

IFT 615 – Intelligence Artificielle

Recherche locale

Professeur: Froduald Kabanza

Assistants: D'Jeff Nkashama

Objectifs

- Comprendre:
 - ◆ La différence entre une recherche complète et une recherche locale.
 - ◆ La méthode *hill-climbing*.
 - ◆ La méthode *simulated-annealing*.
 - ◆ Les algorithmes génétiques.

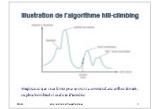
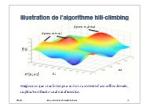
Motivations pour une recherche locale

- Rappel de quelques faits saillants de A*:
 - ◆ Un état final (let but) à atteindre est donné comme entrée.
 - ◆ La solution est un chemin et non juste l'état final.
 - ◆ Idéalement on veut un chemin optimal.
 - ◆ Exploration systématique de l'espace d'états: les états rencontrés sont stockés pour éviter de les revisiter.
- Pour certains types de problèmes impliquant une recherche dans un espace d'états, on peut avoir l'une ou l'autre des caractéristiques suivantes:
 - ◆ La solution recherchée est juste l'état optimal (ou proche) et non le chemin qui y mène.
 - ◆ Il y a une fonction objective à optimiser.
 - ◆ L'espace d'états est trop grand pour être enregistré.
- Pour ce genre de problèmes, une recherche locale peut être la meilleure approche.

Principe d'une recherche locale

- Une recherche locale garde juste certains états visités en mémoire:
 - ◆ Le cas le plus simple est *hill-climbing* qui garde juste **un état** (l'état courant) et l'améliore itérativement jusqu'à converger à une solution.
 - ◆ Le cas le plus élaboré est celui **des algorithmes génétiques** qui gardent **un ensemble d'états** (appelé *population*) et le fait évoluer jusqu'à obtenir une solution.
- En général, il y a une fonction objective à optimiser (maximiser ou minimiser)
 - ◆ Dans le cas de *hill-climbing*, elle permet de déterminer l'état successeur.
 - ◆ Dans le cas des algorithmes génétiques, on l'appelle la *fonction de fitness*. Elle intervient dans le calcul de l'ensemble des états successeurs de l'état courant.
- En général, une recherche locale ne garantie pas de solution optimale. Son attrait est surtout sa capacité de trouver une solution acceptable rapidement.

Méthode *Hill-Climbing*

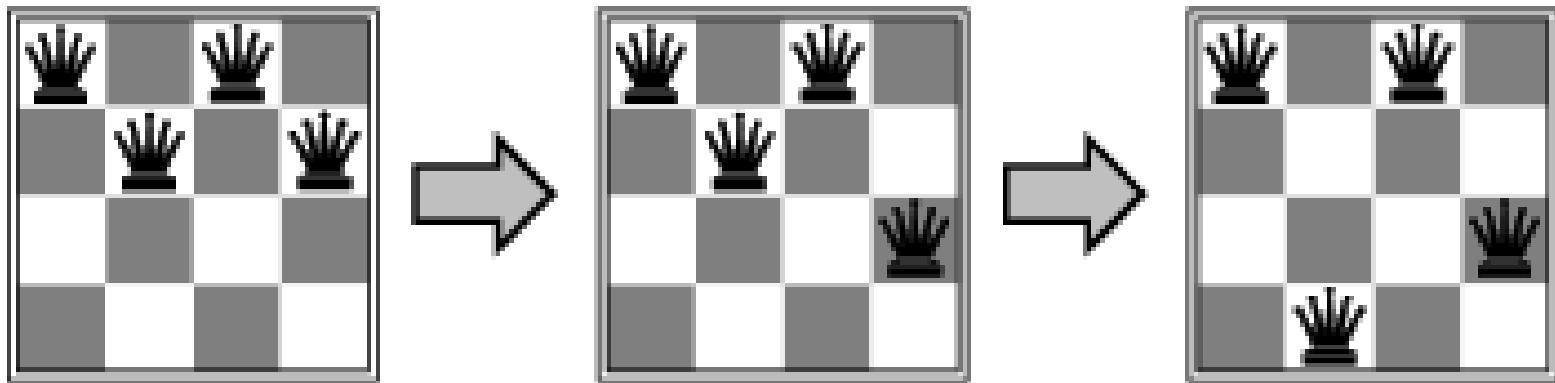
- Entrée :
 - ◆ État initial.
 - ◆ Fonction successeur
 - ◆ Fonction à optimiser:
 - » noté VALUE dans l'algorithme;
 - » parfois noté h aussi.
 - Méthode
 - ◆ Le nœud courant est initialisé à l'état initial.
 - ◆ Itérativement, le nœud courant est comparé à ses successeurs immédiats.
 - » Le meilleur voisin immédiat et ayant la plus grande valeur (selon VALUE) que le nœud courant, devient le nœud courant.
 - » Si un tel voisin n'existe pas, on arrête et on retourne le nœud courant comme solution.
- 
- 

