.

Très peu de perte de généralité.

Exemple : Trouver une clique *vs* savoir si elle existe.

Étant donné un graphe G et un entier k, comment trouver une clique de taille k en utilisant un algorithme pour la version oui/non (un **oracle**) ?

1. Commencer par donner (G, k) à l'oracle. S'il répond non, ce n'est pas la peine de chercher.
2. Puis, pour chaque sommet v de G, on teste $(G \setminus \{v\}, k)$. À chaque fois que la réponse est oui, on enlève réellement ce sommet et on continue.
3. Lorsqu'on a terminé, le graphe restant est une clique de taille k.

On peut donc réduire la version **recherche** du problème à la version **décisionnelle**, en temps **polynomial**.

Autre détail important : Problèmes de décision

Focus sur les problèmes de **décision** : problèmes dont la réponse est **oui** ou **non**.

Très peu de perte de généralité.

Exemple : Trouver une clique *vs* savoir si elle existe.

Étant donné un graphe G et un entier k, comment trouver une clique de taille k en utilisant un algorithme pour la version oui/non (un **oracle**) ?

1. Commencer par donner (G, k) à l'oracle. S'il répond non, ce n'est pas la peine de chercher.
2. Puis, pour chaque sommet v de G, on teste $(G \setminus \{v\}, k)$. À chaque fois que la réponse est oui, on enlève réellement ce sommet et on continue.
3. Lorsqu'on a terminé, le graphe restant est une clique de taille k.

On peut donc réduire la version **recherche** du problème à la version **décisionnelle**, en temps **polynomial**.

Cette propriété est vraie pour tous les problèmes qui nous intéressent aujourd'hui !

La classe **P**

P = ensemble des problèmes de décision
que l'on sait résoudre en temps **polynomial**.

« Résoudre » : produire la bonne réponse, OUI ou NON, sur **toute** entrée.

La classe NP

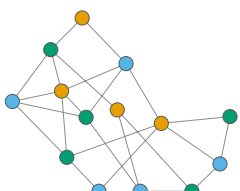
NP : ensemble des problèmes de décisions tels que, si **la réponse est** OUI, alors il existe un **certificat** (une solution, une preuve) tel que :

- la **taille** du certificat est polynomiale en la taille de l'entrée ;
- le certificat permet de **vérifier** la réponse en temps polynomial.

La classe NP

NP : ensemble des problèmes de décisions tels que, si **la réponse est** OUI, alors il existe un **certificat** (une solution, une preuve) tel que :

- la **taille** du certificat est polynomiale en la taille de l'entrée ;
- le certificat permet de **vérifier** la réponse en temps polynomial.



Vous n'aviez pas trouvé cette coloration.

Mais dès qu'on vous l'a **montré**, vous avez su, en 30 secondes, qu'elle était correcte.

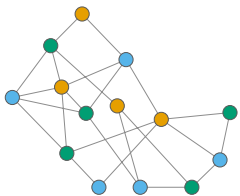
Cette coloration est un **certificat** que le graphe est 3-colorable.

Rien n'est dit sur la manière de **trouver** ce certificat.

La classe NP

NP : ensemble des problèmes de décisions tels que, si **la réponse est** OUI, alors il existe un **certificat** (une solution, une preuve) tel que :

- la **taille** du certificat est polynomiale en la taille de l'entrée ;
- le certificat permet de **vérifier** la réponse en temps polynomial.



Vous n'aviez pas trouvé cette coloration.

Mais dès qu'on vous l'a **montré**, vous avez su, en 30 secondes, qu'elle était correcte.

Cette coloration est un **certificat** que le graphe est 3-colorable.

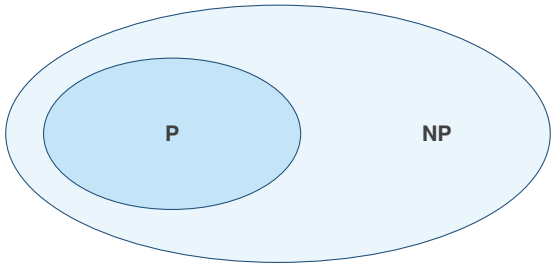
Rien n'est dit sur la manière de **trouver** ce certificat.

