

# INF8175 - Intelligence artificielle

*Méthodes et algorithmes*

## Module 3: Recherche locale



POLYTECHNIQUE  
MONTRÉAL

Quentin Cappart

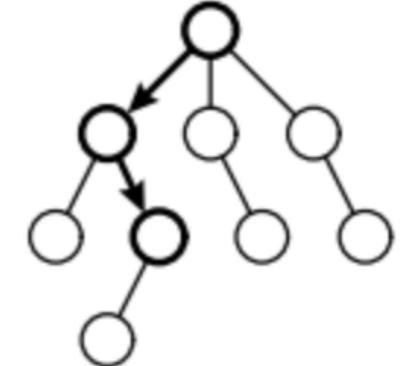
# Contenu du cours

## Raisonnement par recherche (essais-erreurs avec de l'intuition)

**Module 1:** Stratégies de recherche

**Module 2:** Recherche en présence d'adversaires

**Module 3:** Recherche locale



## Raisonnement logique

**Module 4:** Programmation par contraintes

**Module 5:** Agents logiques

ΣΣ ΣΣΣΣ Stench		Breeze	PIT
	Breeze ΣΣ ΣΣΣΣ Stench Gold	PIT	Breeze
ΣΣ ΣΣΣΣ Stench		Breeze	
START	Breeze	PIT	Breeze

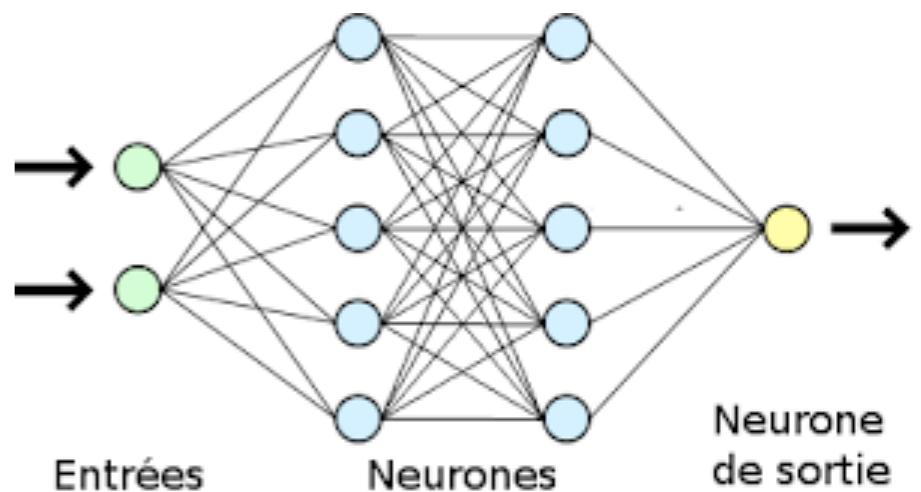
## Raisonnement par apprentissage

**Module 6:** Apprentissage supervisé

**Module 7:** Réseaux de neurones et apprentissage profond

**Module 8:** Apprentissage non-supervisé

**Module 9:** Apprentissage par renforcement



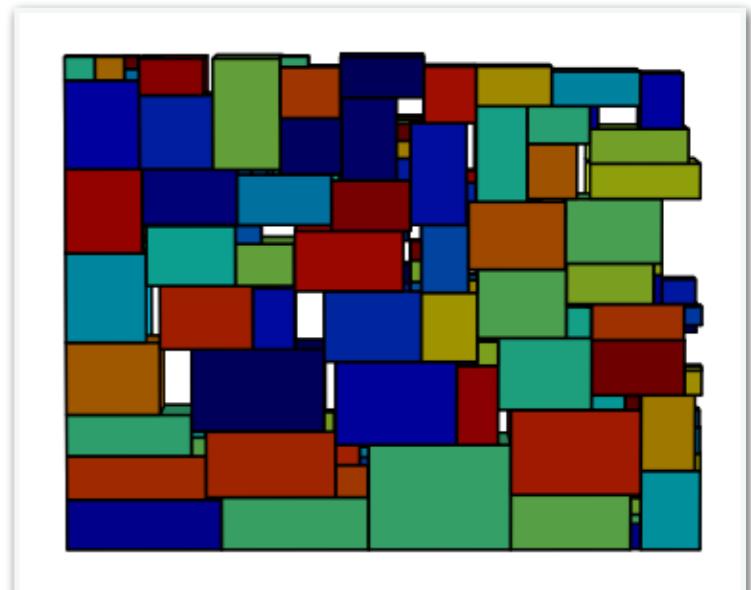
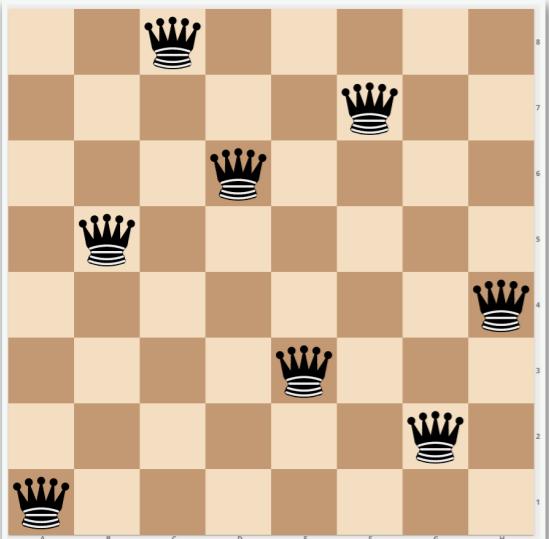
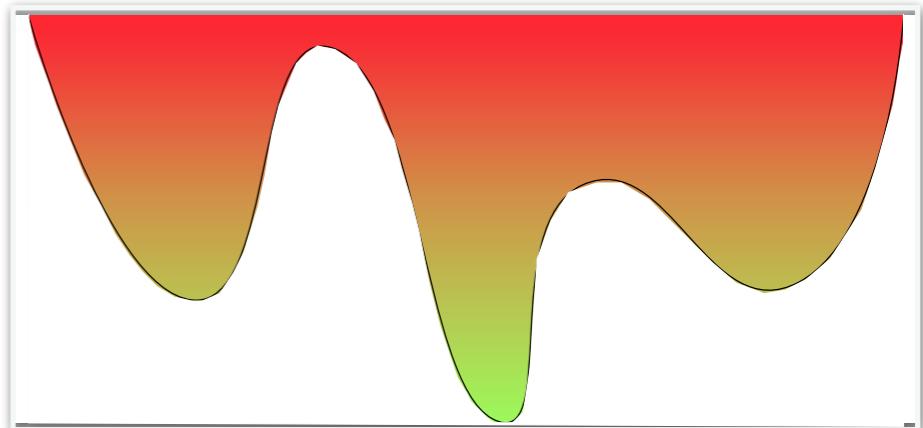
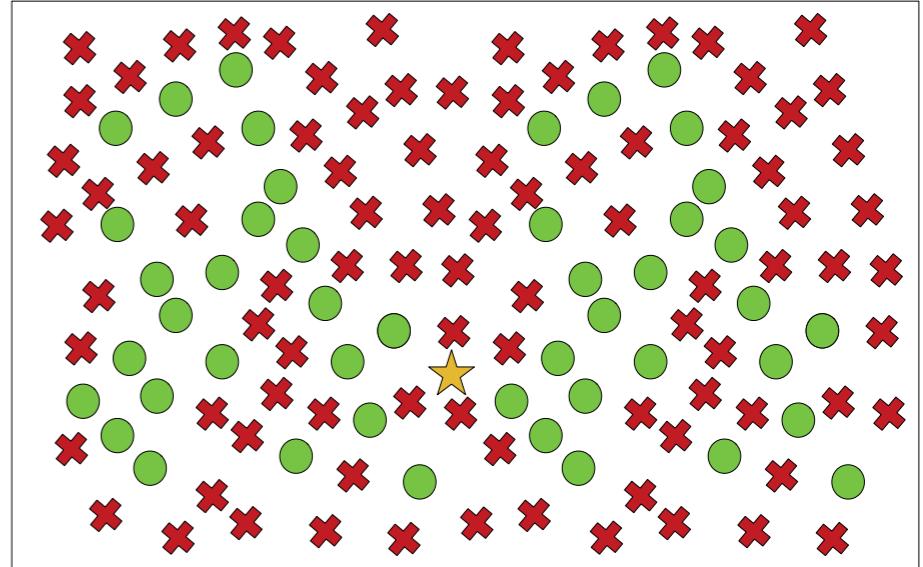
## Considérations pratiques et sociétales

**Module 10:** Utilisation en industrie, éthique, et philosophie

# Table des matières

## Recherche locale

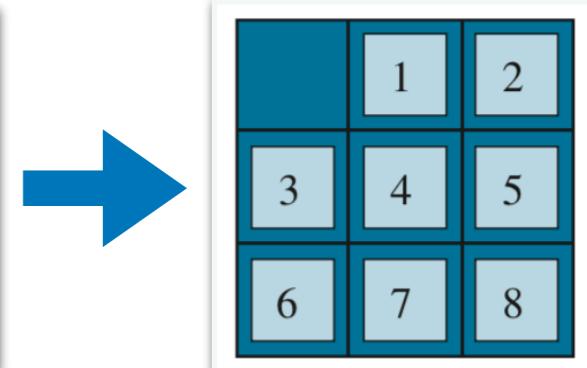
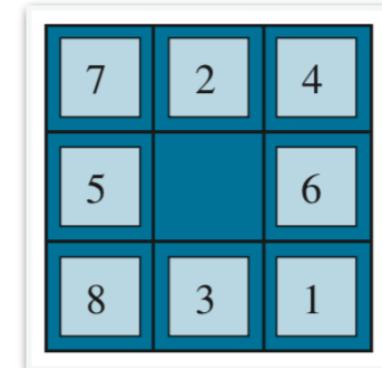
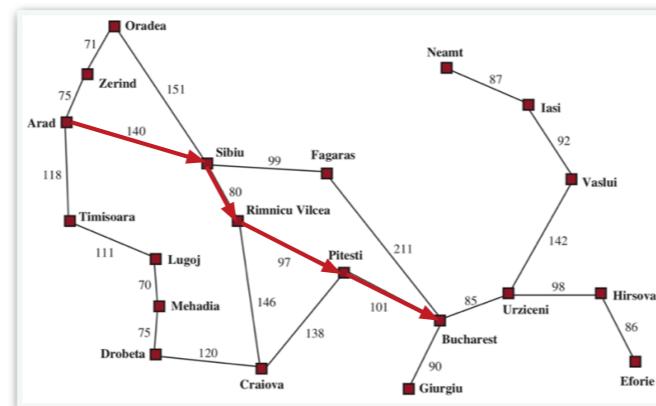
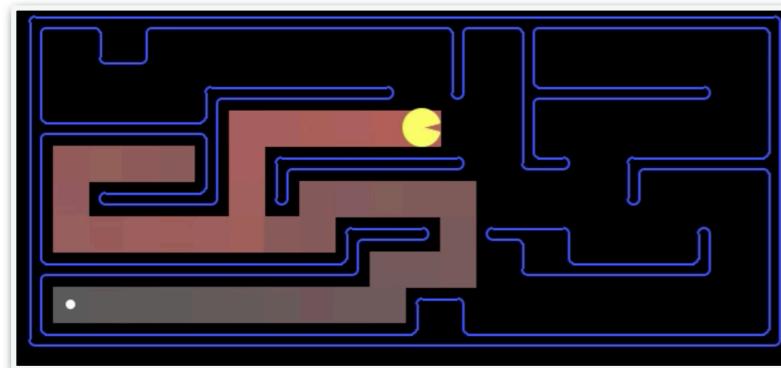
1. Problèmes combinatoires de satisfaction et d'optimisation
2. Concepts et principes fondamentaux de la recherche locale
3. Formalisation de la recherche locale
4. Algorithme du *hill climbing*
5. Difficulté des minima locaux
6. Notion de voisinage connecté
7. Méthodes des redémarrages (*restarts*)
8. Algorithme du recuit simulé (*simulated annealing*)



# Retour sur les modules précédents

## Stratégies de recherche (module 1)

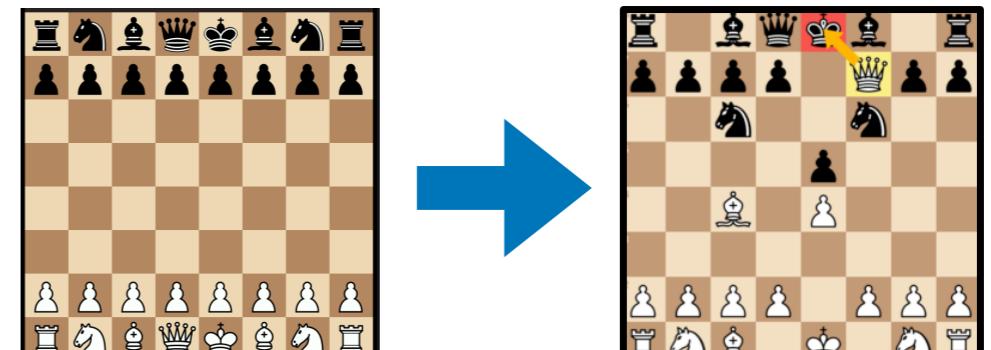
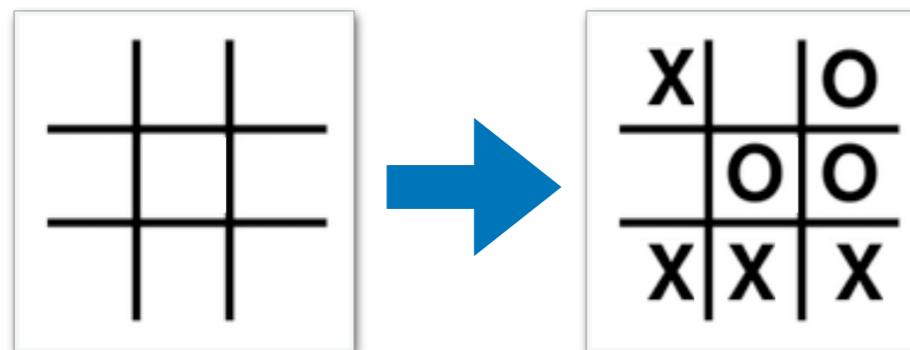
**Objectif:** trouver la meilleure séquence d'actions pour atteindre un état final à partir d'un état initial



**Solution:** une séquence d'actions permettant d'atteindre l'état final

## Recherche adversarielle (module 2)

**Objectif:** trouver les meilleures actions à exécuter dans un environnement compétitif



**Solution:** une politique de sélection d'actions pour atteindre un état final

Dans les deux cas, on souhaite trouver la façon d'atteindre un état final, probablement connu

# Introduction aux problèmes combinatoires



Est-ce que tous les problèmes ont ce même objectif ?

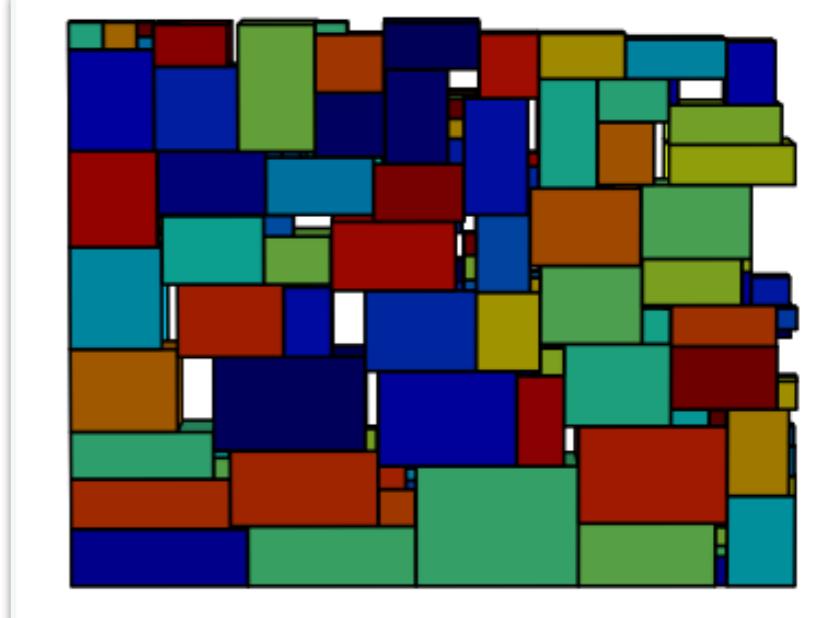
**Non:** il existe certains problèmes pour lesquels on ne cherche pas une séquence d'actions

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

Sudoku

Période	Lundi	Mardi	Mercredi	Jeudi	Vendredi
08h30			INF3405 (01) M-1510		
09h30			INF3405 (01) M-1510	INF8111 (01) C- 631	LOG3430 (02) DISTANCE07
10h30			INF3405 (01) M-1510	INF8111 (01) C- 631	LOG3430 (02) DISTANCE07
11h30				INF8111 (01) C- 631	LOG3430 (02) DISTANCE07
12h45					
13h45				INF8111 (02) L- 4712 Lab. 2 sem. (B2)	
14h45	LOG3430 (02) L-3714 Lab. 2 sem. (B2)			INF8111 (02) L- 4712 Lab. 2 sem. (B2)	INF8215 (01) M-1010
15h45	LOG3430 (02) L-3714 Lab. 2 sem. (B2)		INF8215 (01) L- 3712 Lab. 2 sem. (B1)	INF8111 (02) L- 4712 Lab. 2 sem. (B2)	INF8215 (01) M-1010

Création d'un horaire sans conflit



Entreposage de caisses

**Caractéristiques de ces problèmes:** l'état final nous est inconnu et difficile à trouver

**Difficulté:** l'état final doit respecter plusieurs contraintes

**Solution au problème:** trouver un état qui satisfait toutes les contraintes du problème

**Choix d'une solution:** entre plusieurs solutions faisables, on préfère généralement la moins coûteuse

**Observation fondamentale:** la séquence d'actions pour parvenir à un état final n'est pas importante

Les problèmes de ce type sont connus sous le nom de problèmes combinatoires

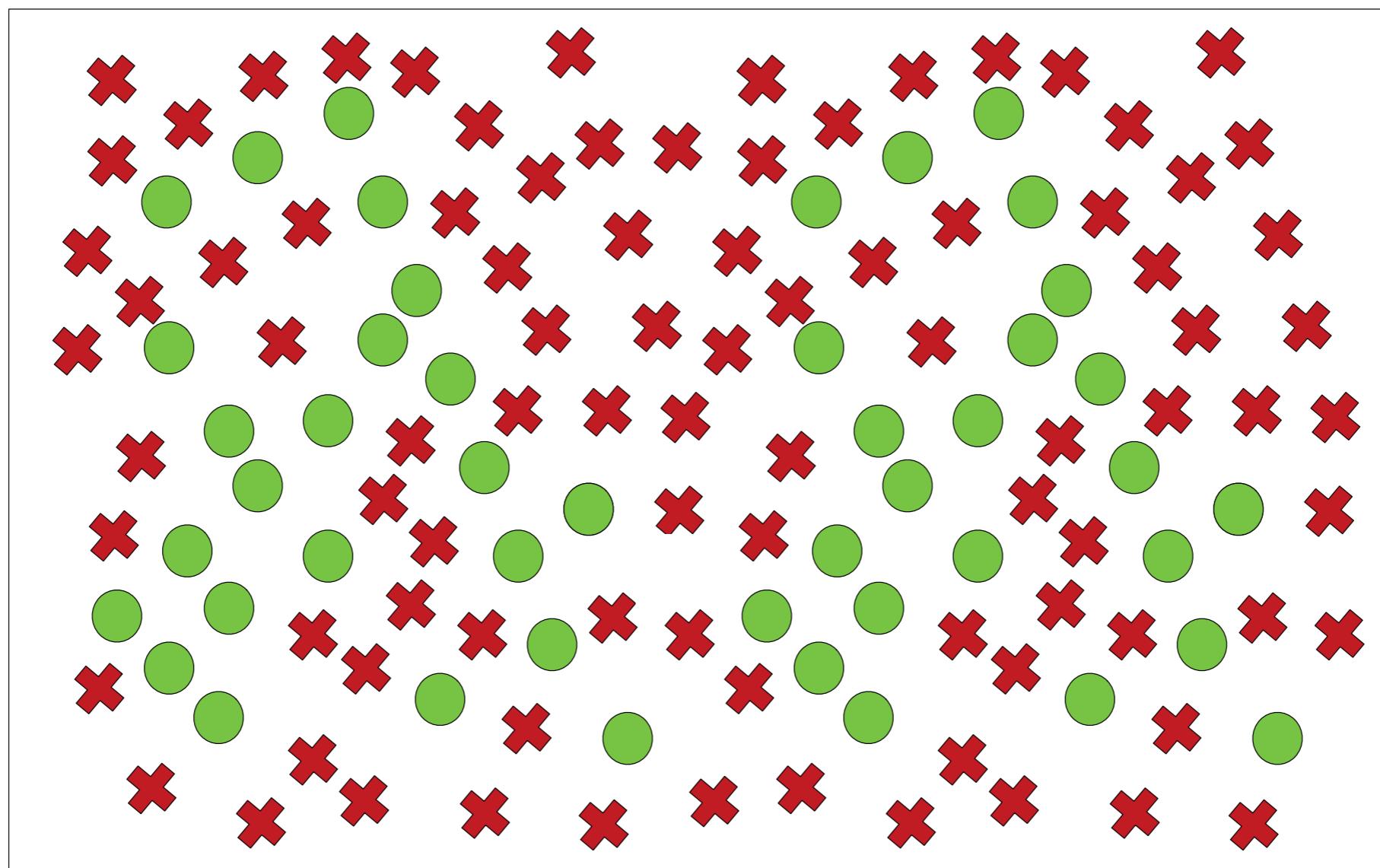
# Problèmes combinatoires de satisfaction

## Problèmes combinatoires de satisfaction (*constraint satisfaction problem - CSP*)

**Objectif:** trouver une solution faisable parmi un ensemble de solutions

**Solutions faisables:** souvent exprimées par le biais de contraintes

**Situation avec aucune solution:** l'objectif est de prouver qu'il n'en existe aucune



*Trouver une aiguille dans une botte de foin*

# Exemples pratiques de CSPs

## Réalisation d'un horaire de maintenance

Objectif: planifier la maintenance d'appareils d'un réseau de distribution d'énergie



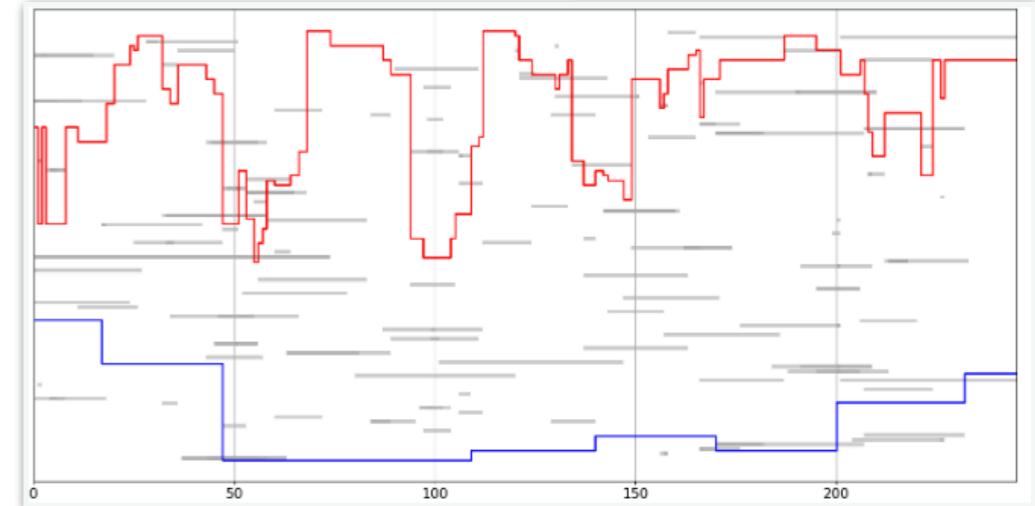
Contraintes:

Finir toutes les maintenance endéans l'année

Pas d'interruption de services dans le réseau

Empêcher la maintenance simultanée de deux appareils

...