Algorithme *Hill-Climbing*

```
function HILL-CLIMBING(problem) returns a state that is a local maximum  
    current  $\leftarrow$  problem.INITIAL  
    while true do  
        neighbor  $\leftarrow$  a highest-valued successor state of current  
        if VALUE(neighbor)  $\leq$  VALUE(current) then return current  
        current  $\leftarrow$  neighbor
```

Exemple: N-Queen

- Problème: Placer n reines sur un échiquier de taille $n \times n$ de sorte que deux reines ne s'attaquent mutuellement:
 - ◆ C-à-d., jamais deux reines sur la même diagonale, ligne ou colonne.



- Avec $N=4$: 256 configurations.
- $N=8$: 16 777 216
- $N=16$: 18,446,744,073,709,551,616 configurations

Hill-Climbing avec 8 reines

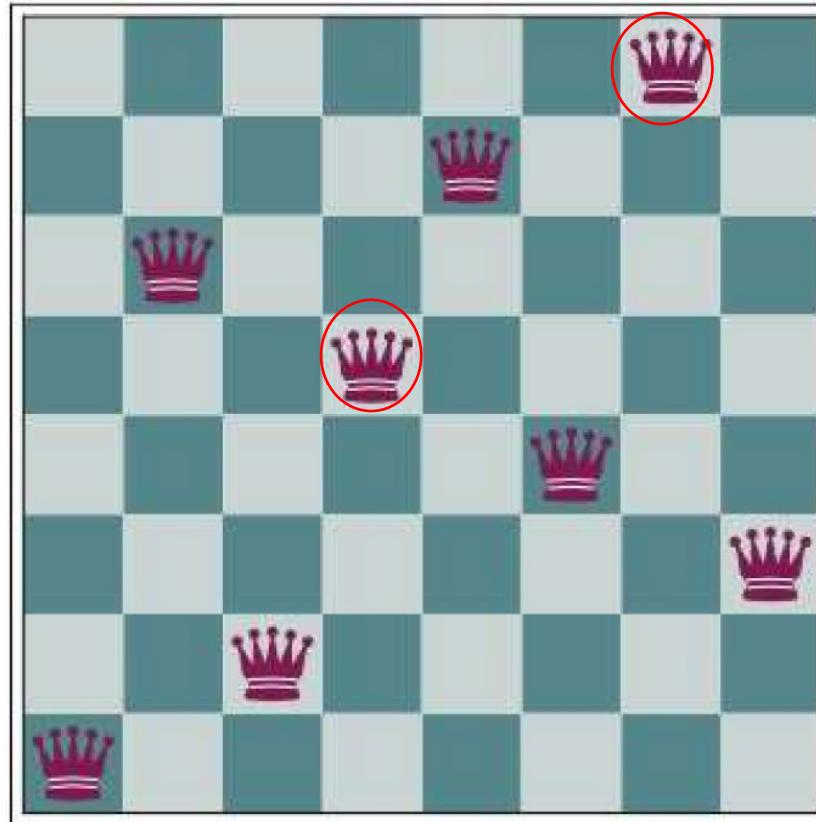
- h (VALUE): nombre de paires de reines qui s'attaquent mutuellement directement ou indirectement.
- On veut le minimiser.