P $=$ problèmes que l'on peut **résoudre** rapidement

NP $=$ problèmes que l'on peut **vérifier** rapidement.

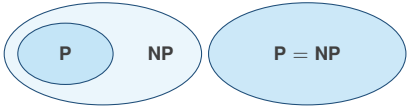
$P \subseteq NP$

Si je sais **résoudre** un problème en temps polynomial,
je sais aussi **vérifier** en temps polynomial :
l'algorithme lui-même est un certificat qu'on peut exécuter pour vérifier.



P versus NP

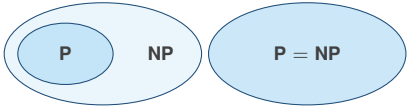
Les questions faciles à vérifier sont-elles faciles à résoudre ?
(Est-ce que P $=$ NP ?)



Deux mondes possibles

P versus NP

Les questions faciles à vérifier sont-elles faciles à résoudre ?
(Est-ce que P $=$ NP ?)



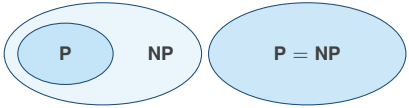
Deux mondes possibles

Importance de la question

- La **majorité** des problèmes utiles sont dans NP.
  Si P $=$ NP, on peut tous les résoudre rapidement.
- Serait-ce une bonne nouvelle ?

P versus NP

Les questions faciles à vérifier sont-elles faciles à résoudre ?
(Est-ce que P $=$ NP ?)



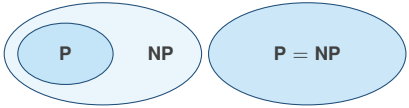
Deux mondes possibles

Importance de la question

- La **majorité** des problèmes utiles sont dans NP.
  Si P $=$ NP, on peut tous les résoudre rapidement.
- Serait-ce une bonne nouvelle ? Oui et non (cryptographie).

P versus NP

Les questions faciles à vérifier sont-elles faciles à résoudre ?
(Est-ce que P $=$ NP ?)



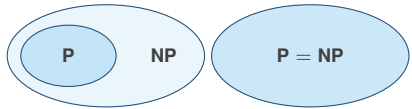
Deux mondes possibles

Importance de la question

- La **majorité** des problèmes utiles sont dans NP.
  Si P $=$ NP, on peut tous les résoudre rapidement.
- Serait-ce une bonne nouvelle ? Oui et non (cryptographie).
- Un des 7 “problèmes du millénaire” (fondation Clay, $1 M / pb).

P versus NP

Les questions faciles à vérifier sont-elles faciles à résoudre ?
(Est-ce que P = NP ?)



Deux mondes possibles

Importance de la question

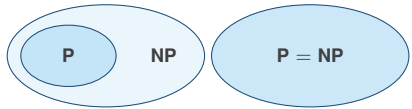
- La **majorité** des problèmes utiles sont dans NP.
  Si P = NP, on peut tous les résoudre rapidement.
- Serait-ce une bonne nouvelle ? Oui et non (cryptographie).
- Un des 7 “problèmes du millénaire” (fondation Clay, $1 M / pb).

Portée philosophique et mathématique ?

- Les énoncés mathématiques dont les preuves sont “faciles” à vérifier sont-ils “faciles” à démontrer ?
- Le fait que je puisse comprendre quelque chose implique-t-il que je puisse le découvrir ?
- Si je sais apprécier une symphonie, cela implique-t-il que j’aurais pu la composer ?
- *etc.*

P versus NP

Les questions faciles à vérifier sont-elles faciles à résoudre ?
(Est-ce que P $=$ NP ?)



Deux mondes possibles

Importance de la question

- La **majorité** des problèmes utiles sont dans NP.
  Si P $=$ NP, on peut tous les résoudre rapidement.
- Serait-ce une bonne nouvelle ? Oui et non (cryptographie).
- Un des 7 “problèmes du millénaire” (fondation Clay, $1 M / pb).