## Configuration autorisée

Objectif: acheter un nouvel ordinateur pour un faire tourner un logiciel

Contraintes:

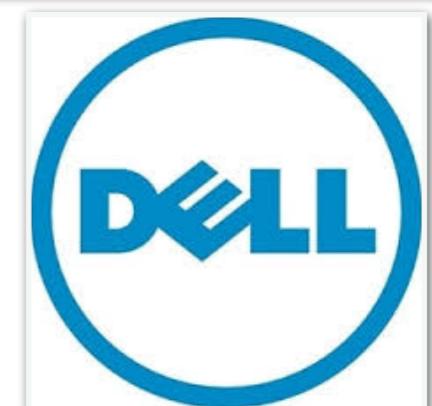
Budget: Max 2000 \$CAD

Mémoire vive: Min 16 Go de RAM

Processeur : Intel Core i7-8700K ou AMD Ryzen 5 3600X

Disque dur: SSD, min 128 Go

...



Cas pratique: utilisé par des fabricants pour donner des produits compatibles avec les besoins d'un client

# Exemples pratiques de CSPs

## Contraintes

Filtrer [Tout supprimer](#)

Intel Core i9  16 Go   
 Intel Core i7 de la gamme K

**Processeur**

- Tout Intel
- Tout AMD
- Intel Core i9
- Intel Core i7 de la gamme K
- Intel Core i7
- Intel Core i5
- AMD Ryzen 9
- Processeur AMD Ryzen™ 7
- Processeur AMD Ryzen™ 5

[Afficher moins](#)

**Mémoire (RAM)**

- 32 Go ou plus
- 16 Go **✓**
- 8 Go

**Shipping**

- Ships Within 5 Days
- Ships Within 10 Days

**Génération de processeur**

- Intel Core de 10e génération

**Classement par la clientèle**

- ★★★★★ 4 étoiles et plus
- ★★★★☆ 3 étoiles et plus
- ★★★☆☆ 2 étoiles et plus
- ★★☆☆☆ 1 étoile et plus

**Taille du disque dur**

## Solutions faisables

5 Résultats Trier par: [Prix le plus bas](#)

Produit	Prix	Économie	Expédition
 Alienware Aurora R11 ★★★★★ 4.4 (4343) 2 699,99 CAD \$ 3 299,99 CAD \$ Économiser 600,00 CAD \$ (18 %) Expédition <a href="#">Gratuit</a>	2 699,99 CAD \$	3 299,99 CAD \$	Économiser 600,00 CAD \$ (18 %) Expédition <a href="#">Gratuit</a>
 Alienware Aurora R11 ★★★★★ 4.4 (4343) 2 999,99 CAD \$ 3 499,99 CAD \$ Économiser 500,00 CAD \$ (14 %) Expédition <a href="#">Gratuit</a>	2 999,99 CAD \$	3 499,99 CAD \$	Économiser 500,00 CAD \$ (14 %) Expédition <a href="#">Gratuit</a>
 Alienware Aurora R11 ★★★★★ 4.4 (4343) 3 549,99 CAD \$ Expédition <a href="#">Gratuit</a>	3 549,99 CAD \$		Expédition <a href="#">Gratuit</a>
 Alienware Aurora R11 ★★★★★ 4.4 (4343) 4 299,99 CAD \$ 4 749,99 CAD \$ Économiser 450,00 CAD \$ (10 %) Expédition <a href="#">Gratuit</a>	4 299,99 CAD \$	4 749,99 CAD \$	Économiser 450,00 CAD \$ (10 %) Expédition <a href="#">Gratuit</a>
 Alienware Aurora R11 ★★★★★ 4.4 (4343) 4 299,99 CAD \$ 4 749,99 CAD \$ Économiser 450,00 CAD \$ (10 %) Expédition <a href="#">Gratuit</a>	4 299,99 CAD \$	4 749,99 CAD \$	Économiser 450,00 CAD \$ (10 %) Expédition <a href="#">Gratuit</a>

**Détails des produits:**

- Alienware Aurora R11 (Intel Core i7 10700KF, 16 Go RAM, RTX 3080 8 Go, 512 Go NVMe, Windows 10 Famille, 2 699,99 CAD \$)
- Alienware Aurora R11 (Intel Core i9 10900f, 16 Go RAM, RTX 3080 10 Go, 512 Go NVMe, Windows 10 Famille, 2 999,99 CAD \$)
- Alienware Aurora R11 (Intel Core i9 10900f, 16 Go RAM, RTX 3080 10 Go, 512 Go NVMe, Windows 10 Famille, 3 549,99 CAD \$)
- Alienware Aurora R11 (Intel Core i9 10900f, 16 Go RAM, RTX 3090 24 Go, 512 Go NVMe, Windows 10 Famille, 4 299,99 CAD \$)
- Alienware Aurora R11 (Intel Core i9 10900f, 16 Go RAM, RTX 3090 24 Go, 512 Go NVMe, Windows 10 Famille, 4 299,99 CAD \$)

**Options et caractéristiques:**

- Couleurs: Gris, Argent
- Voir les offres spéciales
- Financement SFD
- Code de commande: daar11\_s44f
- Afficher les dates de livraison
- [Personnalisez et achetez](#)

<https://www.dell.com/fr-ca/shop/gaming-and-games/sr/game-desktops>

# Problèmes combinatoires d'optimisation

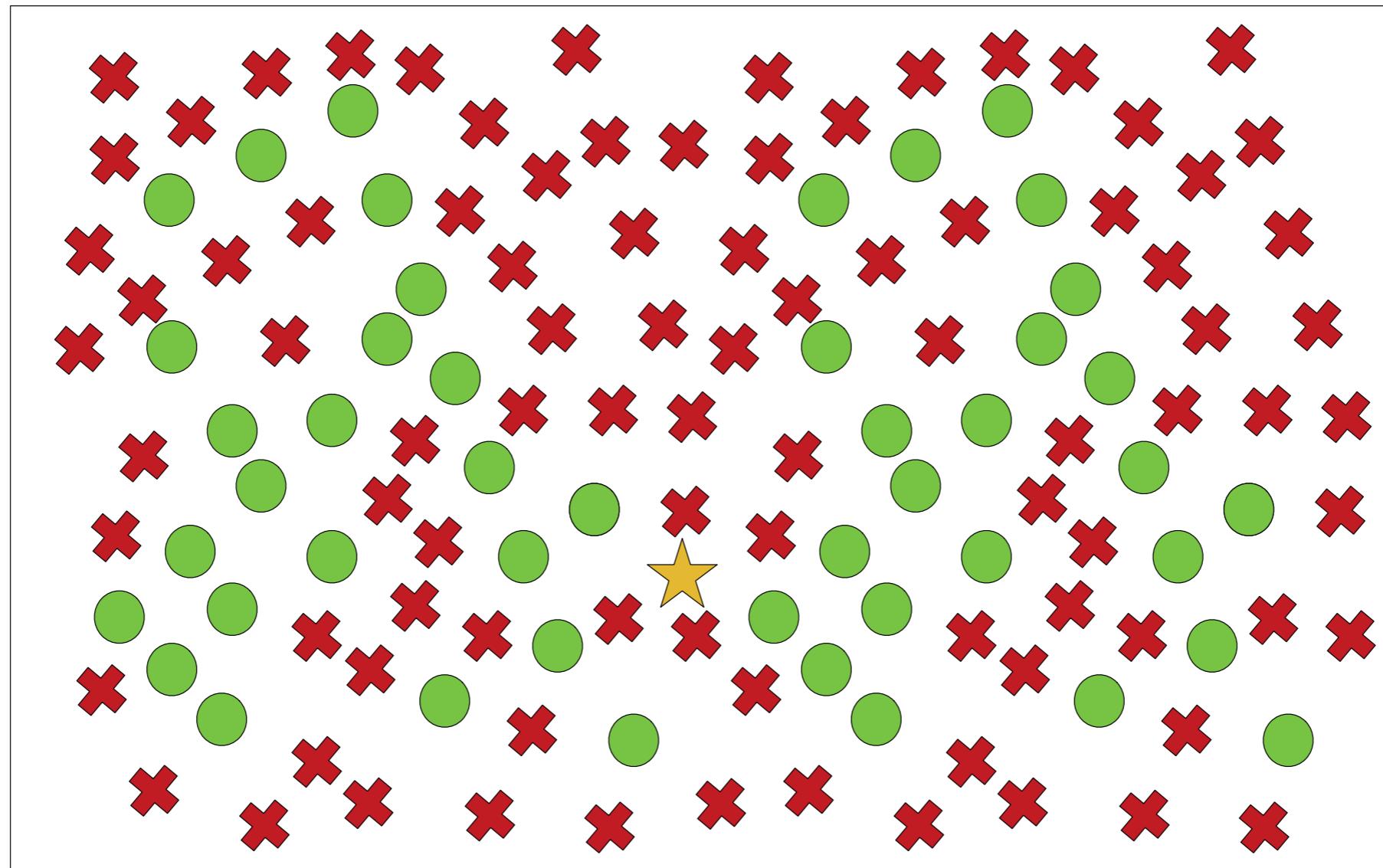
## Problèmes combinatoires d'optimisation (*constraint optimisation problem - COP*)

**Objectif:** trouver la meilleure solution faisable parmi un ensemble de solutions

**Solutions faisables:** souvent exprimées par le biais de contraintes

**Qualité d'une solution:** souvent exprimée par le biais d'une fonction objectif

**Situation avec aucune solution:** l'objectif est de prouver qu'il n'en existe aucune



*Trouver le plus gros lingot d'or dans une botte de foin*

# Exemple pratique de COPs

## Investissement financier

**Objectif:** choisir un ensemble d'investissements maximisant le revenu espéré étant donné un budget limité

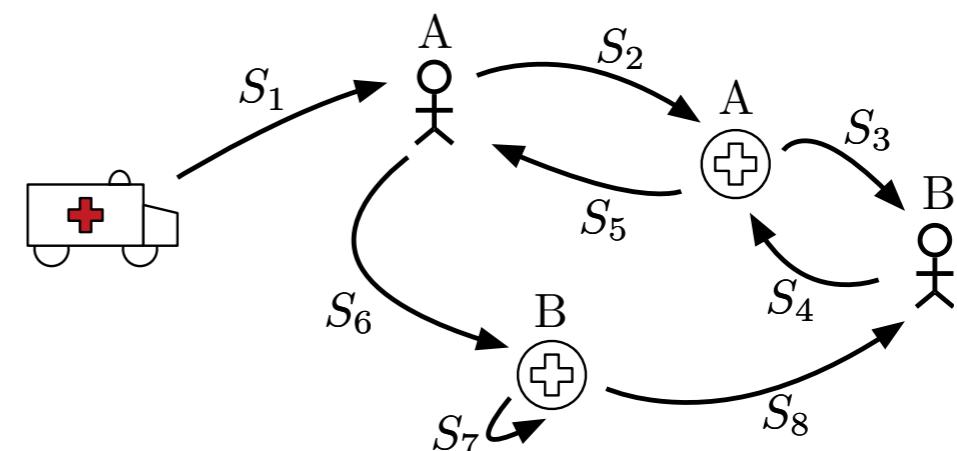
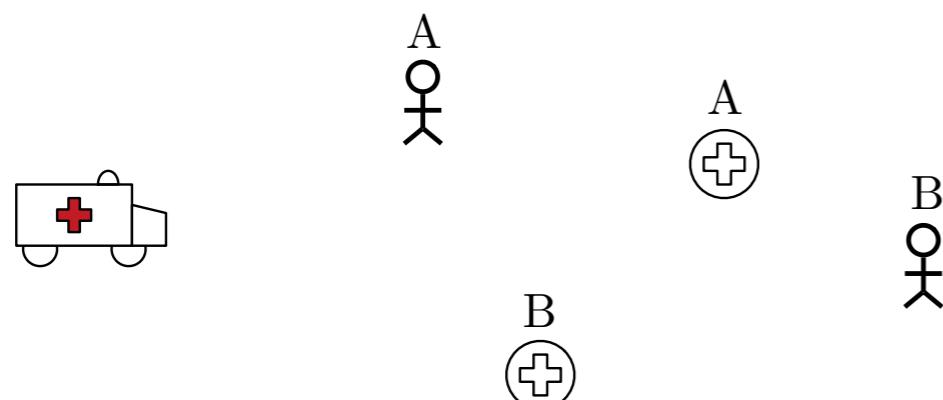
Investissement	Coût (\$)	Revenu espéré dans 10 ans (\$)
A	200	1 000
B	200	1 000
C	200	1 000
D	500	10 000
E	500	10 000
F	800	13 000
G	300	7 000

**Contrainte:** budget maximal de 1000 \$

## Problèmes de transport

**Objectif:** amener des patients à l'hôpital et les ramener après leur soin (minimiser distance des ambulances)

**Contraintes:** capacité des véhicules, respect des horaires des soins



# Intérêt pratique de résoudre ces problèmes

## Exemple: impression d'affiches

**Contexte:** un imprimeur utilise une grande toile d'un format fixe pour imprimer des commandes de photos

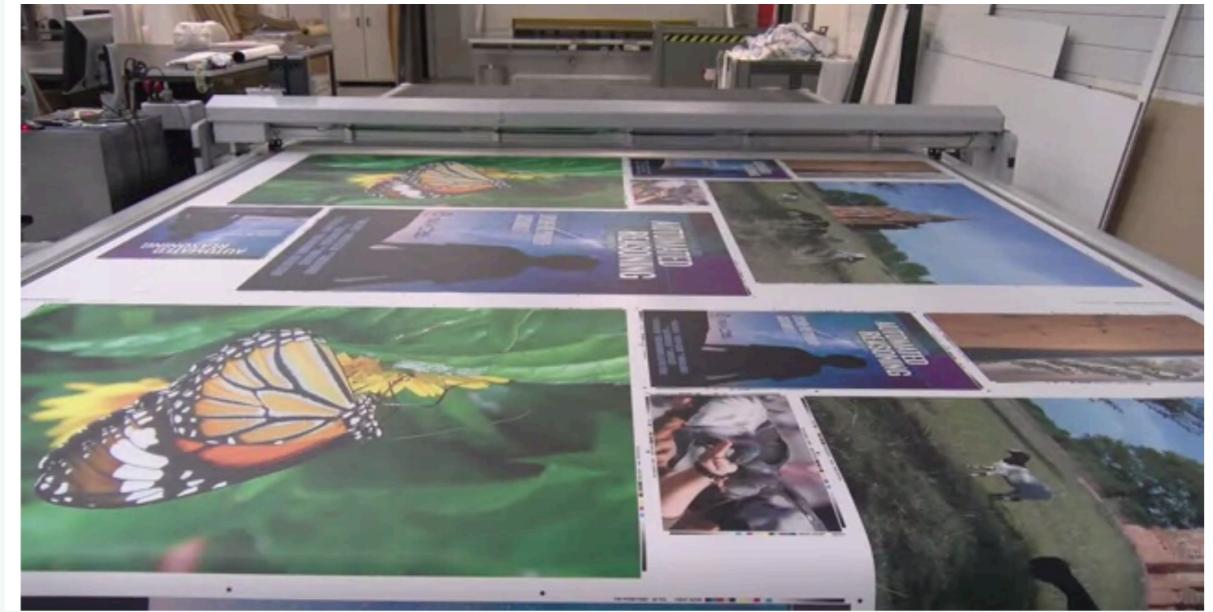
**Objectif:** minimiser la surface non-utilisée par les photos

**Contraintes:** pas de chevauchements entre les affiches

**Economies** :  $1m^2$  de papier spécial par impression

**Nombre d'opérations** : 60 par jour

**Economies annuelles** :  $21900m^2$  de papier



<https://www.coursera.org/lecture/automated-reasoning-sat/general-introduction-and-an-application-to-poster-printing-I7QIO>

## Importance en industrie

**Contexte industriel:** ce genre de problème arrive énormément (formalisation très générale)

**Raison:** beaucoup d'opérations doivent être effectuées un grand nombre de fois

**Exemple:** production de composants, transport de biens, stockage, etc.

**Intérêt pratique:** effectuer efficacement ces opérations entraîne des gains significatifs sur le long terme

**Opportunité pour vous:** pouvoir résoudre efficacement ces problèmes est une compétence très demandée !

**Offre à Polytechnique:** plusieurs cours ne sont dédiés qu'à la résolution de ce type de problèmes

# Difficulté des problèmes combinatoires

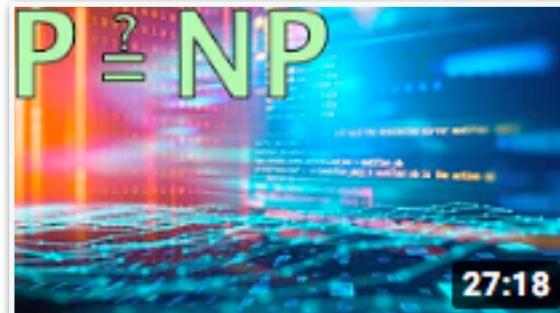


Est-ce que ces problèmes sont difficiles à résoudre ?

**Difficulté intrinsèque:** ces problèmes sont pour la plupart **NP-complets** ou **NP-difficiles**

**Explication:** on ne connaît pas d'algorithmes à complexité temporelle polynomiale pour les résoudre

**Conséquence:** le temps d'exécution devient vite irraisonnable (croissance exponentielle)



[https://www.youtube.com/  
watch?v=AgtOCNCejQ8](https://www.youtube.com/watch?v=AgtOCNCejQ8)

Une étude des classes de complexité est en dehors du cadre de notre cours

**Science étonnante:** vidéo introductory très bien faite à ce sujet

**Autre ressource:** le cours INF6102 que je donne à l'hiver



J'ai accès à un super-calculateur, est-ce vraiment irréaliste de tout tester ?

**Principe:** essayer une solution, évaluer le résultat, et répéter jusqu'à ce qu'une bonne solution soit trouvée

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

9 possibilités par case

→  $9^{51} > 10^{47}$  possibilités

51 cases à compléter

$$\frac{10^{47} \text{ tests}}{1000 \times 10^9 \text{ tests/seconde}} = 4.63 \times 10^{44} \text{ secondes} \approx 5.35 \times 10^{39} \text{ années}$$

Clairement pas réalisable d'utiliser cette stratégie dans cette situation !

# Limitations de la recherche exhaustive simple

## Résolution par recherche exhaustive (*bruteforce*)

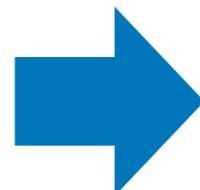
**Principe:** essayer une solution, évaluer le résultat, et répéter jusqu'à ce qu'une bonne solution soit trouvée

**Utilité:** résolution envisageable pour les problèmes ayant un petit espace de recherche (peu de décisions)

**Difficulté:** la méthode atteint vite ses limites (conséquence d'une complexité exponentielle)

**Exemple:** le problème de l'investissement implique des décisions binaires (sélection ou non)

Investissement	Coût (\$)	Revenu espéré dans 10 ans (\$)
A	200	1 000
B	200	1 000
C	200	1 000
D	500	10 000
E	500	10 000
F	800	13 000
G	300	7 000

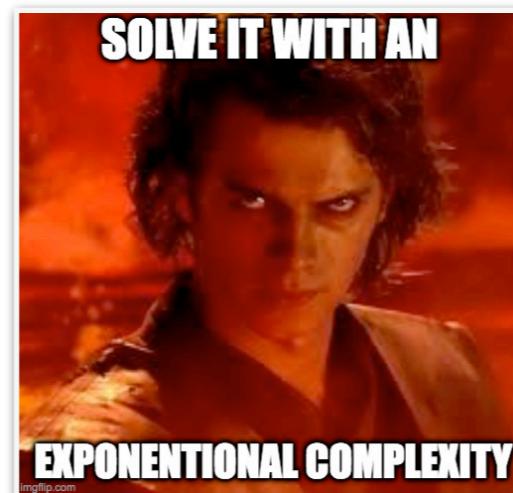


**2 investissements:** 4 possibilités

**3 investissements:** 8 possibilités

**4 investissements:** 16 possibilités

**$n$  investissements:**  $2^n$  possibilités



**Option 1:** ne pas avoir peur d'un algorithme à complexité exponentielle dans le pire des cas

**Option 2:** se contenter d'un algorithme approximatif (non-exhaustif), et s'arrêter si une solution est trouvée

# Option 1 - méthodes de résolution exhaustive intelligente



## Recherche complète (exhaustive)

Stratégie de recherche qui s'arrête lorsqu'on a la certitude que la solution trouvée est optimale

**Principe 1:** ne pas avoir peur d'un algorithme à complexité exponentielle dans le pire des cas

**Principe 2:** rajouter des mécanismes afin d'accélérer le processus de résolution

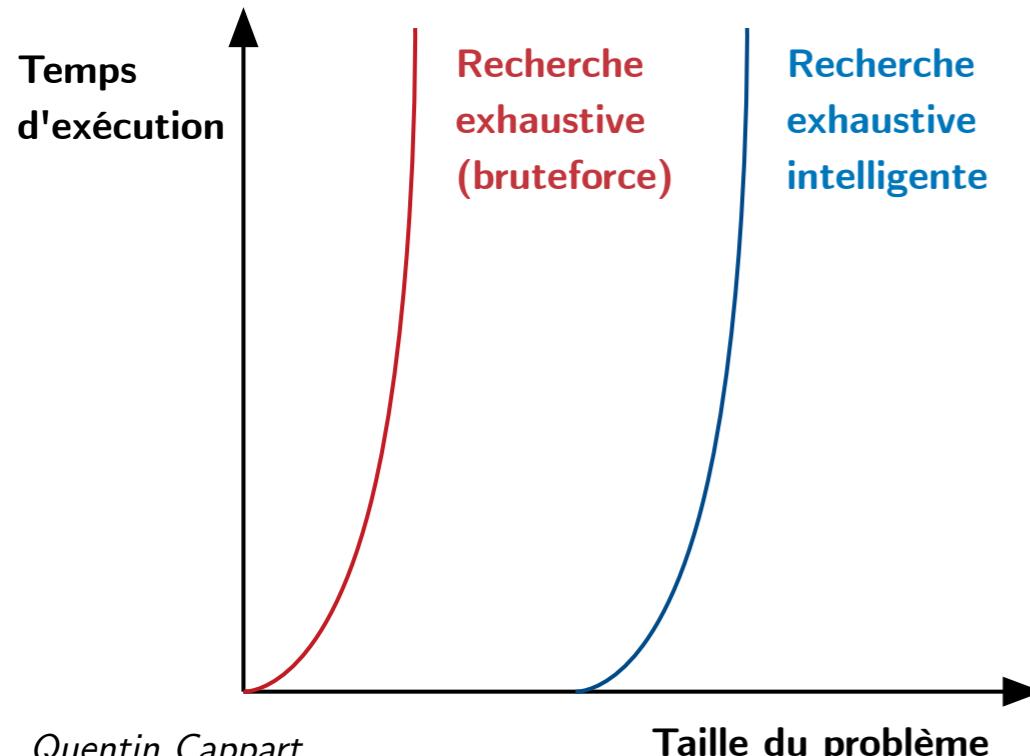
On a ainsi toujours une recherche exhaustive... mais qui intègre une forme d'intelligence



Quels mécanismes peut-on intégrer à une recherche exhaustive ?