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
14	14	17	15	13	14	16	16
17	14	16	18	15	15	15	15
18	14	15	15	15	14	15	16
14	14	13	17	12	14	12	18

- h pour l'état affiché: 17
- Encadrés: les meilleurs successeurs

Hill-Climbing avec 8 reines

- Un exemple de minimum local avec $h(n)=1$



Méthode *simulated annealing* (recuit simulé)

- C'est une amélioration de l'algorithme *hill-climbing* pour **minimiser le risque d'être piégé dans des maxima/minima locaux**
 - ◆ au lieu de regarder le meilleur voisin immédiat du nœud courant, **avec une certaine probabilité on va regarder un moins bon voisin immédiat**
 - » on espère ainsi s'échapper des optima locaux
 - ◆ au début de la recherche, la **probabilité de prendre un moins bon voisin** est plus élevée et **diminue graduellement**
- Le nombre d'itérations et la diminution des probabilités sont définis à l'aide d'un schéma (*schedule*) de « températures », en ordre décroissant
 - ◆ ex.: schéma $[2^{-0}, 2^{-1}, 2^{-2}, 2^{-3}, \dots, 2^{-99}]$, pour un total de 100 itérations
 - ◆ la meilleure définition du schéma va varier d'un problème à l'autre

Algorithme Simulated Annealing

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current  $\leftarrow$  problem.INITIAL
  for  $t = 1$  to  $\infty$  do
     $T \leftarrow \text{schedule}(t)$ 
    if  $T = 0$  then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

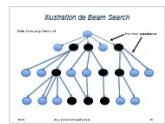
plus T est petit,
plus $e^{-\Delta E/T}$ est petit

Tabu search

- L'algorithme *simulated annealing* minimise le risque d'être piégé dans des optima locaux
- Par contre, il n'élimine pas **la possibilité d'osciller indéfiniment** en revenant à un noeud antérieurement visité
- **Idée:** On pourrait **enregistrer les nœuds visités**
 - ◆ on revient à A* et approches similaires!
 - ◆ mais c'est impraticable si l'espace d'états est trop grand
- **L'algorithme *tabu search*** (recherche taboue) enregistre seulement les k derniers nœuds visités
 - ◆ l'**ensemble taboue** est l'ensemble contenant les k noeuds
 - ◆ le paramètre k est choisi empiriquement
 - ◆ cela n'élimine pas les oscillations, mais les réduit
 - ◆ il existe en fait plusieurs autres façon de construire l'ensemble tabou...

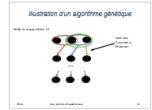
Beam search

- **Idée:** plutôt que maintenir un seul noeud solution n , en pourrait maintenir un ensemble de k noeuds différents
 1. on commence avec un ensemble de k noeuds choisis aléatoirement
 2. à chaque itération, tous les successeurs des k noeuds sont générés
 3. on choisit les k meilleurs parmi ces noeuds et on recommence
 - Cet algorithme est appelé ***local beam search*** (exploration locale par faisceau)
 - ◆ à ne pas confondre avec ***tabu search***
 - Variante ***stochastic beam search*** : plutôt que prendre les k meilleurs, on assigne une probabilité de choisir chaque noeud, même s'il n'est pas parmi les k meilleurs (comme dans ***simulated annealing***)



Algorithmes génétiques

- Idée très similaire à *stochastic beam-search*. Rappel de *beam search*:
 1. On commence avec un ensemble n d'états choisis aléatoirement.
 2. À chaque itération, tous les successeurs des n états sont générés.
 3. Si un d'eux satisfait le but, on arrête.
 4. Sinon on choisit les n meilleurs (local) ou au aléatoirement (stochastic) et on recommence.
- Algorithme génétique
 - ◆ On commence aussi avec un **ensemble n d'états choisis aléatoirement**. Cet ensemble est appelé **une population**.
 - ◆ Un successeur est généré en combinant deux parents.
 - ◆ Un état est représenté par un mot (chaîne) sur un alphabet (souvent l'alphabet binaire).
 - ◆ La **fonction** d'évaluation est appelée *fondction de fitness* (fonction d'adaptabilité, de survie).
 - ◆ La prochaine génération est produite par *sélection, croisement et mutation*.