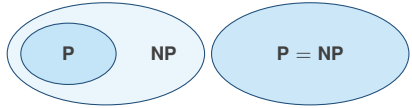
Portée philosophique et mathématique ?

- Les énoncés mathématiques dont les preuves sont “faciles” à vérifier sont-ils “faciles” à démontrer ?
- Le fait que je puisse comprendre quelque chose implique-t-il que je puisse le découvrir ?
- Si je sais apprécier une symphonie, cela implique-t-il que j’aurais pu la composer ?
- *etc.*

bémols : formalisation + quid de $O(n^{100})$?

P versus NP

Les questions faciles à vérifier sont-elles faciles à résoudre ?
(Est-ce que P $=$ NP ?)



Deux mondes possibles

Importance de la question

- La **majorité** des problèmes utiles sont dans NP.
  Si P $=$ NP, on peut tous les résoudre rapidement.
- Serait-ce une bonne nouvelle ? Oui et non (cryptographie).
- Un des 7 “problèmes du millénaire” (fondation Clay, $1 M / pb).

Portée philosophique et mathématique ?

- Les énoncés mathématiques dont les preuves sont “faciles” à vérifier sont-ils “faciles” à démontrer ?
- Le fait que je puisse comprendre quelque chose implique-t-il que je puisse le découvrir ?
- Si je sais apprécier une symphonie, cela implique-t-il que j’aurais pu la composer ?
- *etc.*

bémols : formalisation + quid de $O(n^{100})$?

À ce jour, on ne connaît pas la réponse. Qu’en pensez-vous ?

Retour sur nos protagonistes

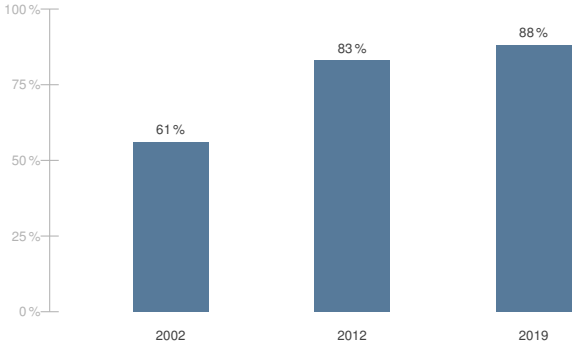
Problème (v. décision)	dans P	dans NP	certificat
2-COLORATION	✓	✓	algo (ou coloration)
3-COLORATION	X (?)	✓	coloration
Primalité	✓	✓	algo (ou Pratt)
Factorization	X (?)	✓	diviseur
ARBRE COUVRANT MINIMUM	✓	✓	algo (ou l'arbre)
VOYAGEUR DE COMMERCE	X (?)	✓	le cycle
CLIQUE	X (?)	✓	le clique
ISOMORPHISME DE GRAPHES	X (?)	✓	correspondance
ISOMORPHISME DE SOUS-GRAPHES	X (?)	✓	le sous-graphe
SAT	X (?)	✓	les valeurs de vérité
Sudoku (généralisé)	X (?)	✓	la grille remplie
Échecs (généralisé)	X (?)	X (?)	?

Retour sur nos protagonistes

Problème (v. décision)	dans P	dans NP	certificat
2-COLORATION	✓	✓	algo (ou coloration)
3-COLORATION	X (?)	✓	coloration
Primalité	✓	✓	algo (ou Pratt)
Factorization	X (?)	✓	diviseur
ARBRE COUVRANT MINIMUM	✓	✓	algo (ou l'arbre)
VOYAGEUR DE COMMERCE	X (?)	✓	le cycle
CLIQUE	X (?)	✓	le clique
ISOMORPHISME DE GRAPHES	X (?)	✓	correspondance
ISOMORPHISME DE SOUS-GRAPHES	X (?)	✓	le sous-graphe
SAT	X (?)	✓	les valeurs de vérité
Sudoku (généralisé)	X (?)	✓	la grille remplie
Échecs (généralisé)	X (?)	X (?)	?

Changement d'avis ?

Ce que pensent les spécialistes



Part des chercheurs interrogés qui parient sur **P** $\neq$ **NP**.