**Utilisation d'heuristique:** essayer d'abord les solutions les plus prometteuses

**Intégration de raisonnements logiques:** supprimer les solutions que l'on sait mauvaises à coup sûr



**Difficulté:** on a toujours une complexité exponentielle

**Conséquence:** la taille deviendra toujours problématique

Exemples d'algorithmes de résolution de ce type

Programmation en nombres entiers (MAGI - MTH6404)

Programmation par contraintes (module suivant)

Résolution SAT (module sur les agents logiques)

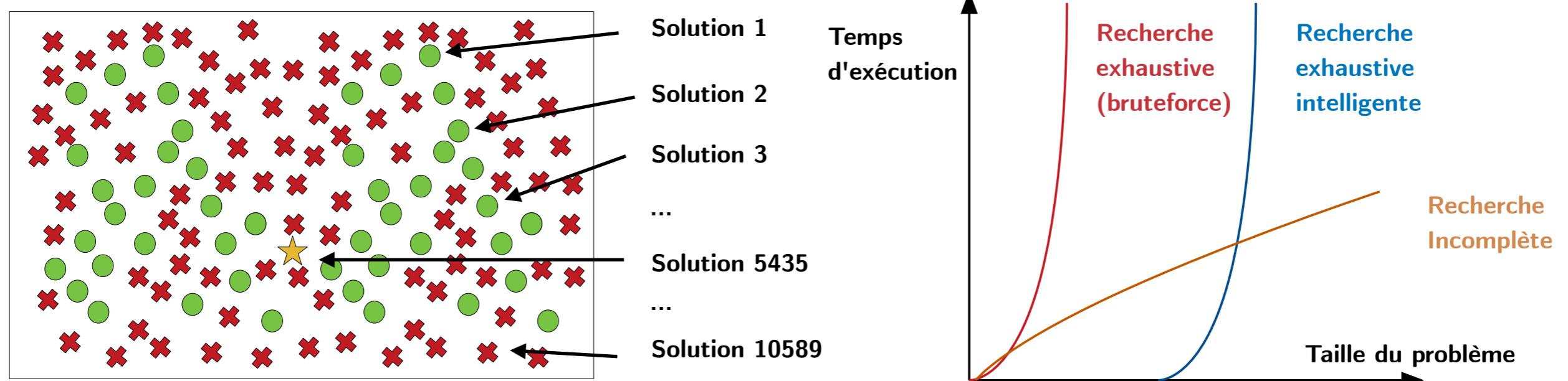
# Option 2 - méthodes de résolution non exhaustive (recherche incomplète)



Note: le critère d'arrêt est autre qu'une exploration complète

**Principe 1:** abandonner l'exhaustivité de la recherche pour améliorer les performances

**Principe 2:** explorer un sous-ensemble des solutions, et prendre la meilleure trouvée actuellement



**Avantage:** processus de résolution beaucoup plus efficace au niveau du temps d'exécution

**Avantage:** fonctionne très bien pour résoudre de très grands problèmes

**Inconvénient:** on perd la garantie de trouver la meilleure solution (l'espace non entièrement exploré)

**Inconvénient:** impossibilité de prouver qu'un problème est infaisable

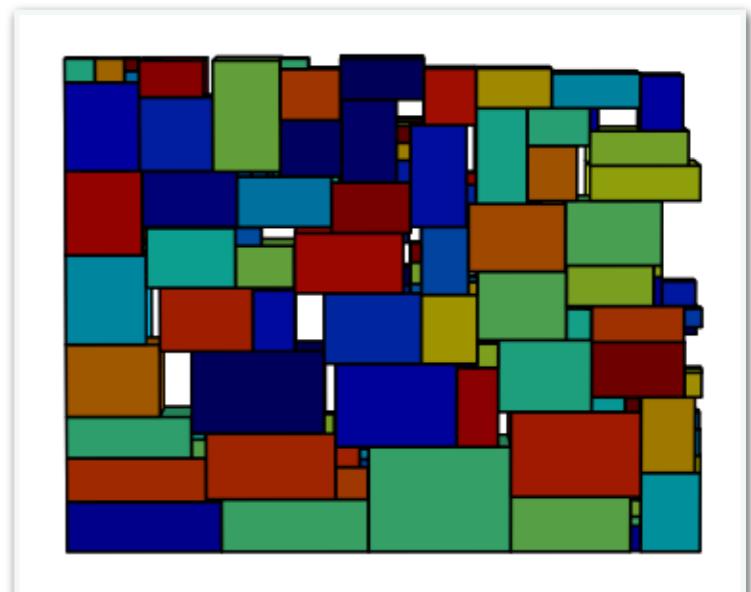
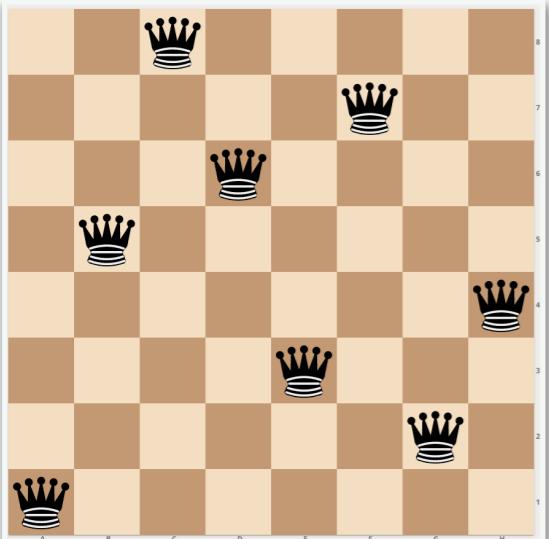
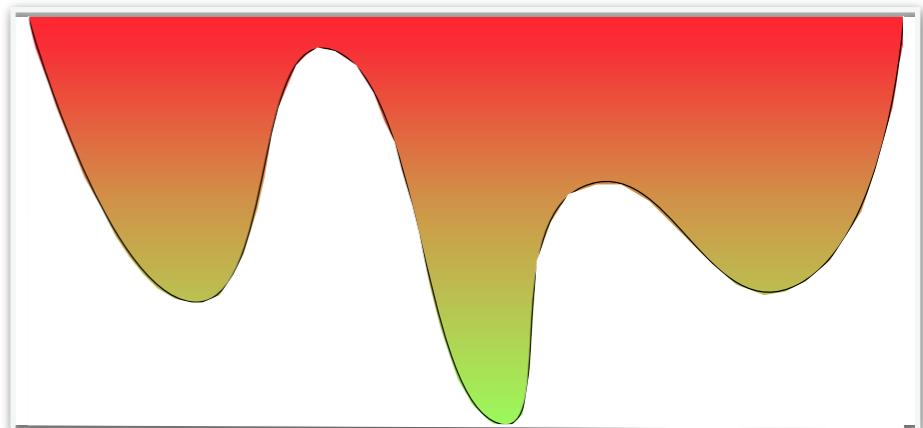
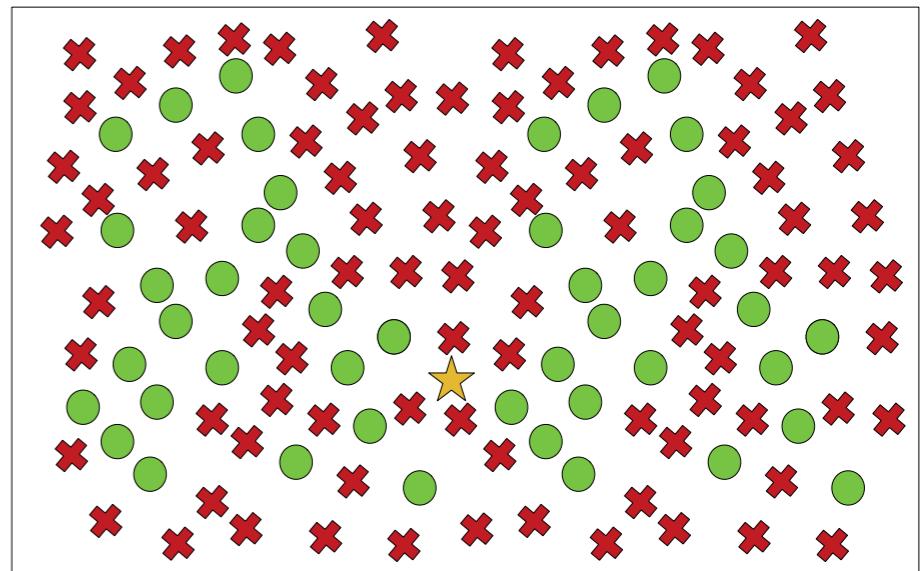
**Recherche locale:** méthode de résolution incomplète par excellence (ce module)

**Métaheuristiques:** améliorations indispensables à une recherche locale (INF6102)

# Table des matières

## Recherche locale

- ✓ 1. Problèmes combinatoires de satisfaction et d'optimisation
- 2. Concepts et principes fondamentaux de la recherche locale
- 3. Formalisation de la recherche locale
- 4. Algorithme du *hill climbing*
- 5. Difficulté des minima locaux
- 6. Notion de voisinage connecté
- 7. Méthodes des redémarrages (*restarts*)
- 8. Algorithme du recuit simulé (*simulated annealing*)



# Recherche locale - exemple introductif

## Exemple: les quatre reines de Westeros

Objectif: placer les reines sur une grille sans qu'elles ne s'attaquent mutuellement




## Résolution du problème par recherche locale

### (1) Initialisation

Chaque reine est placée aléatoirement sur une colonne

Solution initiale, qui est infaisable

### (2) Amélioration de la solution

On bouge la reine ayant le plus de conflits...

... sur la case de sa colonne réduisant le plus ses conflits

On répète jusqu'à qu'on ne sache plus améliorer la solution

### Taille du problème

Total :  $4 \times 4 \times 4 \times 4 = 256$  configurations possibles

Résultat: résolution en seulement 4 étapes

# Principes de la recherche locale

## Procédure simplifiée de recherche locale

**Etape 1:** on démarre d'une solution initiale

**Etape 2:** on se déplace de solutions en solutions en effectuant des mouvements locaux

**Etape 3:** on arrête de se déplacer une fois qu'un critère d'arrêt est atteint

Les mouvements locaux possibles forment un voisinage

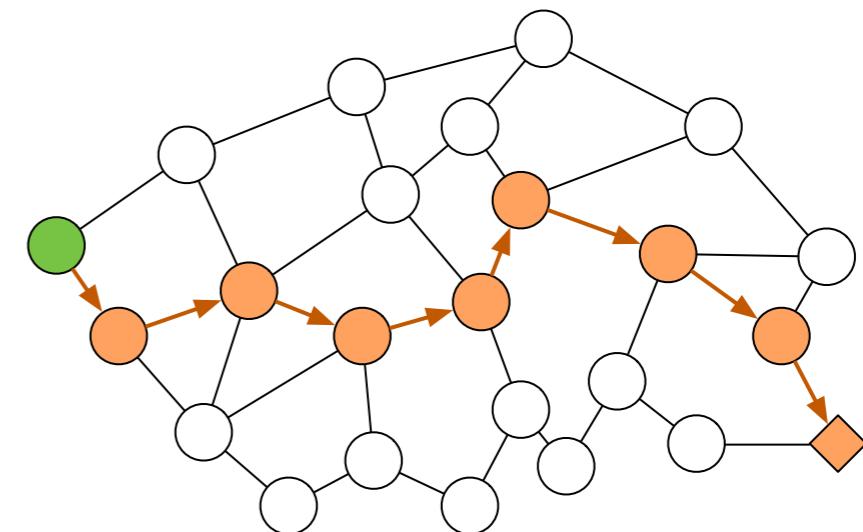
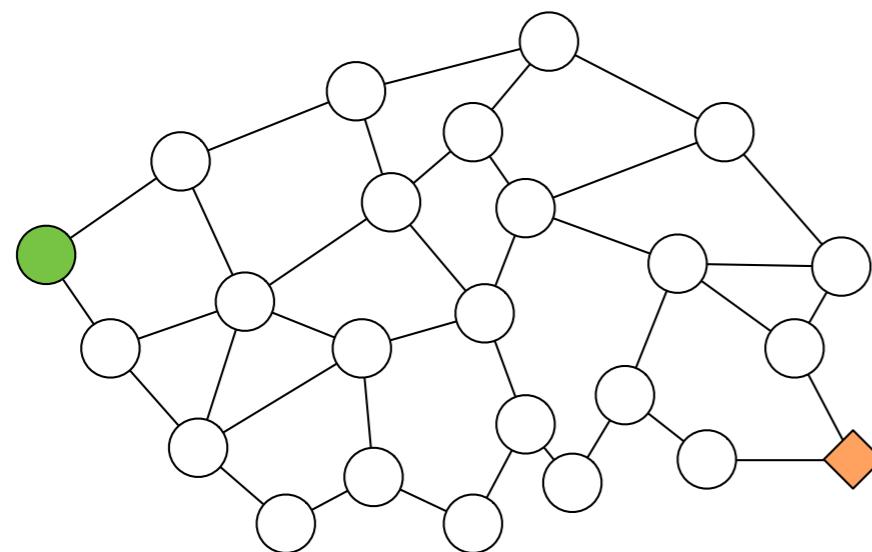
**Voisinage:** ensemble des mouvements locaux pouvant être fait à partir de notre solution actuelle

## Intuition de la procédure

**Objectif:** on veut se déplacer au long terme sur la meilleure solution possible

La recherche locale améliore la solution en ne considérant qu'un sous-ensemble d'autres solutions

La recherche correspond à une exploration dans un graphe où chaque noeud est une solution candidate



Notez bien qu'une solution est un état, et non une séquence d'actions (module 1)

# Recherche locale - formalisation préliminaire

## Formalisation

$S$  : l'ensemble des solutions possibles

$s \in S$  : une solution spécifique

$N : S \rightarrow 2^S$  : une fonction de voisinage

$N(s)$  : le voisinage de  $s$

$f : S \rightarrow \mathbb{R}$  : une fonction d'évaluation

$f(s)$  : la valeur de la solution  $s$

## Problème des quatre reines

### Représentation d'une solution

Vecteur de 4 éléments, donnant la ligne de chaque reine

Ensemble des solutions: les solutions pouvant être générées

$S = \{\langle 1,1,1,1 \rangle, \langle 2,1,1,1 \rangle, \dots, \langle 4,4,4,4 \rangle\}$

Solution initiale: solution où démarre la recherche

$s = \langle 1,2,1,4 \rangle$

Voisinage: bouger une seule reine sur sa colonne

$N(\langle 1,2,1,4 \rangle) = \{\langle 2,2,1,4 \rangle, \langle 3,2,1,4 \rangle, \langle 4,2,1,4 \rangle, \langle 1,1,1,4 \rangle, \dots\}$

Fonction d'évaluation: nombre de conflits de la solution

$f(\langle 1,2,1,4 \rangle) = 10$

1			
2			
3			
4			



# Recherche locale - Algorithme préliminaire

## Pseudo-code d'une recherche locale

```
generate an initial solution  $s$   
 $G = [n \in N(s) \text{ such that } f(n) < f(s)]$   
while  $|G| > 0$  :  
    select a new solution  $s$  from  $G$   
     $G = [n \in N(s) \text{ such that } f(n) < f(s)]$   
return  $s$ 
```



## Questions à résoudre

Question 1: comment choisir une solution initiale ?

Question 2: comment définir un voisinage ?

Question 3: comment définir une fonction d'évaluation ?

Question 4: comment choisir un nouvel état dans notre voisinage ?

Question 5: est-ce que cet algorithme donne la meilleure solution ?

Différents choix existent, et donnent lieu à différents algorithmes

Ces choix dépendent aussi du problème considéré



# Recherche locale - formalisation



## Fonction de voisinage

Fonction qui indique les solutions pouvant être atteintes à partir d'une solution spécifique

$$N(s) : S \rightarrow 2^S$$



## Fonction de validité

Fonction qui indique quels voisins d'une solution sont valides pour une sélection

$$L(N(s), s) : 2^S \times S \rightarrow 2^S$$



## Fonction de sélection

Fonction qui sélectionne un voisin d'une solution parmi ceux éligibles dans le voisinage

$$Q(L(N(s), s), s) : 2^S \times S \rightarrow S$$



## Fonction d'évaluation

Fonction qui donne une valeur sur la qualité d'une solution

$$f : S \rightarrow \mathbb{R}$$

# Recherche locale - Algorithme



## Critère d'arrêt

Fonction qui définit quand la recherche doit se terminer  $\Theta$

**Cas standard:** une limite sur le nombre d'itérations ou le temps d'exécution



## Algorithme de recherche locale

LocalSearch( $N, L, Q, f, \Theta$ ) :

$s = \text{generateInitialSolution}()$

$s^* = s$

    for  $k \in 1$  to  $\Theta$  :

$G = [n \in N(s)]$

$V = [n \in L(G, s)]$

$s = Q(V, s)$

        if  $f(s) < f(s^*)$  :

$s^* = s$

    return  $s^*$

**Entrées:** fonction de voisinage, validité, sélection, évaluation, et d'arrêt

**Génération d'une solution initiale**

**On retient la meilleure solution trouvée actuellement**

**On itère jusqu'à ce qu'un critère d'arrêt soit atteint**

**Définition du voisinage**

**Définition des voisins valides (on filtre les voisins pour ne garder que ceux valides)**

**Sélection d'un voisin**

**Mise à jour de la meilleure solution trouvée (minimisation)**

**Valeur de retour:** la meilleure solution trouvée



## Heuristique de recherche locale

Conception de ces fonctions pour effectuer la recherche

# Recherche locale - Algorithme

## Questions à résoudre

**Question 1:** comment choisir une solution initiale ?

**Question 2:** comment définir ces fonctions ?

**Question 3:** est-ce que cet algorithme nous donne la meilleure solution ?

## Problème des quatre reines

**Fonction de voisinage:** bouger une reine sur sa colonne

$$N(\langle 1,2,1,4 \rangle) : [\langle 2,2,1,4 \rangle, \langle 3,2,1,4 \rangle, \langle 4,2,1,4 \rangle, \langle 1,1,1,4 \rangle, \dots]$$

**Fonction d'évaluation:** compter le nombre de conflits d'une solution

$$f(s) : \# \text{ conflicts in } s$$

**Fonction de validité:** uniquement les voisins réduisant le nombre de conflits

$$L(N(s), s) : [n \in N(s) \text{ such that } f(n) < f(s)]$$

**Fonction de sélection:** prendre le voisin réduisant le plus les conflits

$$Q(L(N(s), s), s) : \operatorname{argmin}_{f(s)}(L(N(s), s))$$

**Remarque:** il ne s'agit qu'un modèle parmi d'autres

```
LocalSearch( $N, L, Q, f, \Theta$ ) :  
     $s = \text{generateInitialSolution}()$   
     $s^* = s$   
    for  $k \in 1$  to  $\Theta$  :  
         $G = [n \in N(s)]$   
         $V = [n \in L(G, s)]$   
         $s = Q(V, s)$   
        if  $f(s) < f(s^*)$  :  
             $s^* = s$   
    return  $s^*$ 
```


# Algorithme du Hill climbing

## Intérêt de notre formalisme

Avantage: facilite la construction d'algorithmes

Avantage: défini sans ambiguïté le fonctionnement dans la recherche

Cela permet ainsi différentes constructions basées sur le même noyau



 **Algorithme du *Hill climbing***

Recherche locale consistant à choisir le meilleur voisin à chaque itération

Fonction de validité: tous les voisins qui améliorent la solution actuelle

$$L(N(s), s) : \left[ n \in N(s) \text{ such that } f(n) < f(s) \right]$$

Fonction de sélection: le meilleur parmi tous ces voisins

$$H : \left[ n \in L(N(s), s) \text{ such that } f(n) = \min_{k \in L(N(s), s)} f(k) \right]$$

En cas d'égalité: on choisit un voisin aléatoirement parmi les meilleurs

$$Q(L(N(s), s), s) : k \sim H \text{ with probability } \frac{1}{|H|}$$

Les fonctions de voisinage et d'évaluation dépendent du problème

```
LocalSearch( $N, L, Q, f, \Theta$ ) :  
     $s = \text{generateInitialSolution}()$   
     $s^* = s$   
    for  $k \in 1$  to  $\Theta$  :  
         $G = [n \in N(s)]$   
         $V = [n \in L(G, s)]$   
         $s = Q(V, s)$   
        if  $f(s) < f(s^*)$  :  
             $s^* = s$   
    return  $s^*$ 
```



A-t-on d'autres choix pour la fonction de sélection ?

# Problème des reines: fonctions de sélection

## Fonction de voisinage considérée

**Principe:** bouger une reine sur une case de sa colonne

**Taille du voisinage:**  $n$  reines donne un voisinage de  $n^2$  voisins

## Fonction de sélection 1: meilleur voisin

**Principe:** sélectionner le meilleur voisin selon la fonction d'évaluation

**Avantage:** retourne le meilleur voisin



?

Quelle est la complexité de cette sélection ?

**Complexité:**  $O(n^2)$  (demande de considérer toutes les paires variables/valeurs)

**Inconvénient:** requiert l'évaluation de tout le voisinage, ce qui peut être coûteux

## Fonction de sélection 2: premier voisin améliorant

**Principe:** sélectionner le premier voisin qui améliore la solution

**Avantage évident:** ne nécessite pas d'évaluer systématiquement tout le voisinage

$Q(L(N(s), s), s)$  : first  $k \in L(N(s), s)$

?

Quelle est la complexité de cette sélection ?