Sondages de William Gasarch, menés auprès de la communauté en théorie de la complexité.

Conviction assez large.

Erreurs fréquentes

NP ne signifie **pas** « non polynomial ».

Le « N » vient de **non-déterministe**, un vocabulaire hérité d'une autre définition équivalente. Beaucoup de documents de vulgarisation se trompent sur ce point.

Erreurs fréquentes

NP ne signifie **pas** « non polynomial ».

Le « N » vient de **non-déterministe**, un vocabulaire hérité d'une autre définition équivalente. Beaucoup de documents de vulgarisation se trompent sur ce point.

NP n'est pas « l'ensemble des problèmes difficiles ».

C'est l'ensemble des problèmes dont les solutions sont faciles à **vérifier**, ça reste une bonne nouvelle.

Erreurs fréquentes

NP ne signifie **pas** « non polynomial ».

Le « N » vient de **non-déterministe**, un vocabulaire hérité d'une autre définition équivalente. Beaucoup de documents de vulgarisation se trompent sur ce point.

NP n'est pas « l'ensemble des problèmes difficiles ».

C'est l'ensemble des problèmes dont les solutions sont faciles à **vérifier**, ça reste une bonne nouvelle.

Mais alors, pourquoi est-ce qu'on a peur quand on entend **NP** ?

NP-complétude

Comparer deux problèmes

On ne sait pas montrer qu'un problème est difficile.

En revanche, on peut **comparer** la difficulté entre problèmes.

C'est peut-être l'idée la plus importante de la théorie

Réductions entre problèmes

Soit A et B deux problèmes. A **se réduit à** B **en temps polynomial**, noté $A \leq_p B$ si :

1. Pour toute instance I_A du problème A, on peut la transformer en une instance I_B pour le problème B
2. Cette transformation se fait en temps polynomial en la taille de l'entrée ($|I_A|$)
3. La réponse pour I_B est la même que la réponse pour I_A.

Réductions entre problèmes

Soit A et B deux problèmes. A **se réduit à** B **en temps polynomial**, noté $A \leq_p B$ si :

1. Pour toute instance I_A du problème A, on peut la transformer en une instance I_B pour le problème B
2. Cette transformation se fait en temps polynomial en la taille de l'entrée ($|I_A|$)
3. La réponse pour I_B est la même que la réponse pour I_A.

Autrement dit, $A \leq_p B$ signifie qu'on peut utiliser un algorithme pour B afin de résoudre A.

Ce qui implique :

Le problème B est **au moins aussi difficile** que le problème A.

Réductions entre problèmes

Soit A et B deux problèmes. A **se réduit à** B **en temps polynomial**, noté $A \leq_p B$ si :

1. Pour toute instance I_A du problème A, on peut la transformer en une instance I_B pour le problème B
2. Cette transformation se fait en temps polynomial en la taille de l'entrée ($|I_A|$)
3. La réponse pour I_B est la même que la réponse pour I_A.

Autrement dit, $A \leq_p B$ signifie qu'on peut utiliser un algorithme pour B afin de résoudre A.

Ce qui implique :

Le problème B est **au moins aussi difficile** que le problème A.

- Si $B \in \mathbf{P}$ alors $A \in \mathbf{P}$.
- Si $A \notin \mathbf{P}$ alors $B \notin \mathbf{P}$.

Réductions entre problèmes

Soit A et B deux problèmes. A **se réduit à** B **en temps polynomial**, noté $A \leq_p B$ si :

1. Pour toute instance I_A du problème A, on peut la transformer en une instance I_B pour le problème B
2. Cette transformation se fait en temps polynomial en la taille de l'entrée ($|I_A|$)
3. La réponse pour I_B est la même que la réponse pour I_A.

Autrement dit, $A \leq_p B$ signifie qu'on peut utiliser un algorithme pour B afin de résoudre A.

Ce qui implique :

Le problème B est **au moins aussi difficile** que le problème A.

- Si $B \in$ **P** alors $A \in$ **P**.
- Si $A \notin$ **P** alors $B \notin$ **P**.

Erreur (très) fréquente : se tromper de sens !

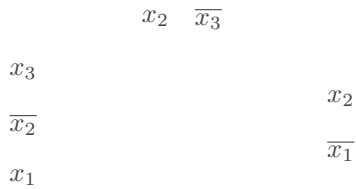
Par exemple, diriez-vous que 2-COLORATION $\leq_p$ 3-COLORATION ou 3-COLORATION $\leq_p$ 2-COLORATION ?

Exemple de réduction : SAT $\leq_p$ CLIQUE

Soit $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$ (3 variables, 3 clauses, 7 litéraux).

Exemple de réduction : SAT $\leq_p$ CLIQUE

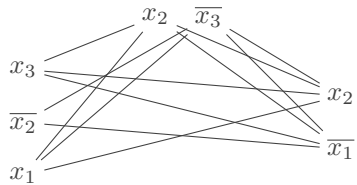
Soit $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$ (3 variables, 3 clauses, 7 litéraux).



- Créer un sommet par litéral.

Exemple de réduction : SAT $\leq_p$ CLIQUE

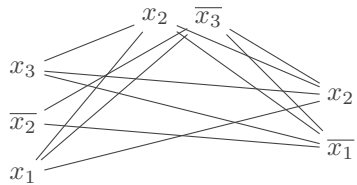
Soit $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$ (3 variables, 3 clauses, 7 litéraux).



- Créer un sommet par litéral.
- Créer une arête entre litéraux qui appartiennent à des clauses **différentes** et sont **compatibles** (p.ex. x_2 et $\overline{x_2}$ ne sont pas compatibles).

Exemple de réduction : SAT $\leq_p$ CLIQUE

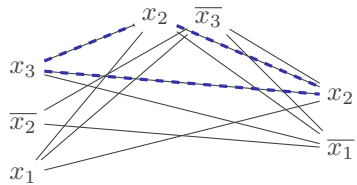
Soit $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$ (3 variables, 3 clauses, 7 litéraux).



- Créer un sommet par litéral.
- Créer une arête entre litéraux qui appartiennent à des clauses **différentes** et sont **compatibles** (p.ex. x_2 et $\overline{x_2}$ ne sont pas compatibles).
- On demande alors : ce graphe contient-il une **clique de taille** 3 ? (nombre de clauses)

Exemple de réduction : SAT $\leq_p$ CLIQUE

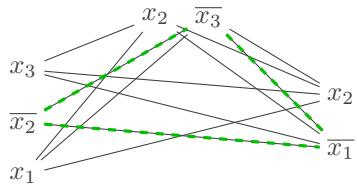
Soit $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$ (3 variables, 3 clauses, 7 litéraux).



- Créer un sommet par litéral.
- Créer une arête entre litéraux qui appartiennent à des clauses **différentes** et sont **compatibles** (p.ex. x_2 et $\overline{x_2}$ ne sont pas compatibles).
- On demande alors : ce graphe contient-il une **clique de taille** 3 ? (nombre de clauses)

Exemple de réduction : SAT $\leq_p$ CLIQUE

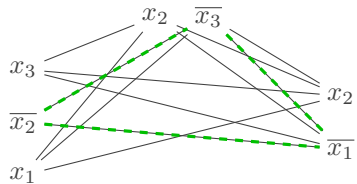
Soit $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$ (3 variables, 3 clauses, 7 litéraux).



- Créer un sommet par litéral.
- Créer une arête entre litéraux qui appartiennent à des clauses **différentes** et sont **compatibles** (p.ex. x_2 et $\overline{x_2}$ ne sont pas compatibles).
- On demande alors : ce graphe contient-il une **clique de taille** 3 ? (nombre de clauses)

Exemple de réduction : SAT $\leq_p$ CLIQUE

Soit $\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2)$ (3 variables, 3 clauses, 7 litéraux).