**Complexité:**  $O(n^2)$  (demande de considérer toutes les paires variables/valeurs dans le pire des cas)

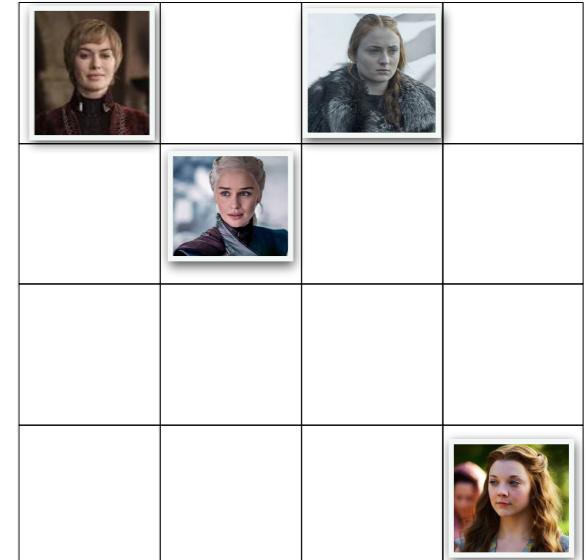
**Inconvénient:** amène à une sélection sous optimale

**En pratique:** la complexité moyenne est nettement meilleure, malgré la complexité quadratique

# Problème des reines: fonctions de sélection

**Limitation actuelle:** on doit considérer un voisinage entier dans le pire des cas

**Amélioration possible:** faire la sélection **sur un sous-ensemble du voisinage**



## Fonction de sélection 3: *max/min-conflicts*

**Etape 1:** on prend la reine ayant le plus de conflits (**max**)

**Etape 2:** on prend la ligne donnant le moins de conflits (**min**)

**?** Quelle est la complexité de cette sélection ?

Etape 1:  $n$  opérations pour trouver le max

Etape 2:  $n$  opérations pour trouver le min

**Complexité:**  $\mathcal{O}(n + n) = \mathcal{O}(n)$

**Avantage:** on garde l'idée qu'on souhaite trouver un bon voisin qui améliore, à plus faible coût

## Fonction de sélection 4: *min-conflicts*

**Etape 1:** on prend une reine aléatoire

**Etape 2:** on prend la ligne donnant le moins de conflits (**min**)

**?** Quelle est la complexité de cette sélection ?

Etape 1: nombre constant d'opérations pour le choix aléatoire

Etape 2:  $n$  opérations pour trouver le min

**Complexité:**  $\mathcal{O}(1 + n) = \mathcal{O}(n)$

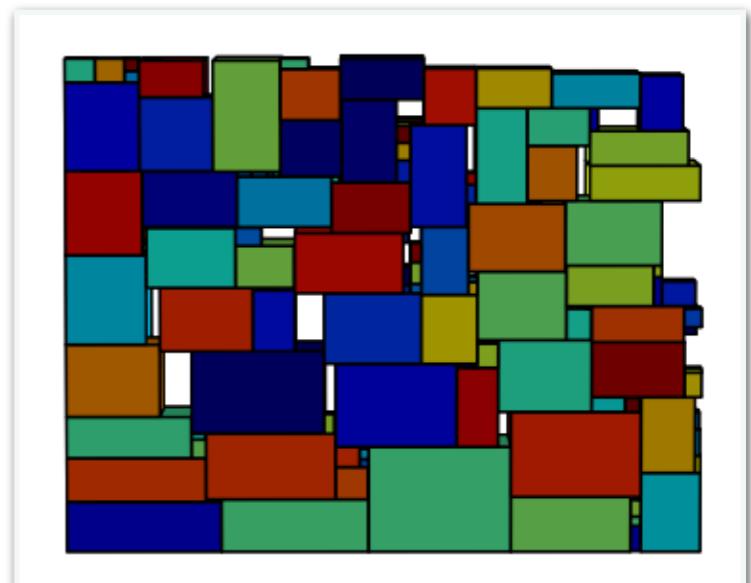
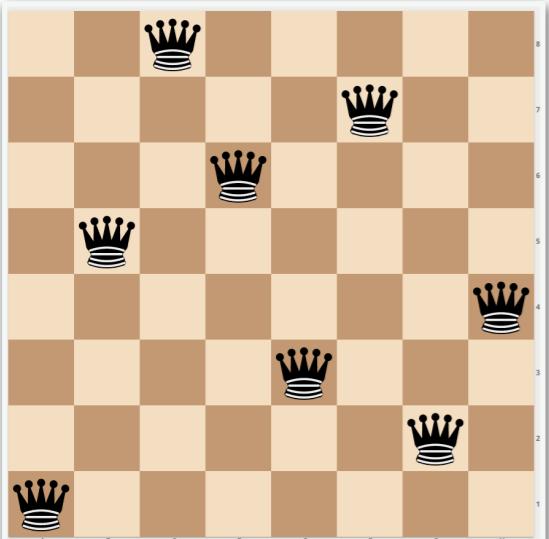
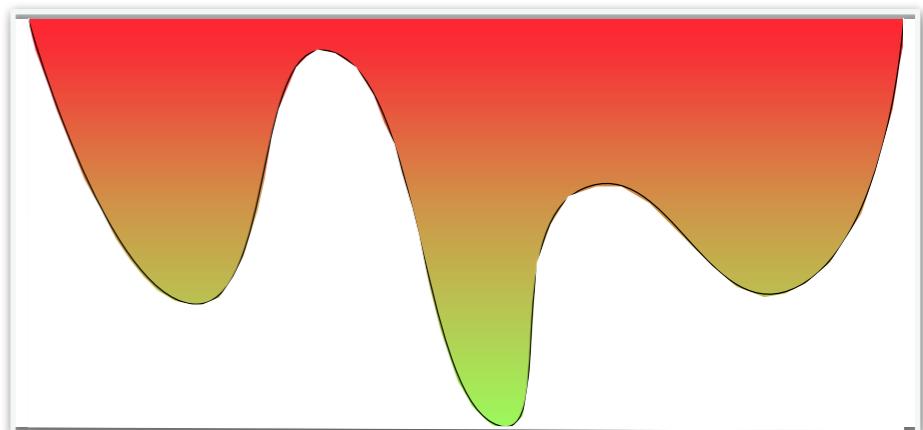
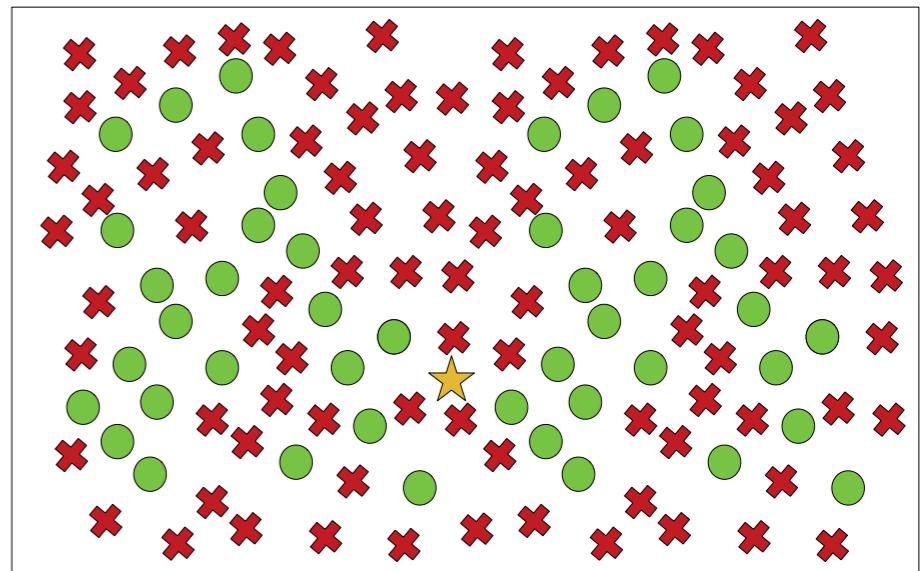
**Avantage supplémentaire:** favorise la diversification (plus à venir là dessus)

**Remarque:** ces quatre fonctions offrent plusieurs compromis entre qualité et rapidité des mouvements

# Table des matières

## Recherche locale

- ✓ 1. Problèmes combinatoires de satisfaction et d'optimisation
- ✓ 2. Concepts et principes fondamentaux de la recherche locale
- ✓ 3. Formalisation de la recherche locale
- ✓ 4. Algorithme du *hill climbing*
- 5. Difficulté des minima locaux
- 6. Notion de voisinage connecté
- 7. Méthodes des redémarrages (*restarts*)
- 8. Algorithme du recuit simulé (*simulated annealing*)



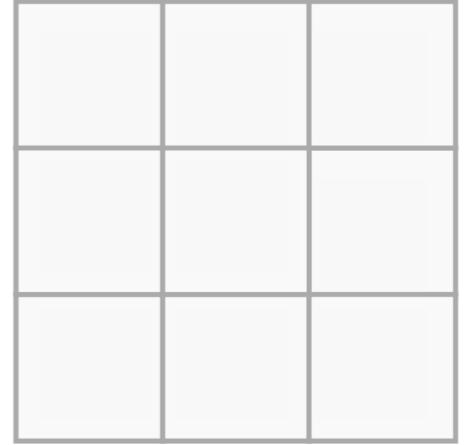
# Problème combinatoire de satisfaction (CSP) - Carré magique

## Carré magique

**Objectif:** remplir un carré  $3 \times 3$  ( $n \times n$ ) avec les nombres de 1 à 9 ( $n^2$ )

**Contraintes:**

- (1) Chaque case doit avoir un nombre différent
- (2) La somme de chaque colonne, rangée, et diagonale doit valoir 15 ( $T$ )



## Modèle de recherche locale

Modéliser une résolution en recherche locale revient à définir différents éléments

**Element 1:** l'ensemble des solutions (espace de recherche)

**Element 2:** la solution initiale (point de départ de la recherche)

**Element 3:** la fonction d'évaluation (qualité d'une solution)

**Element 4:** la fonction de voisinage (génération de nouvelles solutions)

**Element 5:** la fonction de validité (filtrage des solutions non voulues)

**Element 6:** la fonction de sélection (déplacement sur une nouvelle solution)

**Etape suivante:** intégrer ces éléments dans un algorithme de résolution

**Notre modélisation va nous permettre d'introduire de nouveaux concepts**

```
LocalSearch( $N, L, Q, f, \Theta$ ) :  
     $s = \text{generateInitialSolution}()$   
     $s^\star = s$   
    for  $k \in 1$  to  $\Theta$  :  
         $G = [n \in N(s)]$   
         $V = [n \in L(G, s)]$   
         $s = Q(V, s)$   
        if  $f(s) < f(s^\star)$  :  
             $s^\star = s$   
    return  $s^\star$ 
```

# Carré magique - modélisation par recherche locale

## Ensemble des solutions

Définition: toutes les solutions que l'on permet de créer

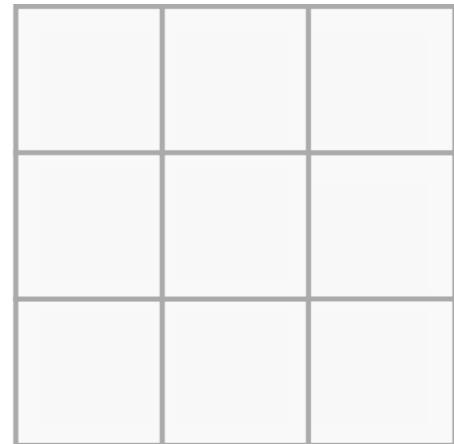
Plusieurs choix de conception sont possibles

Choix 1: chaque case a un nombre entre 1 et 9

Choix 2: les nombres 1 à 9 sont répartis dans la grille

$9^9 = 387420489$  solutions

$9! = 362880$  solutions



Quel choix vous paraît préférable ?

Choix 2: génère un espace environ 1000 fois plus petit



"Les petits espaces de recherche,  
tu préfèreras"

La définition de l'espace de recherche dépend de la façon dont les contraintes sont considérées



Contrainte dure (*hard constraint*)

Contrainte du problème restant toujours satisfaite dans chaque solution de l'espace



Contrainte molle (*soft constraint*)

Contrainte du problème pouvant être non-satisfait dans une solution de l'espace

Choix 1: les deux contraintes sont molles

Choix 2: la contrainte indiquant que les nombres doivent être différents est dure

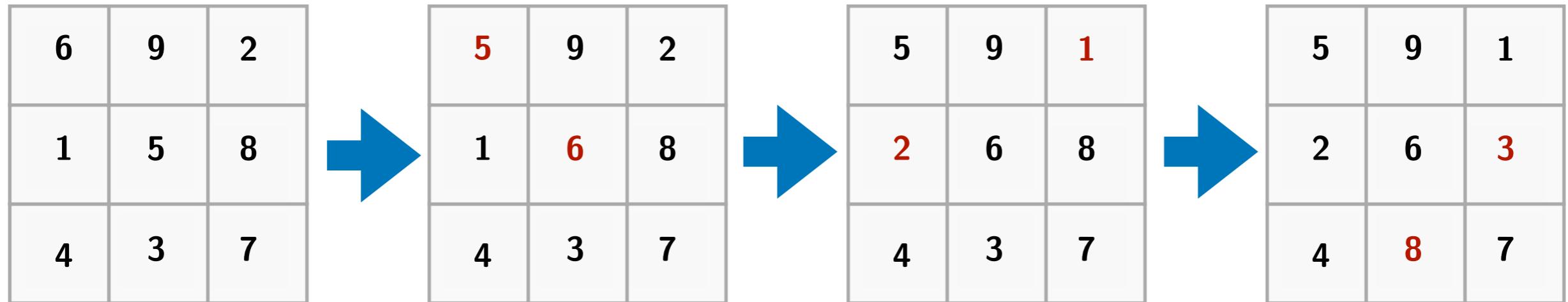
# Carré magique - solution initiale et voisinage

## Solution initiale

Génération aléatoire: en créer une aléatoirement qui satisfait toutes les contraintes dures

Notre choix d'espace de recherche: il suffit de permuter aléatoirement tous les chiffres dans la grille

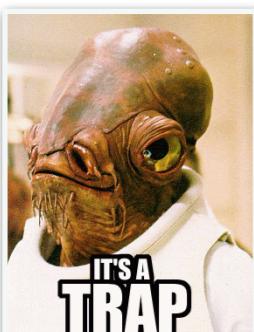
Astuce: une génération aléatoire est souvent simple et efficace (bonne diversification)



Que proposeriez-vous comme fonction de voisinage pour ce problème ?

Souhait: avoir un voisinage permettant d'améliorer la solution, sans être trop couteux à calculer

Voisinage 2-swap: permuter deux chiffres dans le tableau



Erreur fréquente: le voisinage doit aussi toujours préserver nos contraintes dures

Exemple invalide: un voisinage consistant à modifier la valeur d'une case n'est pas valide

Au plus il y a de contraintes dures, au plus on est limité dans la définition du voisinage

# Carré magique - fonction d'évaluation



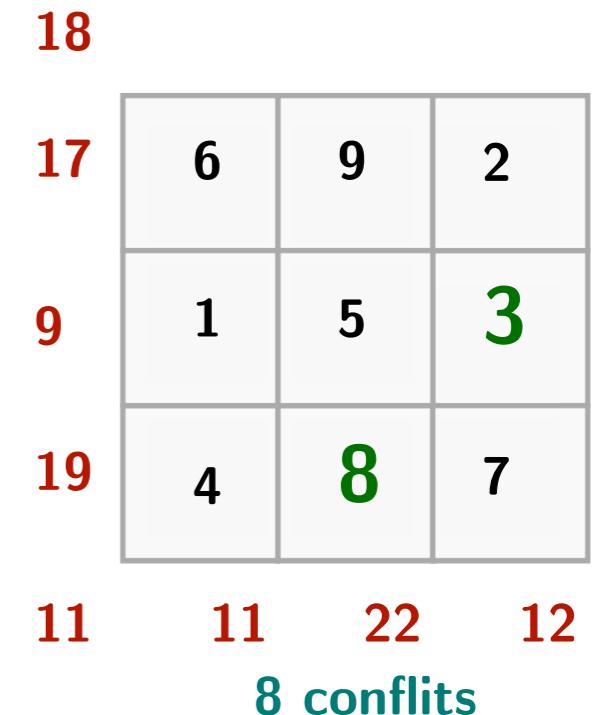
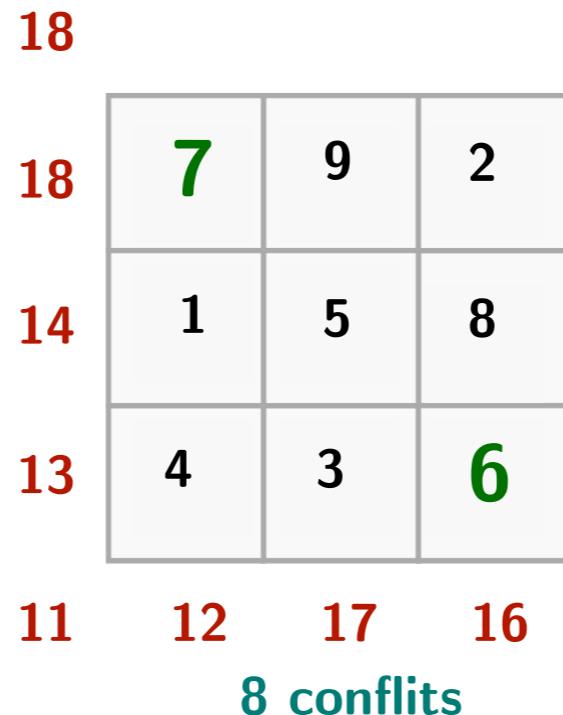
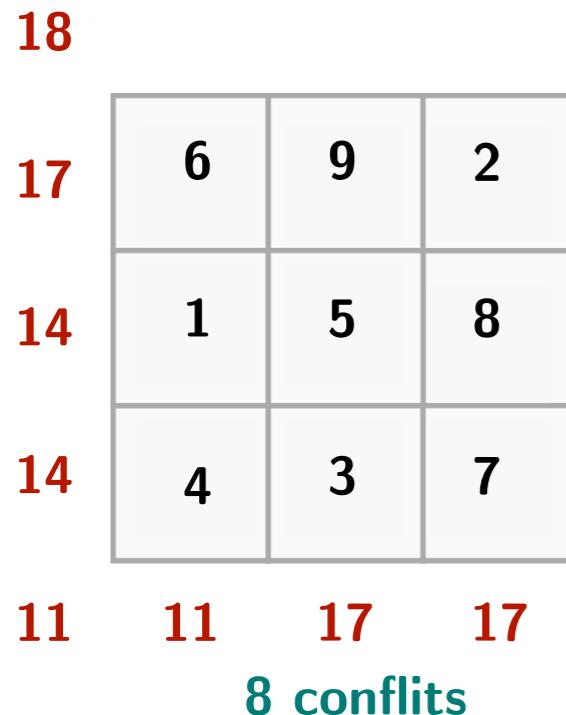
Que proposeriez-vous comme fonction d'évaluation

6	9	2
1	5	8
4	3	7

## Fonction d'évaluation 1: nombre de conflits

Principe: compter le nombre de contraintes molles qui ne sont pas respectées

Dans notre cas, on a un conflit pour chaque ligne, colonne, et diagonale différente de 15

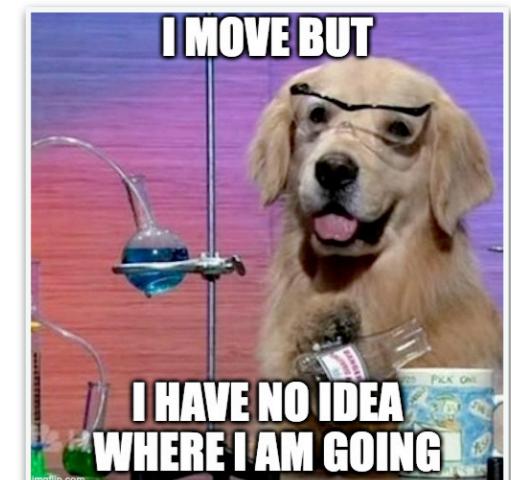


Observation: on a souvent des conflits partout

Inconvénient: la fonction d'évaluation n'est pas très informative

Cause: les solutions du voisinage ne peuvent pas être différencier adéquatement

Conséquence: la recherche risque de devenir complètement aléatoire



# Carré magique - fonction d'évaluation

## Fonction d'évaluation 2: pondération par le degré de non-satisfaction

Principe: pondérer chaque conflit par une mesure d'un degré de non-satisfaction

Dans notre cas, chaque conflit est pondéré par l'écart qu'il a avec la valeur recherchée (15)

6	9	2
1	5	8
4	3	7

3 18

2	17	6	9	2
1	14	1	5	8
1	14	4	3	7
11	11	17	17	

Conflits pondérés : 19

0 15

2	17	6	9	2
1	14	1	5	8
1	14	7	3	4

Conflits pondérés : 9

0 15

1	16	6	8	2
0	15	1	5	9
1	14	7	3	4

Conflits pondérés : 5

6	7	2
1	5	9
8	3	4

Une solution faisable est trouvée

Avantage: cette fonction permet d'orienter plus expressivement la recherche



*"Des fonctions d'évaluation expressives, tu préfèreras"*

# Carré magique - récapitulatif

## Modèle de recherche locale

Espace de recherche: les nombres 1 à 9 sont répartis dans la grille

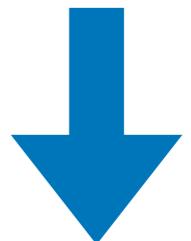
Solution initiale: les nombres sont répartis aléatoirement dans la grille

Voisinage: solutions pouvant être générées par un swap de deux nombres

Fonction d'évaluation: somme des conflits pondérés de chaque contrainte molle

On a un modèle de satisfaction pure (juste trouver une solution faisable)