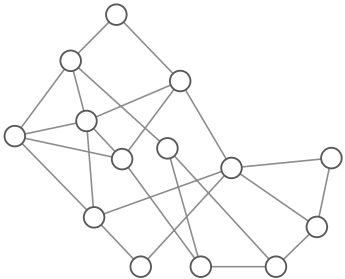
- Créer un sommet par litéral.
- Créer une arête entre litéraux qui appartiennent à des clauses **différentes** et sont **compatibles** (p.ex. x_2 et $\overline{x_2}$ ne sont pas compatibles).
- On demande alors : ce graphe contient-il une **clique de taille** 3 ? (nombre de clauses)

Formule à k clauses satisfaisable $\iff$ Clique de taille k dans le graphe créé.

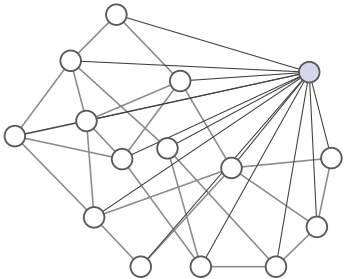
Cette construction prend un temps polynomial. Un algorithme polynomial pour CLIQUE implique donc un algorithme polynomial pour SAT.

CLIQUE est **au moins aussi difficile** que SAT.

3-Coloration $\leq_p$ 4-Coloration

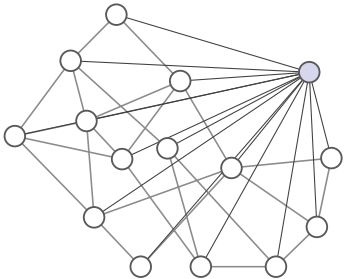


3-Coloration $\leq_p$ 4-Coloration



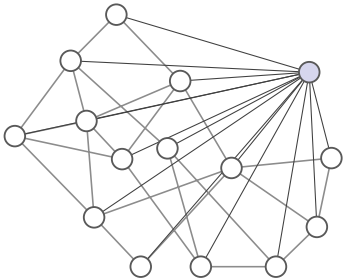
- Créer $G' = G+$ un sommet universel (relié à tous les autres)

3-COLORATION $\leq_p$ 4-COLORATION



- Créer $G' = G+$ un sommet universel (relié à tous les autres)
- Demander à l'algo de 4-COLORATION si G' est 4-colorable.

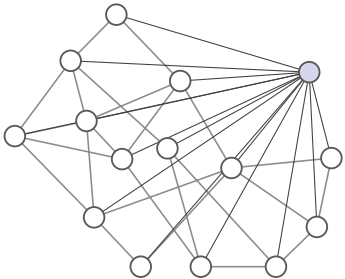
3-COLORATION $\leq_p$ 4-COLORATION



- Créer $G' = G+$ un sommet universel (relié à tous les autres)
- Demander à l'algo de 4-COLORATION si G' est 4-colorable.

G est 3-colorable $\iff$ G' est 4-colorable.

3-COLORATION $\leq_p$ 4-COLORATION

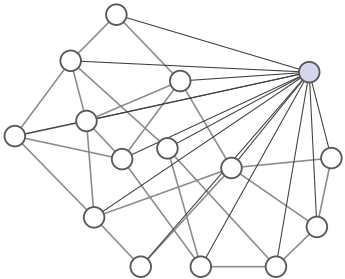


- Créer $G' = G+$ un sommet universel (relié à tous les autres)
- Demander à l'algo de 4-COLORATION si G' est 4-colorable.

G est 3-colorable $\iff$ G' est 4-colorable.

Cette réduction est polynomiale

3-COLORATION $\leq_p$ 4-COLORATION



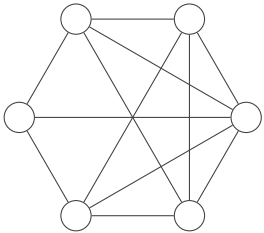
- Créer $G' = G+$ un sommet universel (relié à tous les autres)
- Demander à l'algo de 4-COLORATION si G' est 4-colorable.

G est 3-colorable $\iff$ G' est 4-colorable.

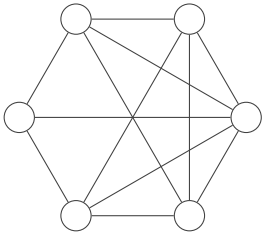
Cette réduction est polynomiale

4-COLORATION est au moins aussi difficile que 3-COLORATION.

Clique $\leq_p$ Isomorphisme de Sous-graphes



CLIQUE $\leq_p$ ISOMORPHISME DE SOUS-GRAPHES

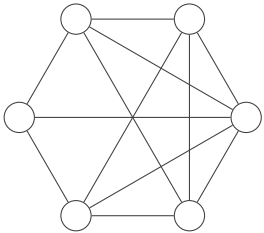


Exemple : On veut savoir si ce graphe G a une clique de taille 4.

1. Créer un graphe complet G' à 4 sommets.

$G' =$ 

CLIQUE $\leq_p$ ISOMORPHISME DE SOUS-GRAPHES



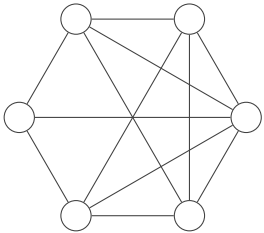
Exemple : On veut savoir si ce graphe G a une clique de taille 4.

1. Créer un graphe complet G' à 4 sommets.

   $G' =$ 

2. Demander à l'algo pour ISOMORPHISME DE SOUS-GRAPHE si $G' \subseteq G$.

CLIQUE $\leq_p$ ISOMORPHISME DE SOUS-GRAPHES



Exemple : On veut savoir si ce graphe G a une clique de taille 4.

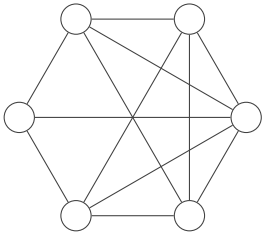
1. Créer un graphe complet G' à 4 sommets.

   $G' =$ 

2. Demander à l'algo pour ISOMORPHISME DE SOUS-GRAPHE si $G' \subseteq G$.

G a une clique de taille $k \iff G' \subseteq G$.

CLIQUE $\leq_p$ ISOMORPHISME DE SOUS-GRAPHES



Exemple : On veut savoir si ce graphe G a une clique de taille 4.

1. Créer un graphe complet G' à 4 sommets.

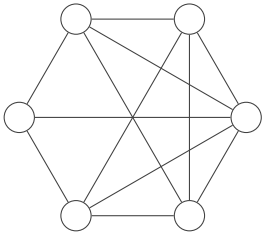
   $G' =$ 

2. Demander à l'algo pour ISOMORPHISME DE SOUS-GRAPHE si $G' \subseteq G$.

$$\boxed{G \text{ a une clique de taille } k \iff G' \subseteq G.}$$

Cette réduction est polynomiale.

CLIQUE $\leq_p$ ISOMORPHISME DE SOUS-GRAPHES



Exemple : On veut savoir si ce graphe G a une clique de taille 4.

1. Créer un graphe complet G' à 4 sommets.

   $G' =$ 

2. Demander à l'algo pour ISOMORPHISME DE SOUS-GRAPHE si $G' \subseteq G$.

G a une clique de taille $k \iff G' \subseteq G$.

Cette réduction est polynomiale.

ISOMORPHISME DE SOUS-GRAPHE est au moins aussi difficile que CLIQUE
(et donc au moins aussi difficile que SAT, c'est transitif !).

Les plus durs de tous

Un problème B est **NP-complet** si

- $B \in$ **NP** ;
- pour **tout** problème $A \in$ **NP**, on a $A \leq_p B$.

Un tel problème contient toute la difficulté de **NP**.

Rien ne garantit a priori qu'il existe un tel problème.

1971 — il en existe

Théorème (Cook, Levin)

SAT est NP-complet.

La preuve :

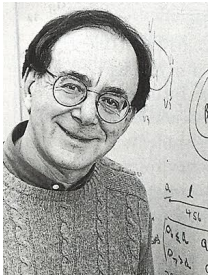
1. tout problème de NP admet un algorithme de **vérification** polynomial
2. l'exécution de cet algorithme peut s'encoder comme une formule booléenne CNF...
3. ... dont la résolution revient à deviner le certificat.