6	9	2
1	5	8
4	3	7



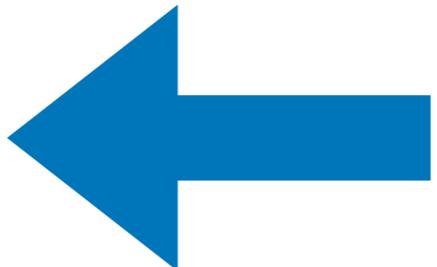
## Algorithme de résolution

Hill climbing: sélection du meilleur voisin

Fonction de validité: implicite dans l'algorithme de résolution

Fonction de sélection: implicite dans l'algorithme de résolution

6	7	2
1	5	9
8	3	4



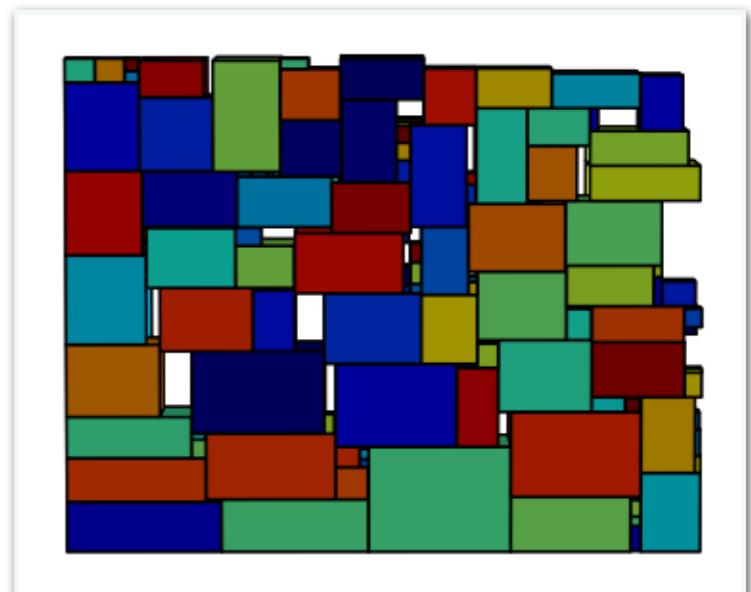
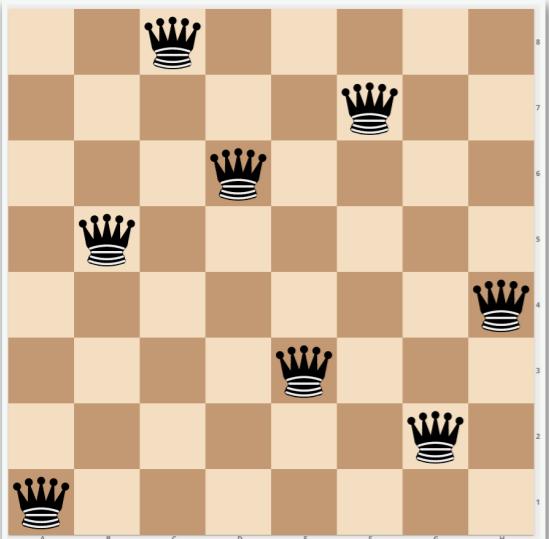
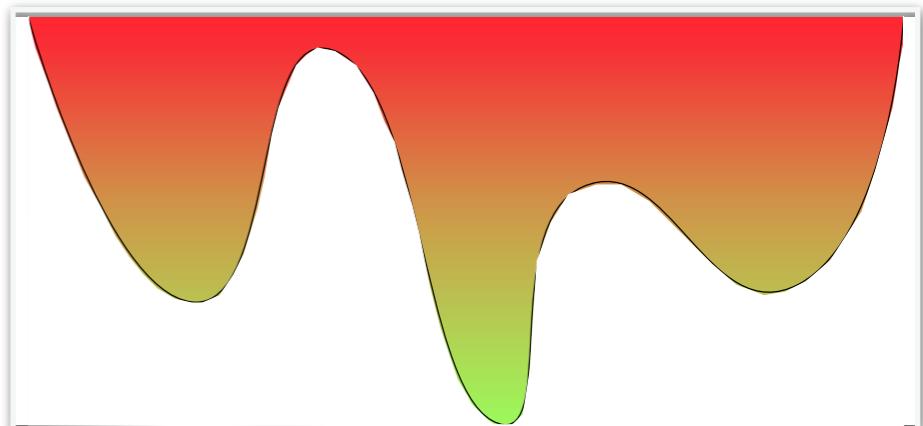
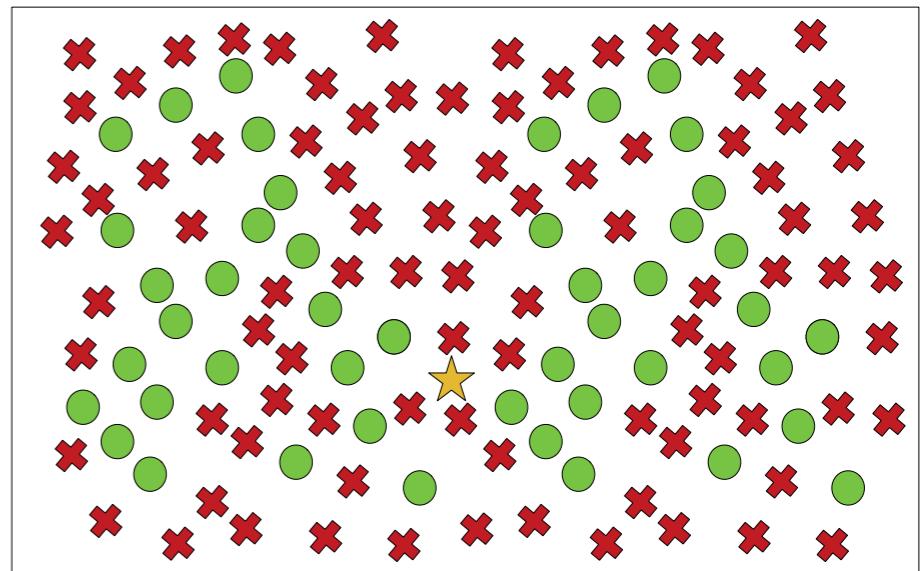
Exemple d'implémentation disponible dans les exercices du module

```
LocalSearch( $N, L, Q, f, \Theta$ ) :  
     $s = \text{generateInitialSolution}()$   
     $s^* = s$   
    for  $k \in 1$  to  $\Theta$  :  
         $G = [n \in N(s)]$   
         $V = [n \in L(G, s)]$   
         $s = Q(V, s)$   
        if  $f(s) < f(s^*)$  :  
             $s^* = s$   
    return  $s^*$ 
```

# Table des matières

## Recherche locale

- ✓ 1. Problèmes combinatoires de satisfaction et d'optimisation
- ✓ 2. Concepts et principes fondamentaux de la recherche locale
- ✓ 3. Formalisation de la recherche locale
- ✓ 4. Algorithme du *hill climbing*
- 5. Difficulté des minima locaux
- 6. Notion de voisinage connecté
- 7. Méthodes des redémarrages (*restarts*)
- 8. Algorithme du recuit simulé (*simulated annealing*)



# Etat actuel de la situation

## Modélisation d'une recherche locale

**Objectif:** concevoir un espace de recherche, une fonction de voisinage, de sélection, et d'évaluation

On a vu plusieurs techniques et bonnes pratiques de conception pour rendre la recherche efficace

Définition de l'espace de recherche: contraintes dures ou molles

Fonction de sélection: trouver un bon compromis entre qualité de la sélection et complexité calculatoire

Ces notions ont été appliquées concrètement sur deux problèmes (un autre à venir dans le devoir)

Heuristique de recherche: définition de ce modèle

## Algorithme de résolution

**Objectif:** exécuter la recherche locale définie précédemment

**Hill climbing:** sélectionner à chaque étape le meilleur voisin améliorant

**Variantes possibles:** sélectionner un voisin améliorant



Est-ce qu'on obtient toujours la meilleure solution ?



Est-ce qu'on obtient toujours une solution faisable ?

LocalSearch( $N, L, Q, f, \Theta$ ) :

$s = \text{generateInitialSolution}()$

$s^* = s$

for  $k \in 1$  to  $\Theta$  :

$G = [n \in N(s)]$

$V = [n \in L(G, s)]$

$s = Q(V, s)$

if  $f(s) < f(s^*)$  :

$s^* = s$

return  $s^*$

# Difficultés d'un minimum local

## Algorithme du *hill climbing*

**Principe:** choisir le meilleur voisin à chaque itération

**En cas d'égalité:** en choisir un aléatoirement parmi les meilleurs



Quelles sont les limitations de cet algorithme ?



**Exécution:** la recherche s'arrête une fois que la solution actuelle est meilleure que tous ses voisins

**Difficulté:** en général, on a aucune garantie que cette solution est optimale, voire même faisable



### Minimum local (ou maximum local)

Solution meilleure que tous les voisins, mais qui n'est pas forcément la meilleure globalement

$s$  est un minimum local  $\leftrightarrow f(s) \leq f(n) \forall n \in N(s)$



### Minimum global (ou maximum global)

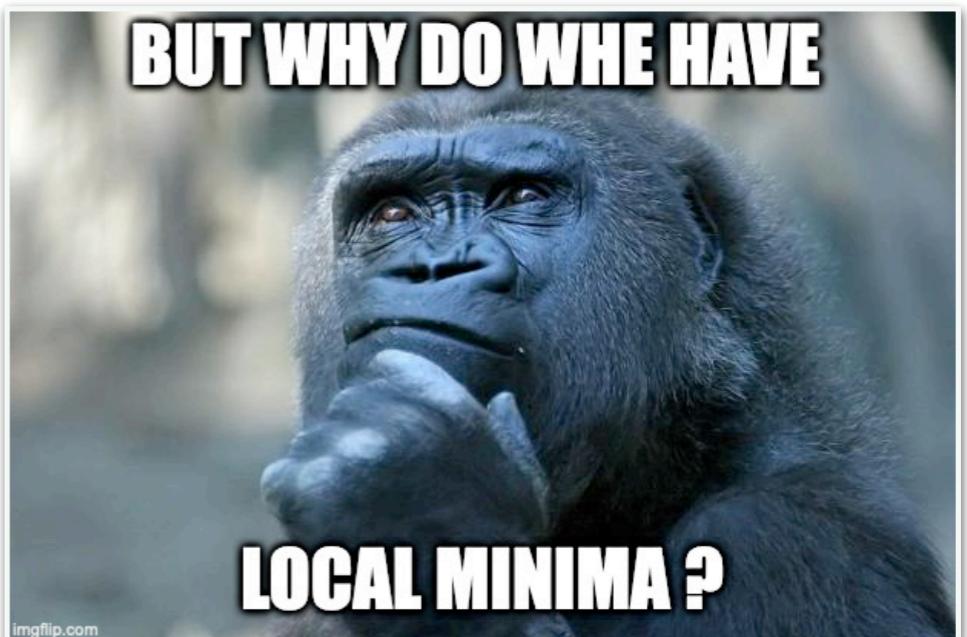
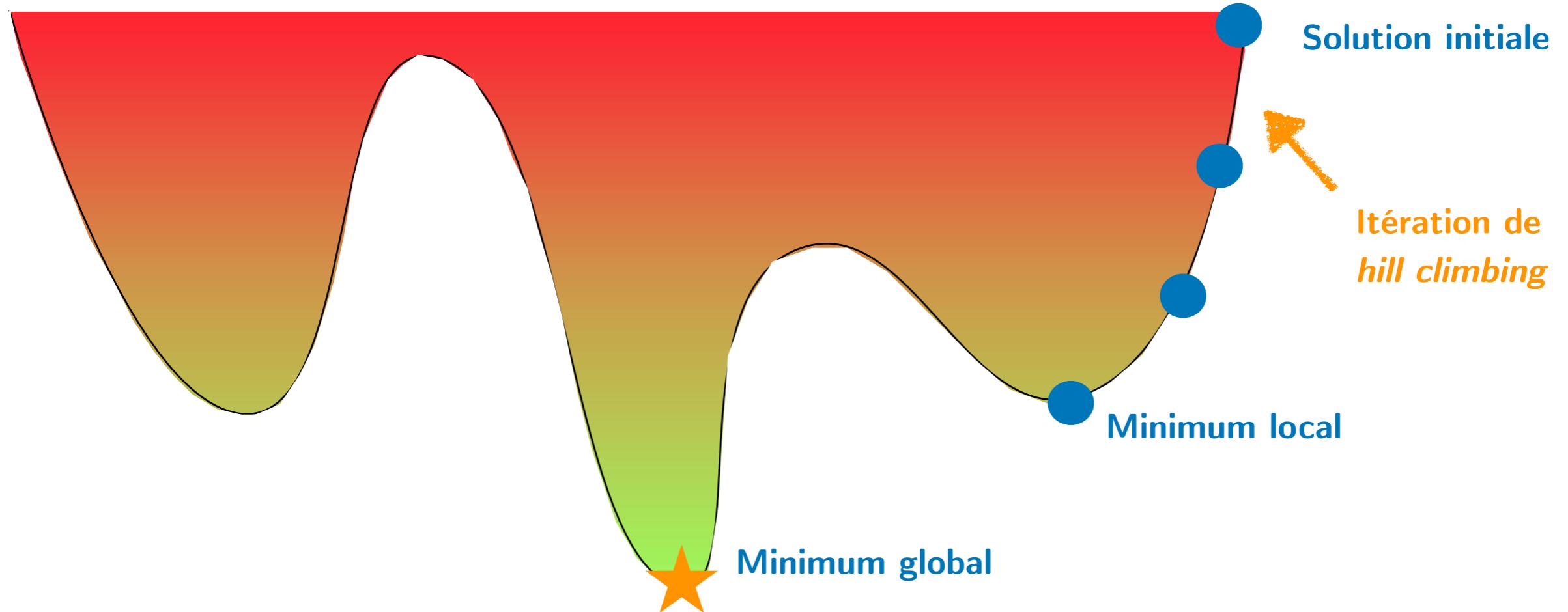
La meilleure (ou l'une des meilleures) solution de l'espace de recherche ( $S$ )

$s$  est un minimum global  $\leftrightarrow f(s) \leq f(s') \forall s' \in S$

**Remarque 1:** par définition, un minimum global est également un minimum local (mais particulier)

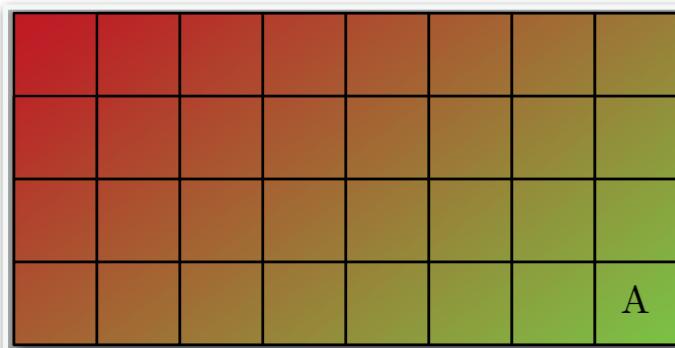
**Remarque 2:** notez bien que ces notions dépendent de la fonction d'évaluation utilisée

# Visualisation d'un minimum local



# Difficultés d'un minimum local

## Visualisation d'une recherche locale



**Voisinage:** effectuer un mouvement (G,D,H,B)

**Fonction d'évaluation:** indiqué par le dégradé de couleur

**Solution initiale:** coin supérieur gauche



Converge t-on vers un minimum global ?

Oui: dans le cas de cet exemple précis

En pratique: c'est rarement le cas



Qu'est-ce qui cause l'apparition de minima locaux ?

## Raison 1: le voisinage n'est pas connecté

**Intuition:** le minimum global est parfois impossible à atteindre à partir d'une autre solution

**Raison d'apparition:** éventuellement la **conséquence d'une mauvaise conception du voisinage**

## Raison 2: il peut exister plus d'un minimum local

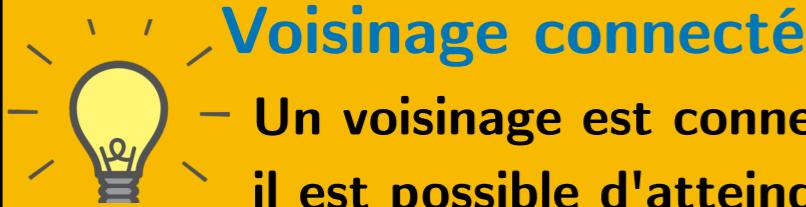
**Intuition:** il existe parfois (souvent) plusieurs minima locaux et seulement peu d'entre eux sont globaux

**Hill climbing:** donne seulement l'assurance d'amener à un minimum local

**Conséquence:** un minimum global pourra ainsi être manqué

**Raison d'apparition:** la fonction d'évaluation est non convexe (cas très fréquent)

# Voisinage connecté et non connecté



## Voisinage connecté

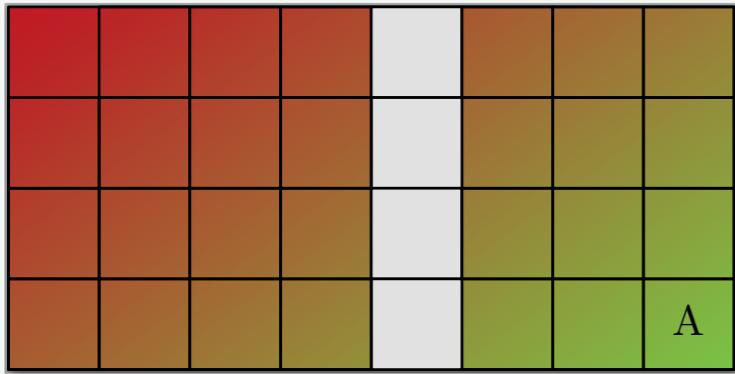
Un voisinage est connecté, si à partir de n'importe quelle solution de l'espace de recherche, il est possible d'atteindre un minimum global via la fonction de voisinage

Réiproquement, un voisinage non connecté est un voisinage qui ne respecte pas cette propriété

**Problème combinatoire de satisfaction:** revient à pouvoir obtenir une solution faisable

**Problème combinatoire d'optimisation:** revient à pouvoir obtenir la meilleure solution faisable

En supposant que la fonction d'évaluation est cohérente par rapport à la fonction objectif



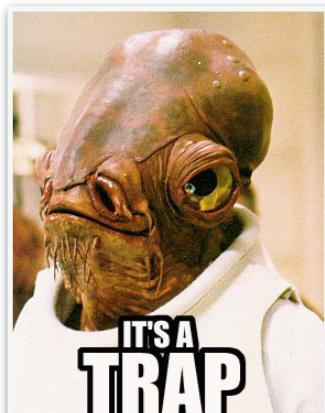
Est-ce que ces voisinages sont connectés ?

**Voisinage 1:** mouvement d'une case (G,D,H,B)

NON

**Voisinage 2:** mouvement d'une ou de deux cases (G,D,H,B)

OUI



**Notion de connectivité:** propriété du voisinage et non du problème

**Action possible:** pallier la difficulté d'un voisinage non connecté en un trouvant un autre

C'est pourquoi il est important de pouvoir repérer si un voisinage est connecté

# Preuve de connectivité

## Intérêt de la connectivité

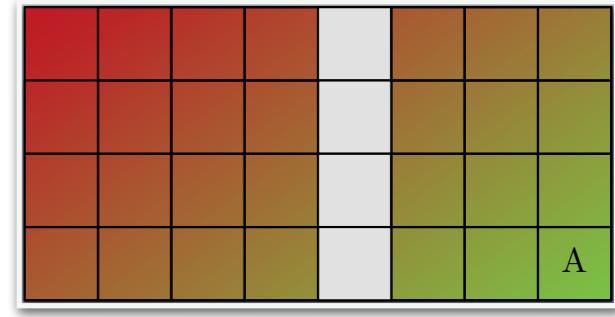
Il est toujours intéressant d'avoir un voisinage connecté

Avantage: plus grande liberté à la recherche pour trouver de bonnes solutions

Avantage: peut donner des garanties théoriques de convergence (p.e., *simulated annealing*)

Avantage: un voisinage non connecté peut être très difficile à exploiter pour obtenir la solution optimale

Difficulté: définir un voisinage connecté n'est pas toujours aisé



Comment prouver qu'un voisinage est connecté ?

Idée: construire une recherche amenant au minimum global en utilisant que les mouvements du voisinage

Intuition: si on peut la construire pour une solution quelconque, alors le voisinage est connecté



Comment construire cette recherche ?

Astuce: supposer qu'on connaisse le minimum global et utiliser cette information pour diriger la recherche

La construction de cette recherche est communément appelée **preuve de connectivité**

Exemples regardons cela pour le problème du carré magique

# Preuve de connectivité: problème du carré magique

## Problème du carré magique

Espace de recherche: toutes les permutations de chiffres

Voisinage: permuter deux chiffres (2-swap)

$n \times n$ : taille de la grille

$s$ : on pose  $s_{i,j}$  le chiffre à la position  $(i,j)$  d'une solution initiale quelconque (permutation de chiffres)

$o$ : on pose  $o_{i,j}$  le chiffre à la position  $(i,j)$  d'une solution faisable

2	7	6
9	5	1
4	3	8

1	2	3
4	5	6
7	8	9


$$o : \langle o_{1,1}, \dots, o_{3,3} \rangle \quad s : \langle s_{1,1}, \dots, s_{3,3} \rangle$$

MagicSquareConnectivity():

$\langle s_{1,1}, \dots, s_{n,n} \rangle = \text{generateRandomPermutation}()$

for  $(i, j) \in 1$  to  $n$ :

if  $s_{i,j} \neq o_{i,j}$ :

swap( $s_{i,j}, o_{i,j}$ )

return  $\langle s_{1,1}, \dots, s_{n,n} \rangle$

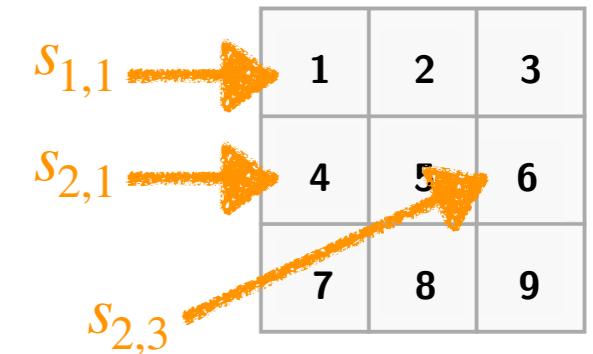
On génère une solution quelconque (permutation aléatoire)

Pour chaque case de la grille

si le chiffre de la case n'est pas celui de la solution faisable...

...On permute avec le bon chiffre (mouvement local autorisé par le voisinage)

On retourne finalement la solution construite, qui est optimale

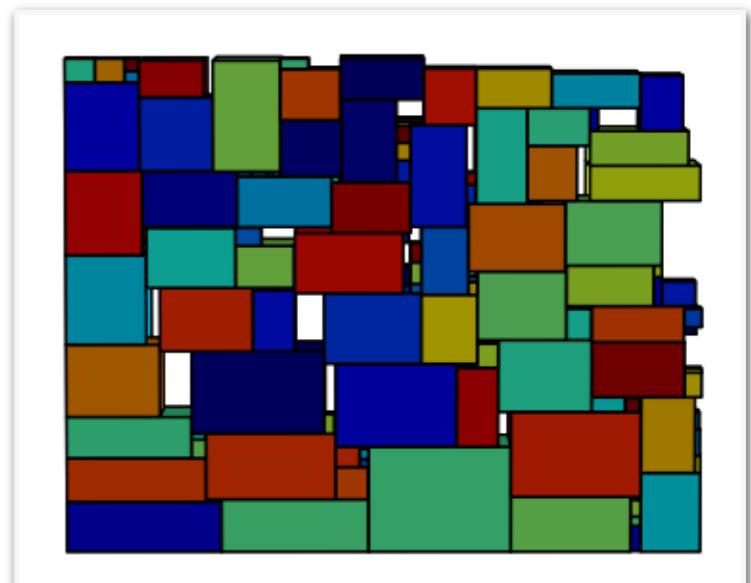
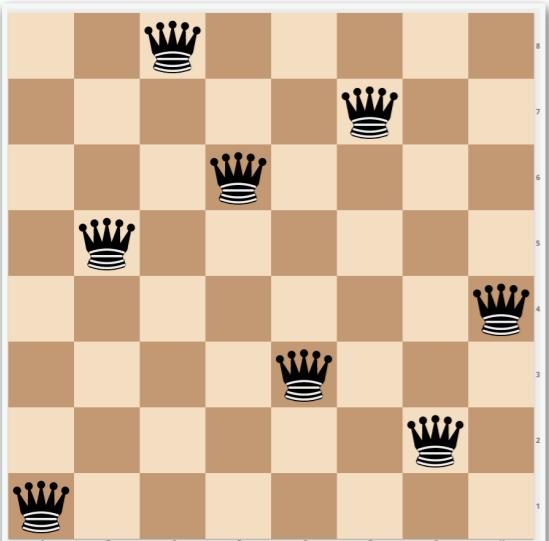
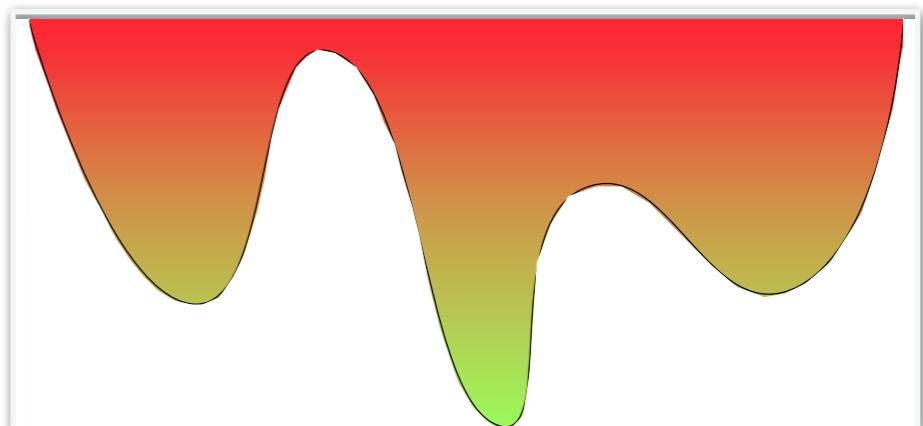
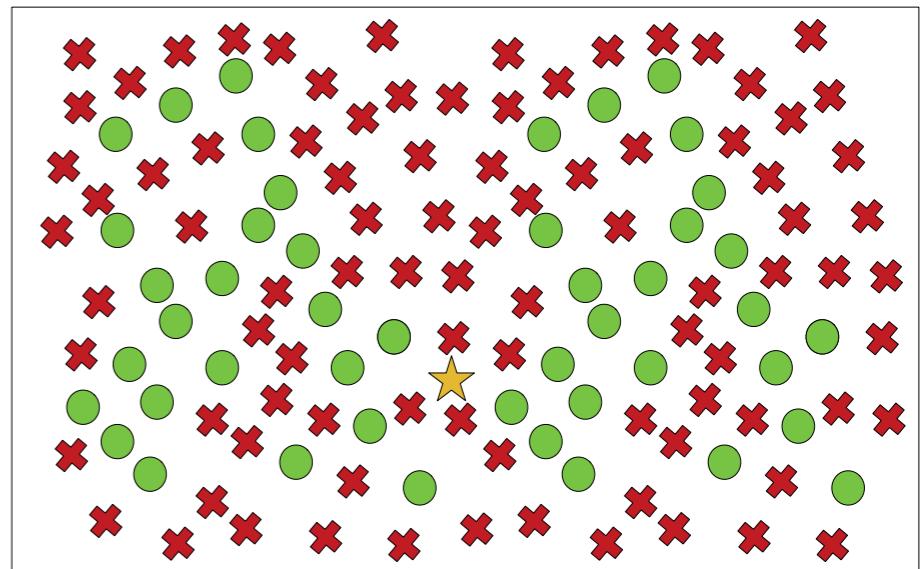


Résultat: on a construit une recherche amenant à une solution faisable, le voisinage est donc connecté !

# Table des matières

## Recherche locale

- ✓ 1. Problèmes combinatoires de satisfaction et d'optimisation
- ✓ 2. Concepts et principes fondamentaux de la recherche locale
- ✓ 3. Formalisation de la recherche locale
- ✓ 4. Algorithme du *hill climbing*
- ✓ 5. Difficulté des minima locaux
- ✓ 6. Notion de voisinage connecté
- 7. Méthodes des redémarrages (*restarts*)
- 8. Algorithme du recuit simulé (*simulated annealing*)



# Insuffisance des voisinages connectés



**Est ce qu'avoir un voisinage connecté est suffisant pour garantir une solution faisable ?**

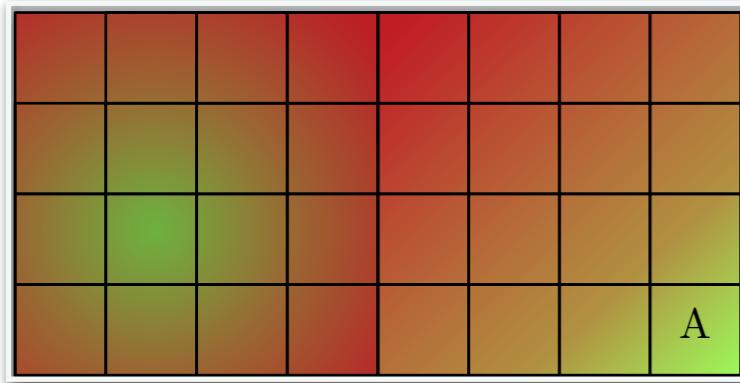


# Minimum local

## Défi des minima locaux

**Mauvaise nouvelle:** la connectivité n'est pas une propriété suffisante pour obtenir un minimum global

**Minimum local:** solution meilleure que tous les voisins, mais pas la meilleure globalement



$$\forall n \in N(s) : f(s) \leq f(n)$$

**Voisinage:** mouvement d'une case (G,D,H,B)

**Opposition avec un minimum global, qui est le meilleur de tous les minima**

**Cause:** apparition lorsqu'on a plusieurs minima à notre fonction d'évaluation

**Risque:** l'algorithme de *hill climbing* risque de tomber misérablement dans un minimum local

**Conséquence:** s'échapper d'un minimum local est un des défis principaux dans une recherche locale

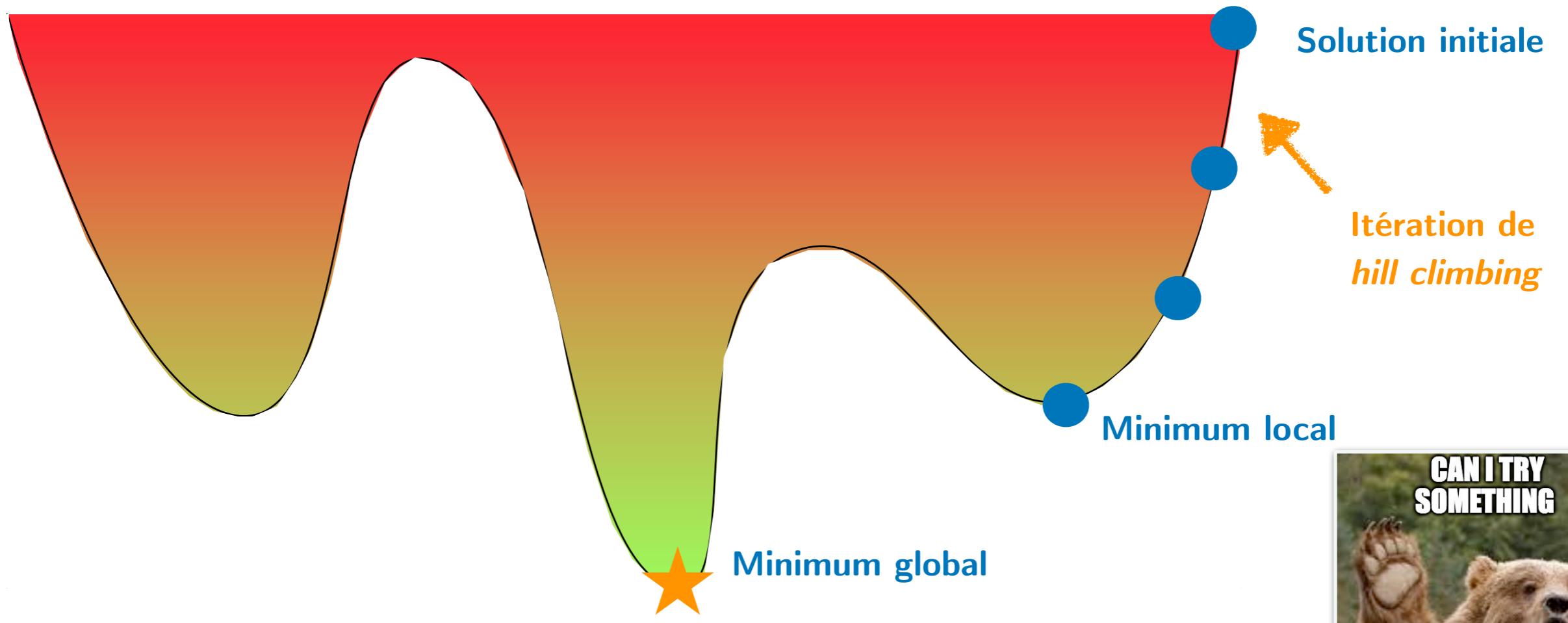
**Besoin:** leur présence justifie à elle seule le développement d'un grand nombre de nouvelles stratégies

**Problématique aussi présente dans d'autres champs de l'intelligence artificielle (p.e., *machine learning*)**



Comment peut-on s'échapper d'un minimum local ?

# Minimum local: stratégies de résolution



Stratégie 1: changer son voisinage (p.e., agrandissement simple)

Stratégie 2: commencer à un autre point de départ (p.e., redémarrages aléatoires)

Stratégie 3: accepter de dégrader notre solution actuelle (p.e., *simulated annealing*)

Stratégie 4: accepter de dégrader notre solution actuelle avec une mémoire (p.e., recherche tabou)

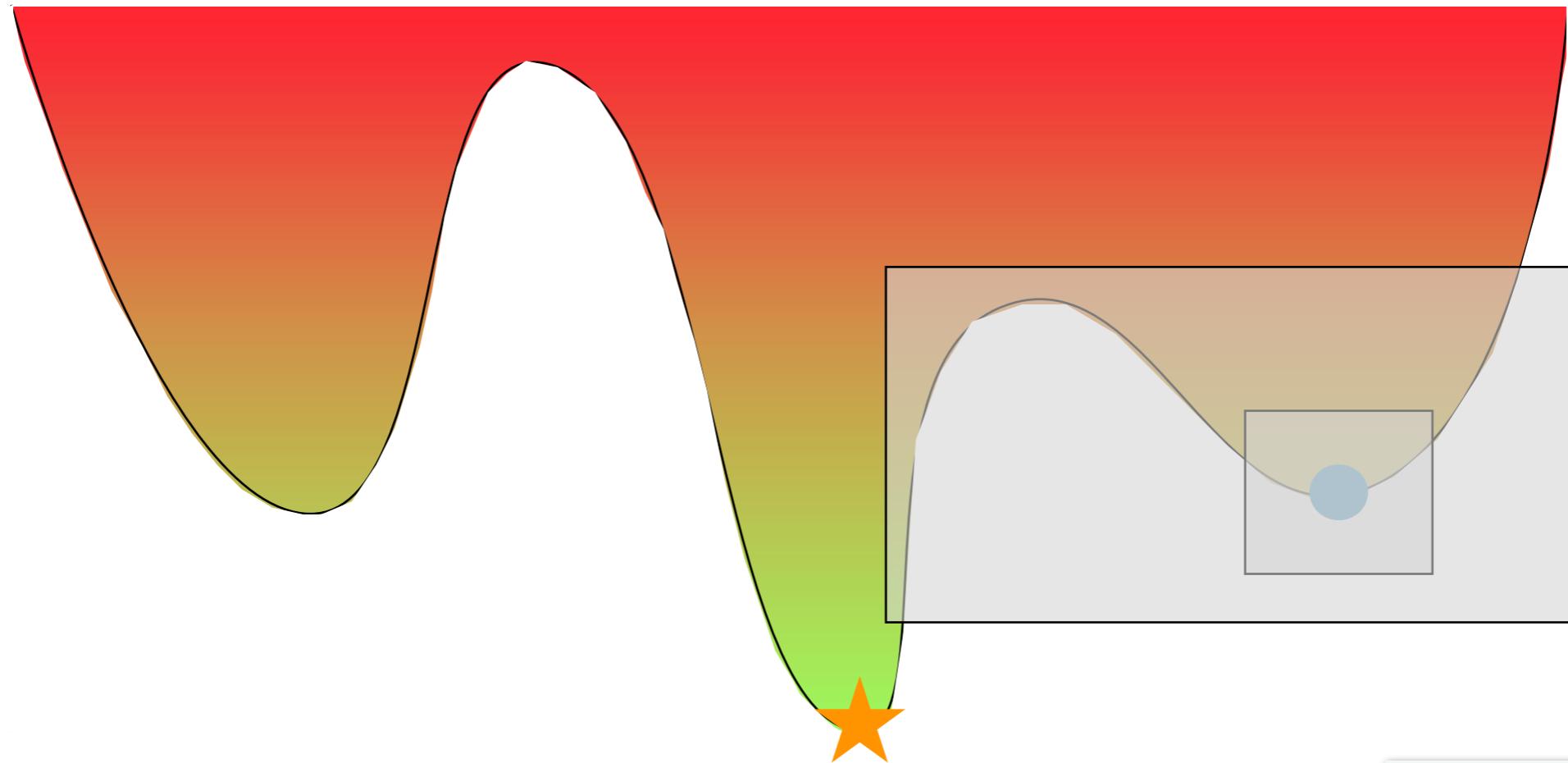
Les mét-heuristiques sont une réalisation concrète de ces stratégies



# Idée 1: Augmentation de la taille du voisinage

## Idée 1: augmentation de la taille du voisinage

Principe: définir la fonction de voisinage de sorte à intégrer un plus grand nombre de voisins



**Avantage:** donne la possibilité de faire de meilleurs mouvements

**Inconvénient:** devient plus coûteux à explorer

**Inconvénient:** pas toujours facile à définir

**Inconvénient:** solution souvent insuffisante pour s'échapper d'un minimum local



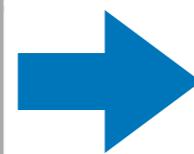
# Carré magique - voisinage plus grand



Avez-vous une idée d'un voisinage plus grand ?

6	9	2
1	5	8
4	3	7

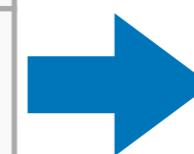
2-swap



5	9	2
1	6	8
4	3	7

6	9	2
1	5	8
4	3	7

3-swap



5	9	2
1	8	6
4	3	7



Quelle est la complexité d'une sélection du meilleur voisin dans ce voisinage ?

**2-swap:** complexité temporelle de  $O(n^2)$ , avec  $n$  le nombre de cases

**3-swap:** complexité temporelle de  $O(n^3)$ , avec  $n$  le nombre de cases

**4-swap:** complexité temporelle de  $O(n^4)$ , avec  $n$  le nombre de cases

**$k$ -swap:** complexité temporelle de  $O(n^k)$ , avec  $n$  le nombre de cases

L'exploration du voisinage devient vite beaucoup très coûteux (similaire au  $k$ -opt)



Est-ce que cela nous garantit au moins d'obtenir de meilleures solutions ?

# Carré magique - voisinage plus grand

## Carré magique de taille 4x4

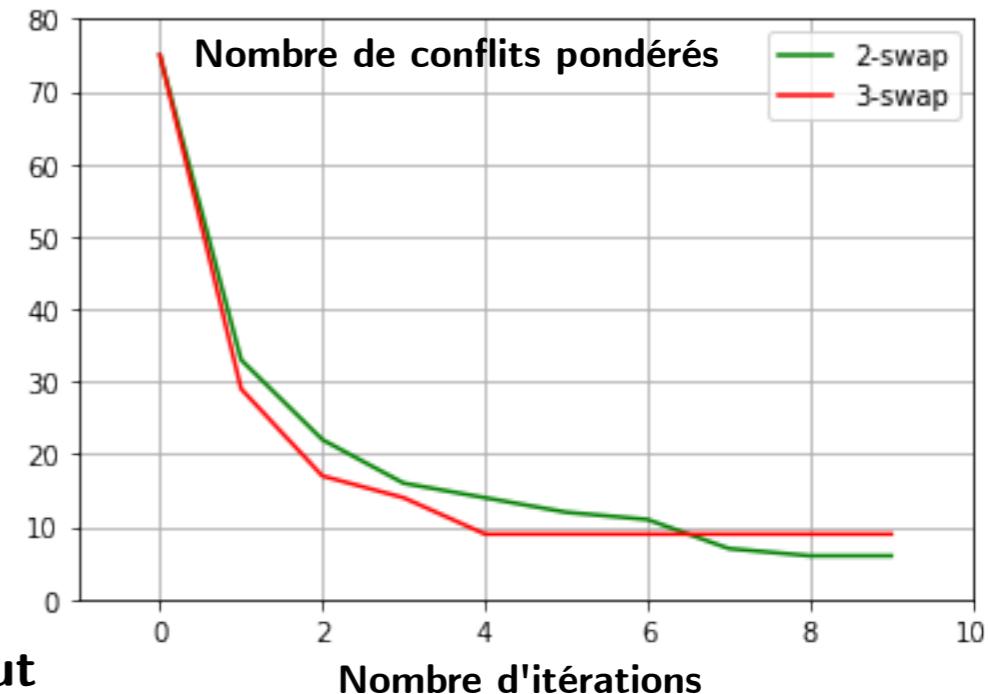
Voisinage du 2-swap: 0.04 secondes par itération

Voisinage du 3-swap: 3.32 secondes par itération

Fonction de sélection: le meilleur voisin



Qu'observe t-on dans ce résultat ?



Observation 1: le 3-swap réalise de meilleurs mouvements au début

Observation 2: il tombe dans un minimum local de moins bonne qualité

## Carré magique de taille 5x5

Voisinage du 2-swap: 0.11 secondes par itération

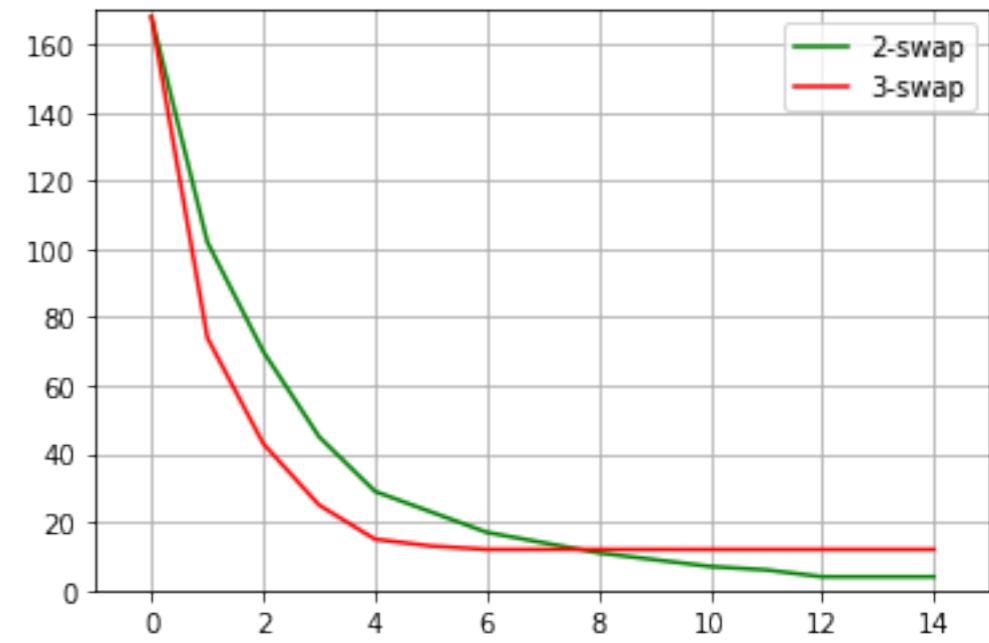
Voisinage du 3-swap: 13.82 secondes par itération

Fonction de sélection: le meilleur voisin

On a des observations similaires

Observation 3: le temps du 3-swap a considérablement augmenté

Observation 4: le temps du 2-swap augmente plus faiblement



Ainsi, considérer seulement un voisinage plus grand n'est généralement pas une amélioration suffisante

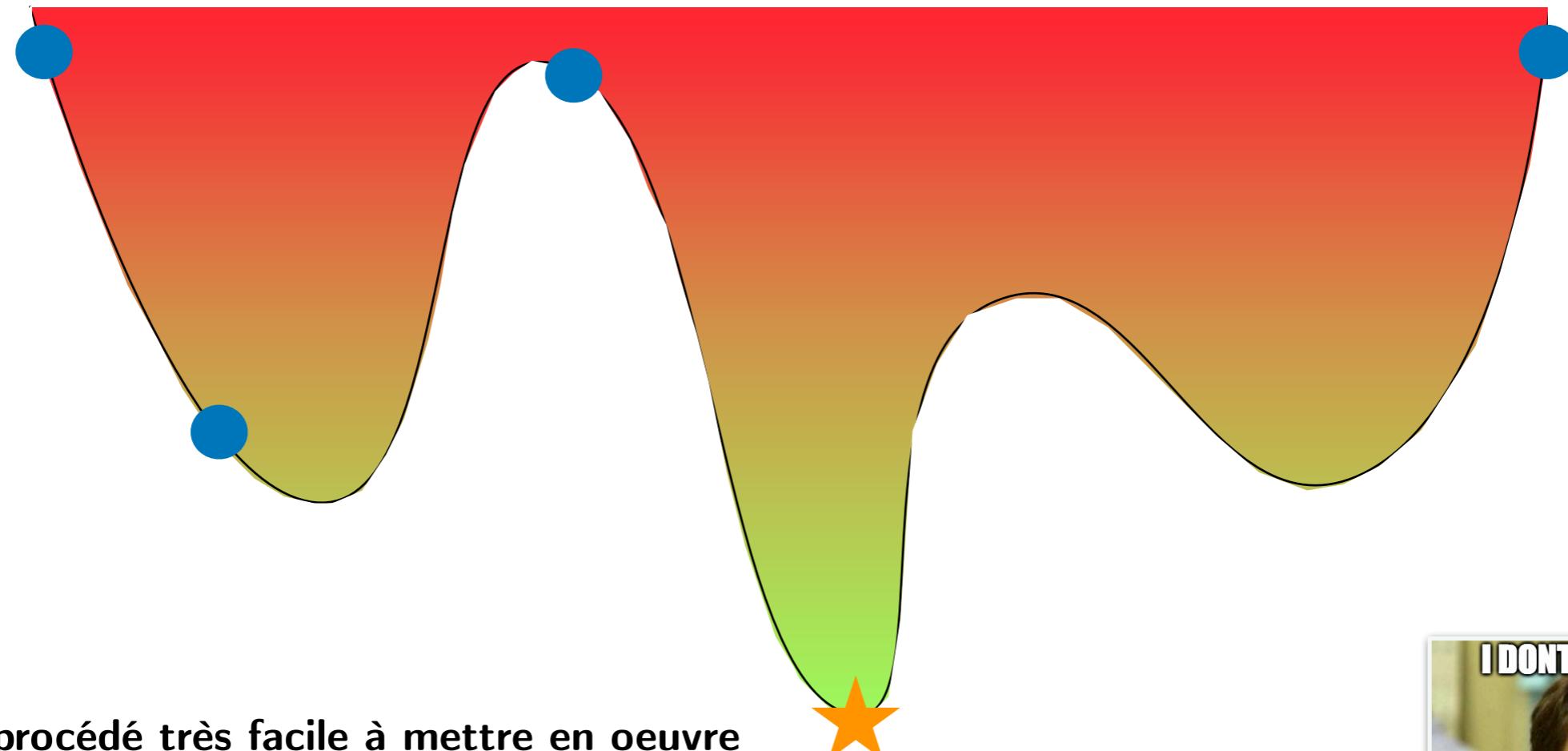
# Redémarrage à une autre situation initiale

## Idée 2: redémarrer aléatoirement la recherche

Etape 1: lancer une procédure de recherche locale jusqu'à tomber dans un minimum (peut-être local)

Etape 2: une fois le minimum est atteint, relancer une recherche à partir d'une autre solution initiale

La nouvelle solution initiale est déterminée aléatoirement parmi celles de l'espace de recherche



**Avantage:** procédé très facile à mettre en oeuvre

**Avantage:** favorise une exploration diversifiée de l'espace

**Inconvénient:** mécanisme pas suffisant pour découvrir les meilleures solutions

**Inconvénient:** les redémarrages augmentent le temps d'exécution de la recherche



# Carré magique - méthodes des redémarrages

## Carré magique de taille 4x4

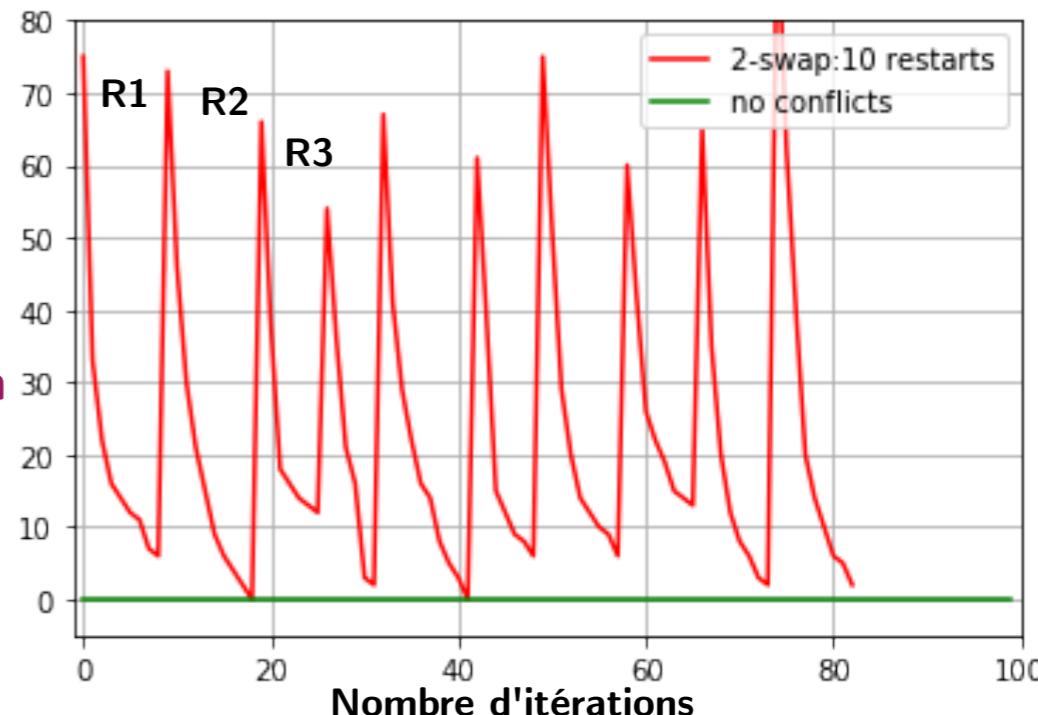
Chaque crête indique qu'un restart est effectué

2-swap avec 10 redémarrages: 3.53 secondes au total

Les redémarrages permettent de découvrir des nouveaux minima

Certains des redémarrages ont permis de résoudre le problème !