Plus techniquement, la preuve repose sur la définition de NP utilisant les **machines de Turing non-déterministes**.

Stephen Cook (1939–)

Leonid Levin (1948–)

1972 — ils sont partout



Richard Karp (1935–)

Reducibility Among Combinatorial Problems

Karp montre que **21 problèmes** classiques — issus de la recherche opérationnelle, de la théorie des graphes, de l'arithmétique — sont tous NP-complets.

Parmi eux : CLIQUE, 3-COLORATION, SUDOKU, VOYAGEUR DE COMMERCE, ...

Le domaine change de nature : ce n'est plus une curiosité logique, c'est une propriété partagée par les problèmes que les gens essaient réellement de résoudre.

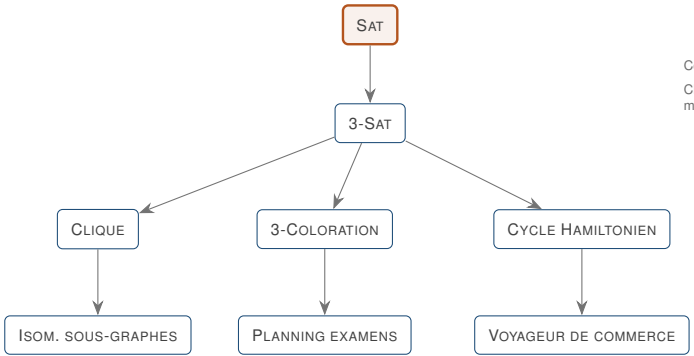
Aujourd'hui : plusieurs milliers de problèmes NP-complets recensés.

Comment on prouve la NP-complétude

Une fois que SAT est montré NP-complet, c'est plus facile.

Pour montrer qu'un problème $B \in$ NP est NP-complet, il suffit de réduire SAT à ce problème !

Quelques réductions classiques :



Cook–Levin fournit la racine.

Chaque flèche est une réduction polynomiale.

Ah mais alors...

Oui, tous les problèmes NP-complets se réduisent à n'importe lequel d'entre eux.

Ah mais alors...

Oui, tous les problèmes NP-complets se réduisent à n'importe lequel d'entre eux.

Un **seul** algorithme polynomial
pour un **seul** problème NP-complet

$$\Longrightarrow$$

$\mathbf{P} = \mathbf{NP}$, et tout NP tombe avec lui.

Ah mais alors...

Oui, tous les problèmes NP-complets se réduisent à n'importe lequel d'entre eux.

Un **seul** algorithme polynomial
pour un **seul** problème NP-complet

$$\Longrightarrow$$

$\mathbf{P} = \mathbf{NP}$, et tout NP tombe avec lui.

Symétriquement : une seule preuve d'impossibilité pour un seul d'entre eux règle la question dans l'autre sens.

Retour (encore) sur nos protagonistes

Problème (v. décision)	dans P	dans NP	NP-complets
2-COLORATION	✓	✓	X
3-COLORATION	X (?)	✓	✓
Primalité	✓	✓	X
Factorization	X (?)	✓	X (?)
ARBRE COUVRANT MINIMUM	✓	✓	X
VOYAGEUR DE COMMERCE	X (?)	✓	✓
CLIQUE	X (?)	✓	✓
ISOMORPHISME DE GRAPHES	X (?)	✓	X (?)
ISOMORPHISME DE SOUS-GRAPHES	X (?)	✓	✓
SAT	X (?)	✓	✓
Sudoku (généralisé)	X (?)	✓	✓
Échecs (généralisé)	X (?)	X (?)	X (?)

Petit bilan

Fin de la première partie...

1. Chercher et vérifier n'ont pas le même coût.
2. Le temps se mesure en fonction de la **taille** de la donnée.
3. Une frontière naturelle est le temps **polynomial**.
4. P : on sait résoudre. NP : on sait vérifier. $P \subseteq NP$.
5. On peut réduire certains problèmes à d'autres problèmes.
6. Les problèmes **NP-complets** concentrent toute la difficulté.
7. La majorité des spécialistes pense que $P \neq NP$
8. Personne n'arrive à le démontrer.