En pratique: on s'arrête une fois une solution faisable trouvée

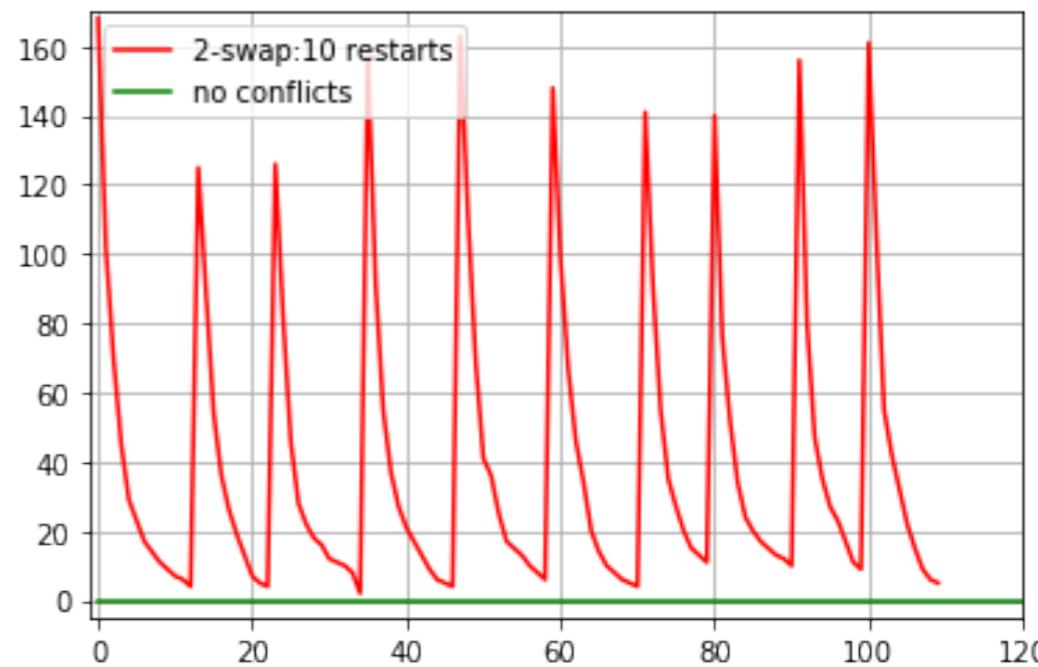
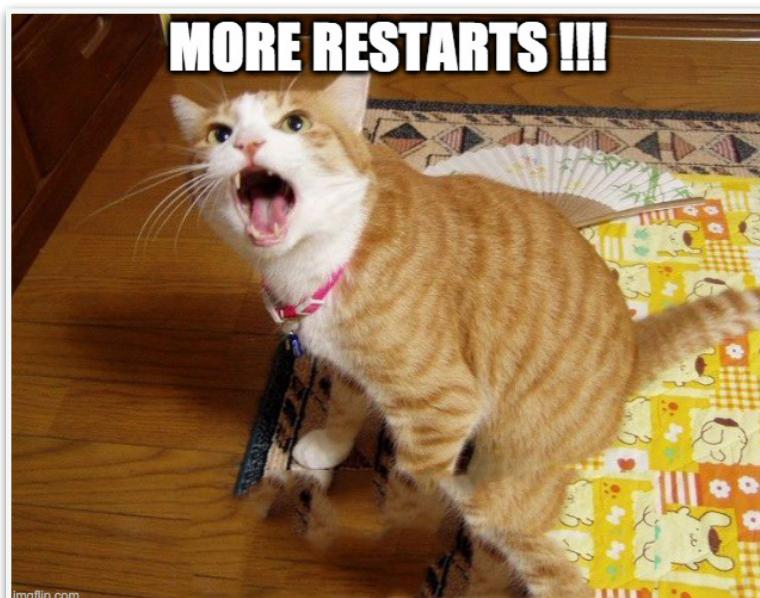


## Carré magique de taille 5x5

2-swap avec 10 redémarrages: 12.58 secondes au total

Cette fois ci, aucune solution faisable n'est trouvée

Au mieux, il reste 3 conflits



Avez-vous une idée pour améliorer les performances ?

# Carré magique - méthodes des redémarrages

## Carré magique de taille 5x5

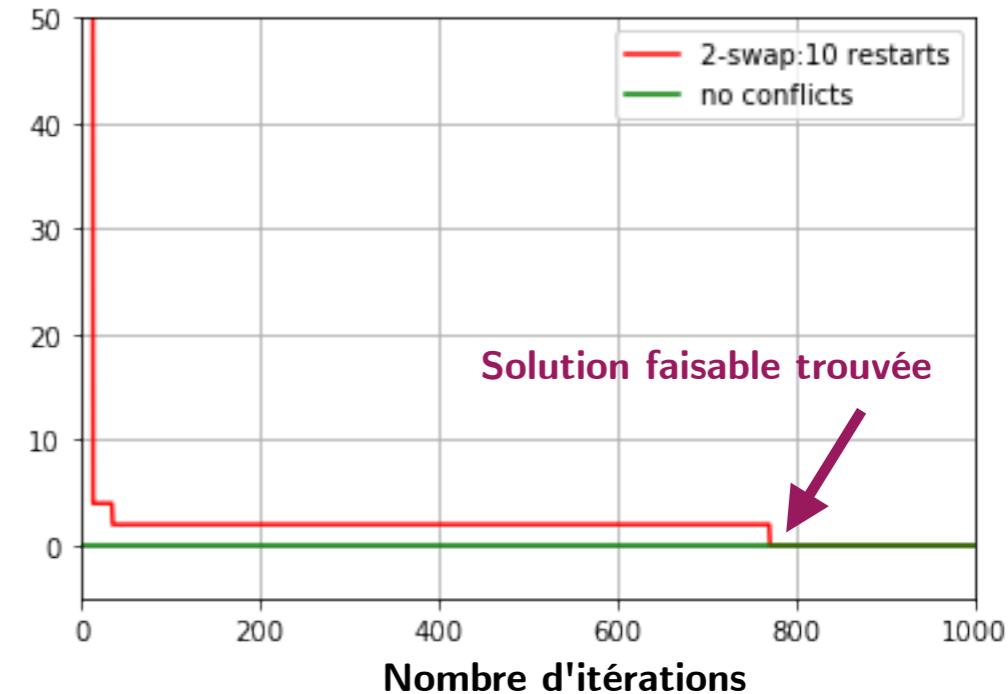
Nombre de redémarrages: 100

Temps d'exécution: 122.27 secondes

La figure indique la meilleure solution trouvée actuellement

Une solution faisable est trouvée après 780 mouvements locaux

Meilleure solution actuellement trouvée



## Carré magique de taille 6x6

Nombre de redémarrages: 100

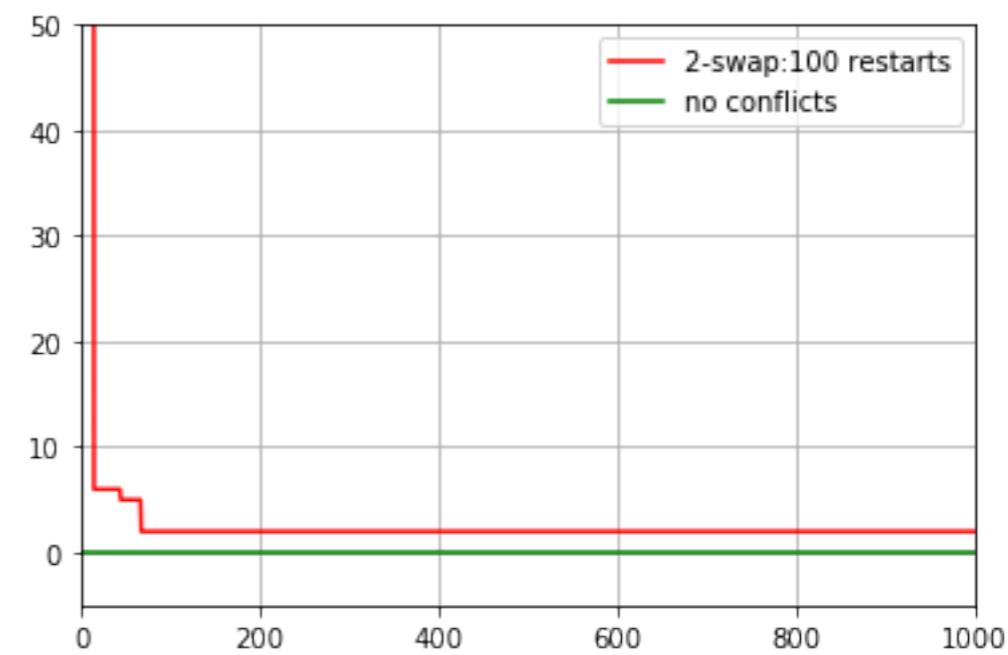
Temps d'exécution: 300 secondes

Aucune solution faisable n'est trouvée

La meilleure solution trouvée a deux conflits



Et avec plus de restarts ou un voisinage plus grand ?



Vous pourrez expérimenter cela par vous même (via le code donné dans les exercices du module)

Difficulté: la probabilité qu'une solution aléatoire amène à une solution faisable devient plus petite

# Recherche locale avec restarts

## Algorithme de recherche locale avec redémarrage

```
LocalSearchWithRestart( $N, L, Q, f, \Theta, \Gamma$ )
     $s^* = \perp$ 
    for  $i \in 1$  to  $\Gamma$  :
         $s = \text{LocalSearch}(N, L, Q, f, \Theta)$ 
        if  $f(s) < f(s^*)$  :
             $s^* = s$ 
    return  $s^*$ 
```

```
LocalSearch( $N, L, Q, f, \Theta$ ) :
     $s = \text{generateInitialSolution}()$ 
     $s^* = s$ 
    for  $k \in 1$  to  $\Theta$  :
         $G = [n \in N(s)]$ 
         $V = [n \in L(G, s)]$ 
         $s = Q(V, s)$ 
        if  $f(s) < f(s^*)$  :
             $s^* = s$ 
    return  $s^*$ 
```

**Principe:** on exécute simplement plusieurs recherches locales

**Paramètre à définir:** un critère d'arrêt sur le nombre de redémarrages

**Avantage 1:** mécanisme très simple qui s'intègre facilement avec les algorithmes de recherche locale

**Avantage 2:** favorise la diversification de la recherche



**Bonne pratique:** exploitez entièrement le temps d'exécution qui vous est alloué

Si votre recherche locale prend 1 minute, et que vous en avez 10 à disposition...

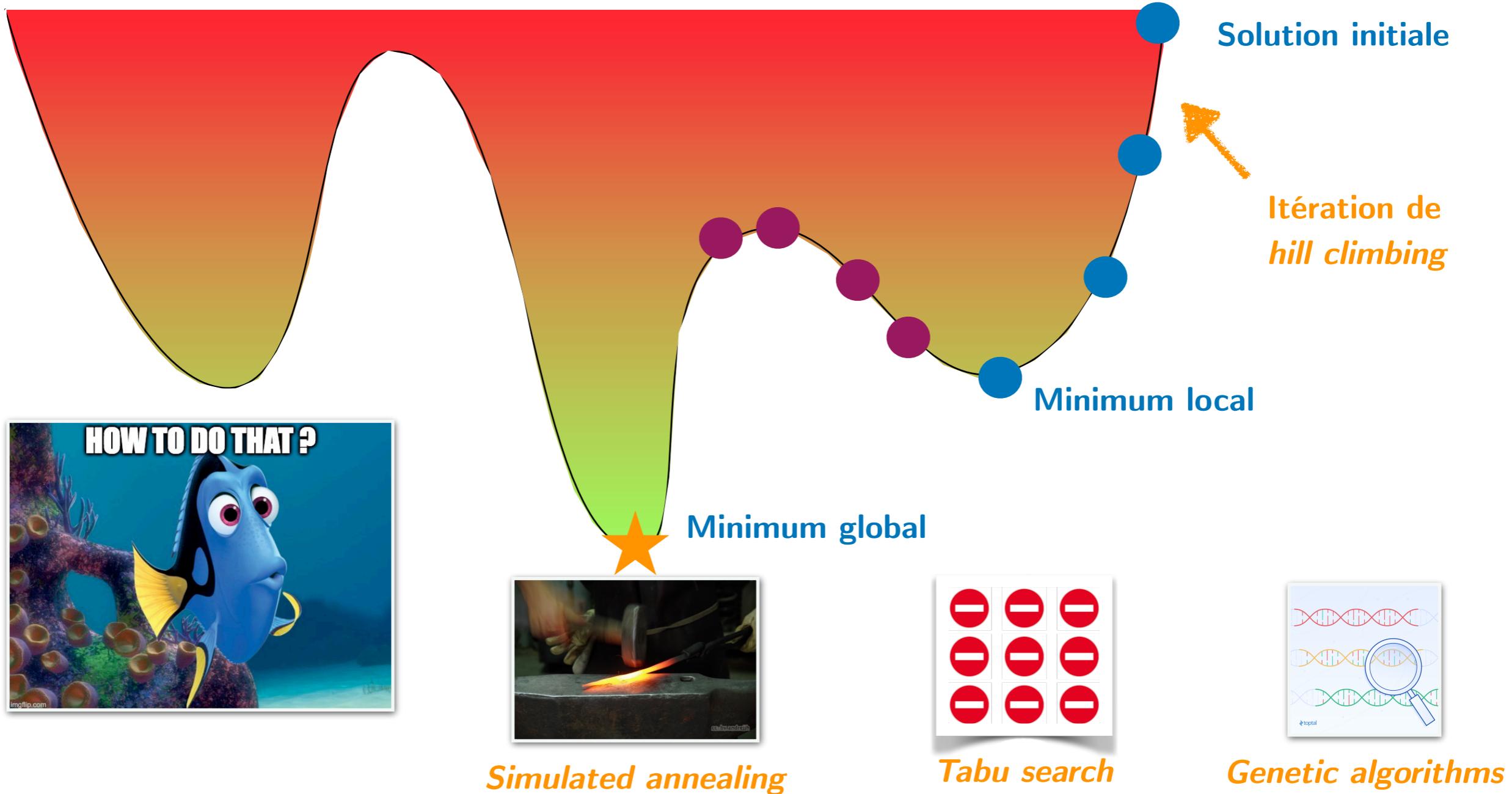
... redémarrez ! et gardez le meilleur résultat trouvé

Les redémarrages permettent de s'adapter à n'importe quel temps d'exécution

# Idée 3: dégrader notre solution actuelle

## Idée 3: autoriser de dégrader notre solution

Objectif: dégrader intelligemment la solution pour s'échapper d'un minimum local



Ce module: étude du *simulated annealing* (recuit simulé)

Lectures complémentaires (et INF6102): recherche tabou et algorithmes génétiques

# Simulated annealing - inspiration naturelle

## Principes de la métallurgie

**Objectif:** forger une forme avec du métal

**Métal chaud:** malléable et peut être modelé

**Métal froid:** non malléable et prend une forme définitive



cc-by andrejhh

## Début du processus

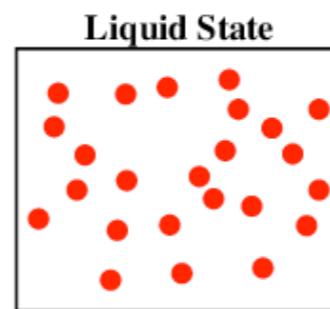
**Haute température:** le métal est malléable

Donne au forgeron la possibilité de modifier la forme du métal, selon le produit final voulu

## Tout au long du processus

**Principe:** diminution progressive de la température

Donne la possibilité de maîtriser la solidification du métal

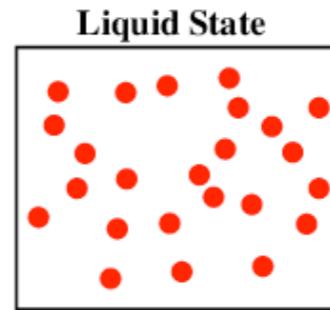


Une réduction brusque risque de donner un résultat non stable

## Fin processus

**Faible température:** le métal est solide et non malléable

Le métal prend sa forme finale



# Métaheuristique du *simulated annealing* (recuit simulé)

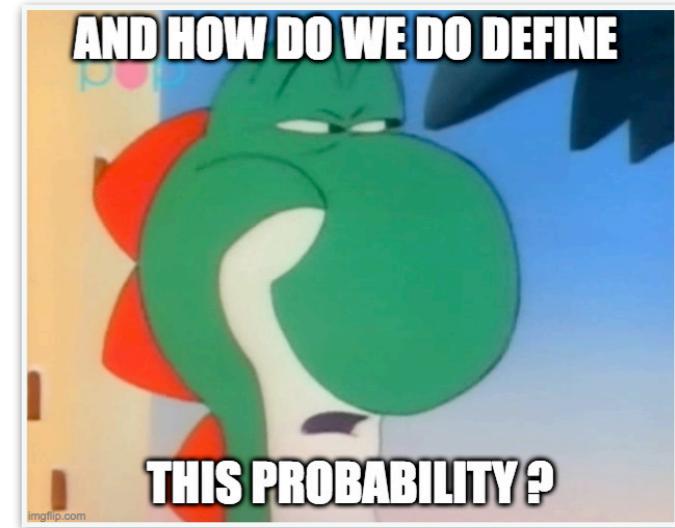
## Fonctionnement général

**Principe:** permettre la sélection de voisins moins bon que la solution actuelle

**Règle de sélection:** on tire aléatoirement un voisin

**Cas 1:** s'il améliore la solution courante, on l'accepte

**Cas 2:** sinon, on l'accepte avec une certaine probabilité  $p$



## Probabilité de sélection

**Idée générale:** la probabilité dépend de la qualité du voisin, et du nombre d'itérations déjà effectuées

**(1) On considère la différence de coût du voisin  $n$ , par rapport à notre solution actuelle  $s$**

$$\Delta(n, s) = f(n) - f(s)$$

$\Delta(n, s) \leq 0 \rightarrow f(n) \leq f(s)$  : voisin qui améliore ou maintient la solution (pour une minimisation)

$\Delta(n, s) > 0$  et élevé : voisin qui dégrade fort la solution

$\Delta(n, s) > 0$  et faible : voisin qui dégrade faiblement la solution

**Interprétation:** cette valeur donne une mesure d'à quel point un voisin est pire que la solution actuelle

**Intuition de la sélection:** plus le mouvement est mauvais, plus la probabilité de sélection sera faible

**(2) Au plus la recherche avance, au plus on va éviter de dégrader la solution**

**Interprétation:** on veut faire converger la recherche

**Note:** ce schéma probabiliste est également connu sous le nom d'algorithme de *Metropolis-Hastings*

# Simulated annealing - Principe de sélection

## Fonction de sélection

Principe 1: on accepte toujours un voisin qui améliore la solution actuelle

Principe 2: on accepte un voisin dégradant avec une certaine probabilité, qui dépend de la qualité du voisin

Principe 3: cette probabilité décroît avec le temps et dépend d'un paramètre

$\Delta(n, s) = f(n) - f(s)$  (différence du coût d'un voisin avec la solution actuelle)

## Sélection en cas de non-dégradation

Situation de non-dégradation:  $\Delta(n, s) \leq 0$

Un voisin améliorant est toujours choisi

## Sélection en cas dégradation

Situation de dégradation:  $\Delta(n, s) > 0$

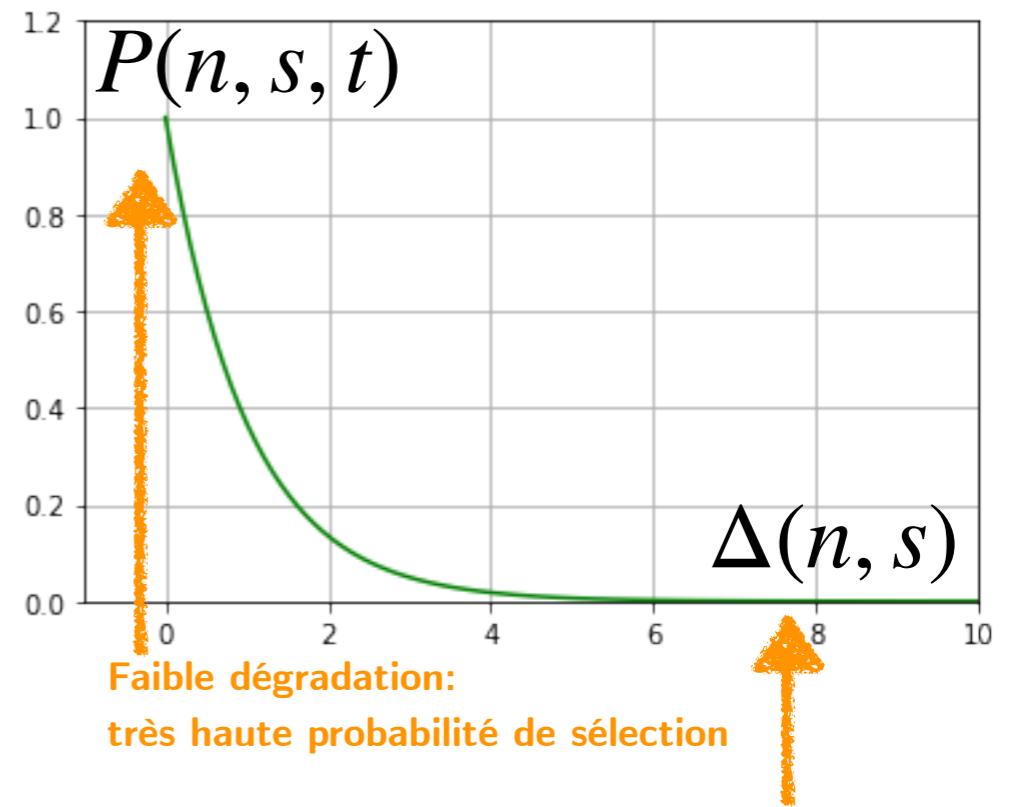
$$P(n, s, t) = e^{\frac{-\Delta(n, s)}{t}} \text{ (probabilité de sélection)}$$

$\Delta(n, s)$  élevé : on a une faible probabilité de sélection

$\Delta(n, s)$  faible : on a une haute probabilité de sélection

Intuition 1: la probabilité est modulée pour être plus propice à accepter des voisins faiblement dégradant

Intuition 2: La probabilité est paramétrée par une valeur  $t$ , qui est appelée la **température** de l'algorithme



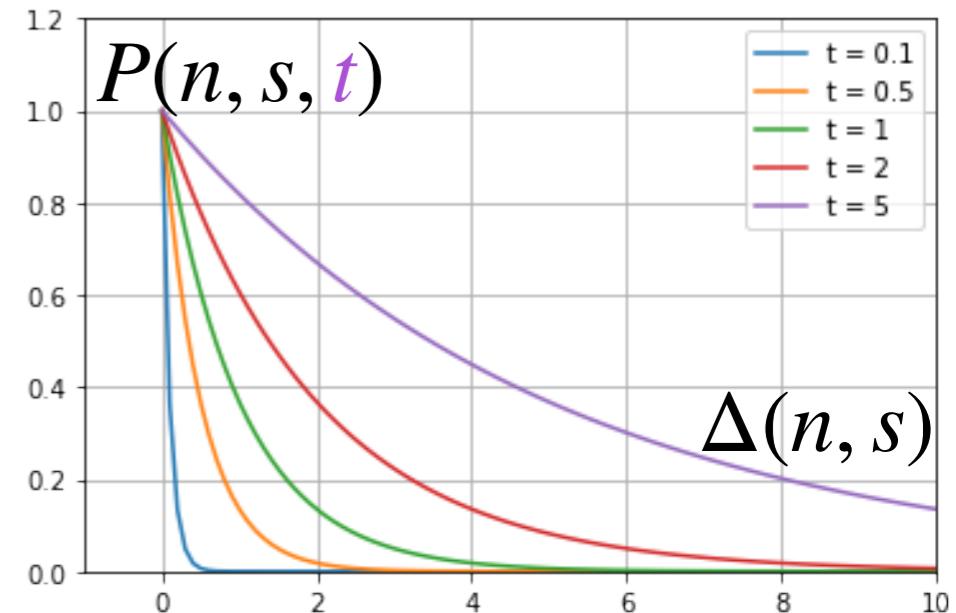
# Simulated annealing - température de l'algorithme

$$P(n, s, t) = e^{\frac{-\Delta(n, s)}{t}} \text{ (probabilité de sélection)}$$

Quel est l'impact de la température sur la sélection ?

Faible température: faible probabilité d'accepter un mauvais voisin

Haute température: haute probabilité d'accepter un mauvais voisin



Quand préfère t-on avoir une faible ou haute température ?

Début de la recherche: on veut favoriser l'exploration de l'espace afin de découvrir plein de solutions

Fin de la recherche: on veut favoriser l'exploitation d'une solution afin d'obtenir le meilleur coût possible

Procédure générale: diminuer la température au fil de la recherche

## Schéma de décroissance géométrique

Principe: réduire la température selon une suite géométrique

Température initiale :  $t_0$  (valeur assez élevée pour autoriser toutes les sélections)

Schéma de décroissance :  $t_{k+1} = \alpha t_k$  (typiquement,  $\alpha$  est entre 0.8 et 0.99)

Critère d'arrêt : lorsqu'on n'observe plus aucune amélioration dans la recherche

En pratique, la constante multiplicative de décroissance est un paramètre à calibrer par l'utilisateur

# Simulated annealing - algorithme

```
SimulatedAnnealing( $N, L, Q, f, \Theta, t_0, \alpha$ ) :
```

```
     $s = \text{generateInitialSolution}()$ 
```

```
     $s^* = s$ 
```

```
     $t = t_0$ 
```

```
    for  $k \in 1$  to  $\Theta$  :
```

```
         $G = [n \in N(s)]$ 
```

```
         $V = [n \in L(G, s)]$ 
```

```
         $c \sim V$  with uniform probability
```

```
         $\Delta = f(c) - f(s)$ 
```

```
        if  $\Delta \leq 0$  :
```

```
             $s = c$ 
```

```
        elif  $\Delta > 0$   $\wedge$  success with probability  $e^{-\frac{\Delta}{t}}$  :
```

```
             $s = c$ 
```

```
        if  $f(s) < f(s^*)$  :
```

```
             $s^* = s$ 
```

```
         $t = \alpha t$ 
```

```
    return  $s^*$ 
```

**Entrée:** critère d'arrêt, température initiale, et taux de décroissance

On commence avec la recherche avec la température initiale

La fonction de validité doit permettre des voisins dégradant

On choisi aléatoirement un voisin

Un voisin non-dégradant est toujours pris

Un voisin qui dégrade est choisi selon la probabilité de sélection

Mise à jour de la température

CONVERGENCE TO OPTIMAL SOLUTION



**Résultat théorique:** convergence vers l'optimum si les paramètres sont bien calibrés

**En pratique:** cette convergence est plus lente qu'une recherche exhaustive

**Variante 1:** schéma de décroissance polynomiale

**Variante 2:** intégration d'un mécanisme de redémarrage

**Variante 3:** augmentation de la température quand la recherche ne progresse plus

SLOWER THAN EXHAUSTIVE SEARCH



# Zoo des métaheuristiques

## Zoo des métaheuristiques

Il existe un nombre impressionnant de méthodes autres que le *simulated annealing*

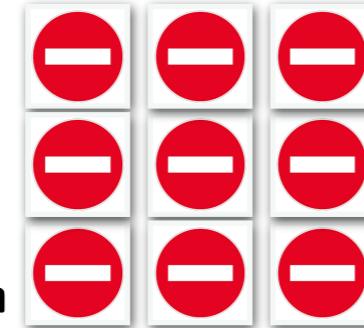
Chacune ont leurs forces, faiblesses, et champs d'application

Ces méthodes sont communément appelées **métaheuristiques**

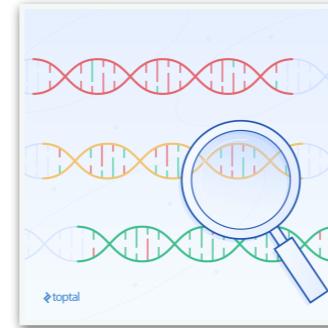
Aller plus loin: Métaheuristiques appliquées au génie informatique (INF6102 - cours que je donne)

Exemples: quelques unes parmi les plus connues (et efficaces)

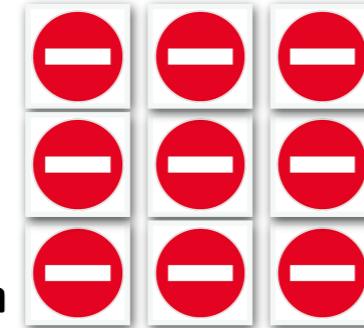
Tabu search



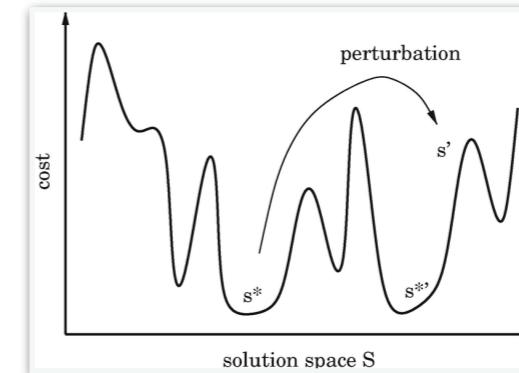
Genetic algorithms



Iterated local search



Ant colony optimization



Tendance: la pertinence de nombreuses métaheuristiques a été justifiée par leur *inspiration naturelle*

Cuckoo search



African Buffalo optimization



Jellyfish Search



Emperor Penguins Colony

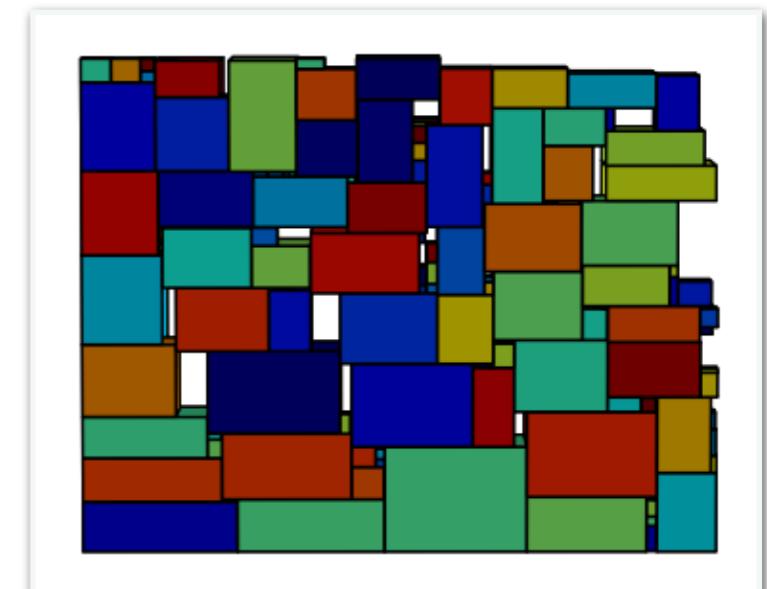
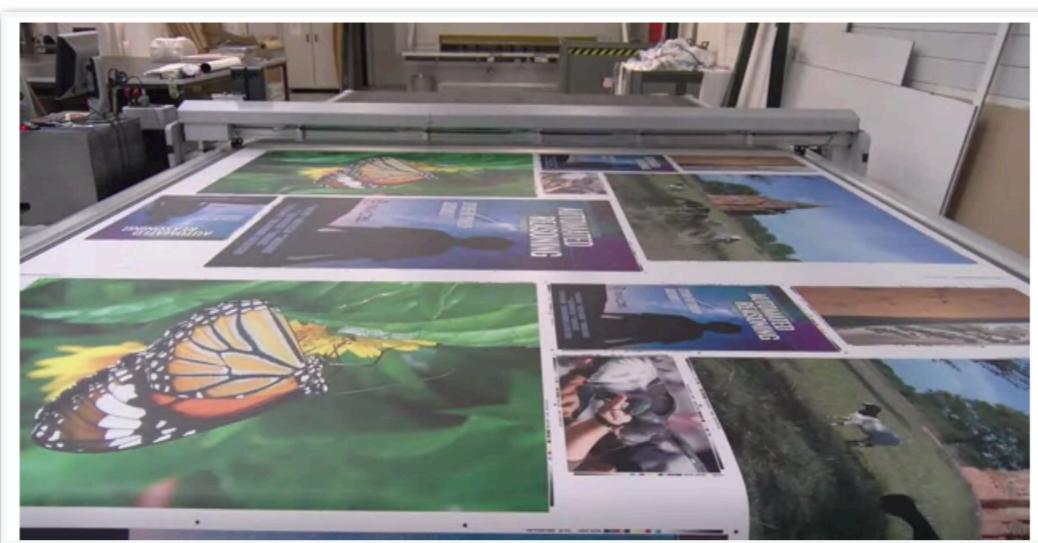
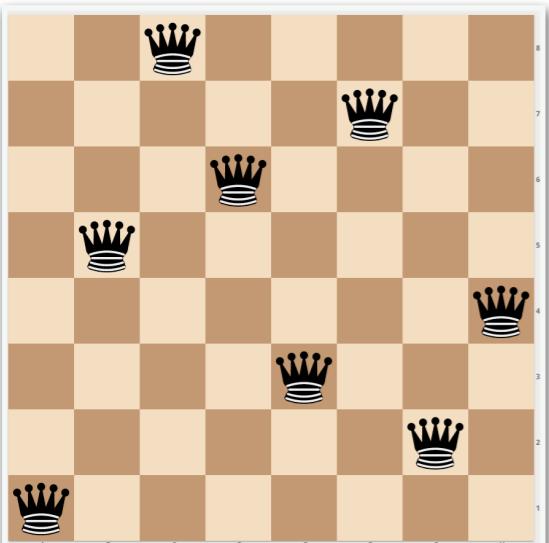
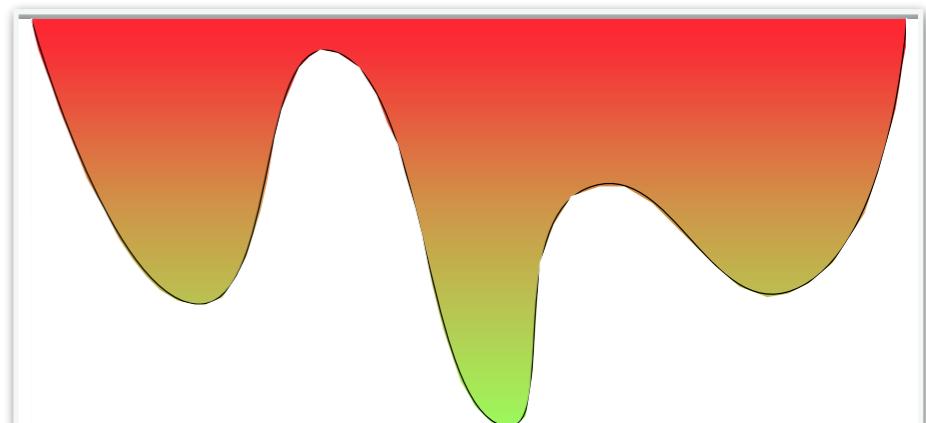
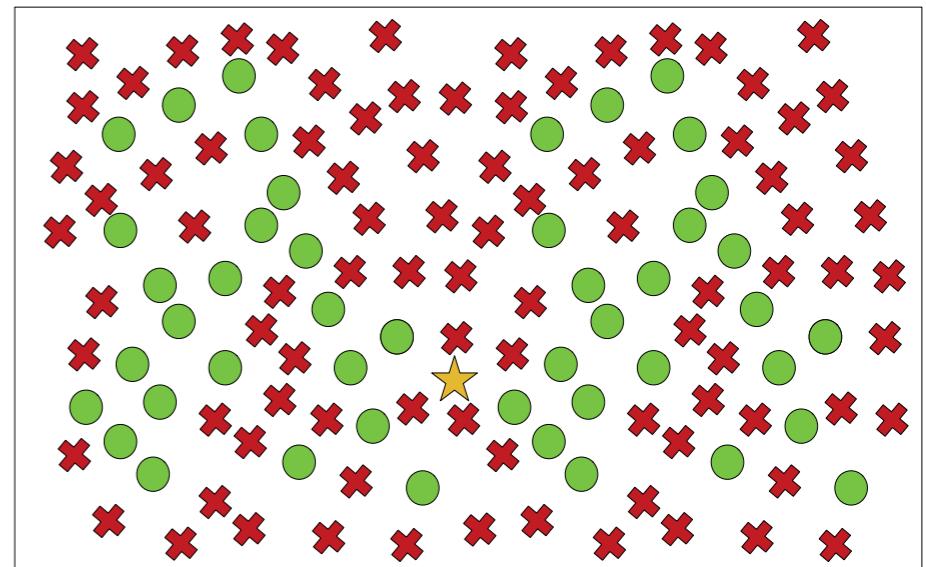


L'inspiration naturelle des méta-heuristiques est souvent un argument marketing et non scientifique

# Table des matières

## Recherche locale

-  1. Problèmes combinatoires de satisfaction et d'optimisation
-  2. Concepts et principes fondamentaux de la recherche locale
-  3. Formalisation de la recherche locale
-  4. Algorithme du *hill climbing*
-  5. Difficulté des minima locaux
-  6. Notion de voisinage connecté
-  7. Méthodes des redémarrages (*restarts*)
-  8. Algorithme du recuit simulé (*simulated annealing*)



# Synthèse des notions vues

## Problèmes combinatoires

**Satisfaction (CSP):** trouver une solution satisfaisant un ensemble de contraintes

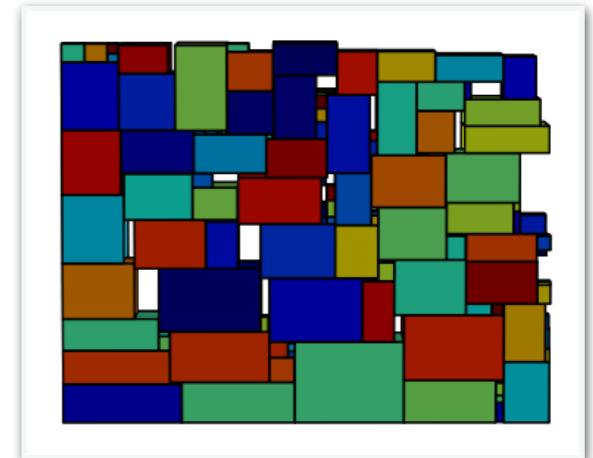
**Optimisation (COP):** CSP où on souhaite également optimiser une fonction objectif

**Une solution est un état, qui nous est inconnu, et non une séquence d'action**

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

Investissement	Coût (\$)	Revenu espéré dans 10 ans (\$)
A	200	1 000
B	200	1 000
C	200	1 000
D	500	10 000
E	500	10 000
F	800	13 000
G	300	7 000

Budget maximal de 1000 \$

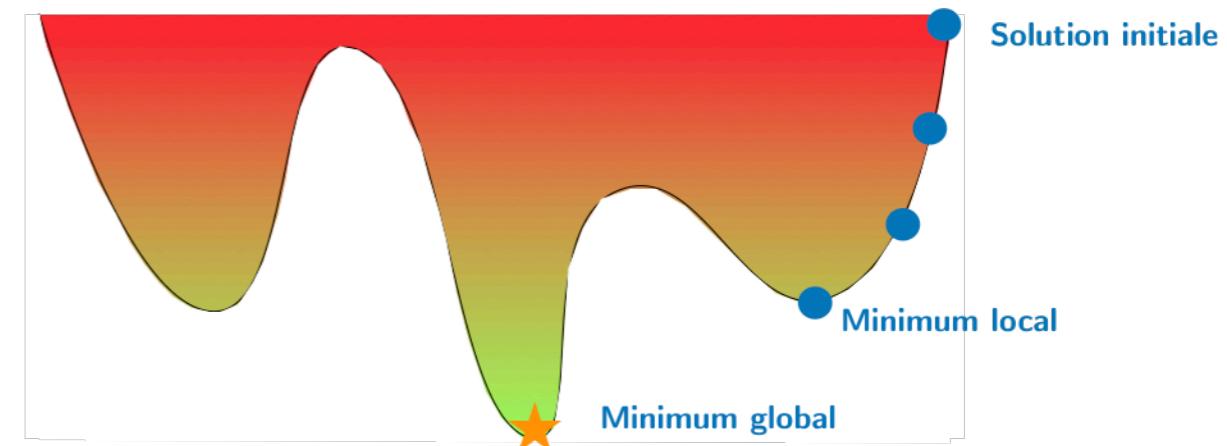


## Recherche locale

**Principe:** résolution en se déplaçant de solutions en solutions via des mouvements locaux

**Hill climbing:** sélection systématique du meilleur voisin

**Risque:** être bloqué dans un minimum local



## Mécanismes d'amélioration

**Idée 1:** utiliser un voisinage connecté

**Idée 2:** agrandir le voisinage

**Idée 3:** redémarrer aléatoirement la recherche

**Idée 4:** accepter de dégrader sa solution (p.e., *simulated annealing*)

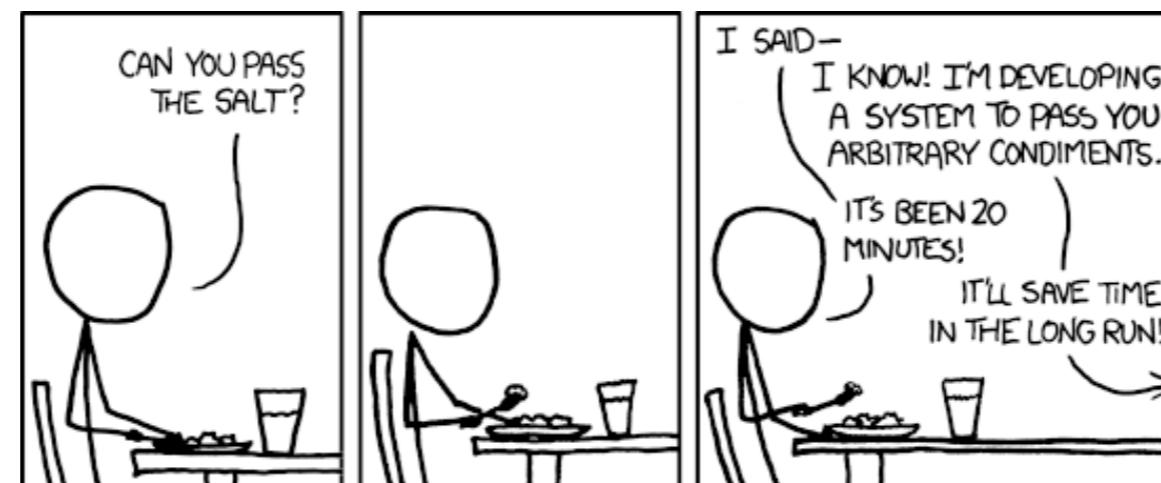
# Exemples de questions d'examen

## Théorie

1. Expliquer le fonctionnement d'un algorithme vu
2. Donner les avantages/inconvénients d'un voisinage large au lieu de restreint
3. Expliquer les principes généraux de la recherche locale

## Pratique

1. Savoir appliquer un algorithme de recherche locale
2. Proposer une résolution basée sur la recherche locale pour résoudre un problème (solution initiale, fonction de voisinage, fonction de sélection, etc.)
3. Peser le pour et le contre entre deux fonctions de voisnages





DALLE: *A queen climbing a mountain in an impressionist style*