Algorithmes génétiques

- Les algorithmes génétiques sont inspirés du processus de l'évolution naturelle des espèces:
 - ◆ Après tout l'intelligence humaine est le résultat d'un processus d'évolution sur des millions d'années :
 - » Théorie de l'évolution (Darwin, 1858)
 - » Théorie de la sélection naturelle (Weismann)
 - » Concepts de génétiques (Mendel)
 - ◆ La simulation de l'évolution n'a pas besoin de durer des millions d'années sur un ordinateur.

Algorithmes génétiques

- On représente l'espace des solutions d'un problème à résoudre par une *population* (ensemble de *chromosomes*).
 - ◆ Un *chromosome* est une chaîne de bits (*gènes*) de taille fixe.
 - ◆ Par exemple : 101101001
- Une population génère des enfants par un ensemble de procédures simples qui manipulent les chromosomes
 - ◆ Croisement de parents
 - ◆ Mutation d'un enfant généré
- Les parents à croiser sont choisis en fonction de leur *adaptabilité (fitness)* déterminée par une fonction d'adaptabilité donnée, $f(x)$.

Algorithmes génétiques

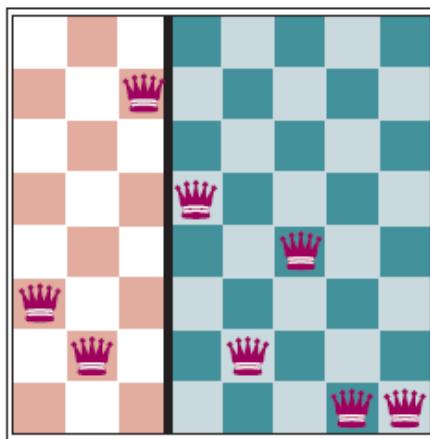
1. Générer aléatoirement une population de N chromosomes.
2. Calculer la valeur d'adaptabilité (*fitness*) de chaque chromosome x .
3. Créer une nouvelle population de taille N .
 - 3.1 Sélectionnant 2 parents chromosomes (chaque parent est sélectionné avec une probabilité proportionnelle à son adaptabilité) et en les croisant avec une certaine probabilité.
 - 3.2. Mutant les deux enfants obtenus avec une certaine probabilité.
 - 3.3 Plaçant les enfants dans la nouvelle population.
 - 3.4 Répéter à partir de l'étape 3.1 jusqu'à avoir une population de taille N .
4. Si la population satisfait le critère d'arrêt, arrêter.
Sinon, recommencer à l'étape 2.

Algorithmes génétiques

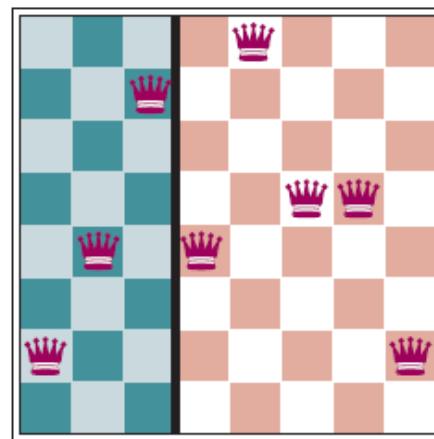
```
function GENETIC-ALGORITHM(population, fitness) returns an individual
repeat
    weights  $\leftarrow$  WEIGHTED-BY(population, fitness)
    population2  $\leftarrow$  empty list
    for i = 1 to SIZE(population) do
        parent1, parent2  $\leftarrow$  WEIGHTED-RANDOM-CHOICES(population, weights, 2)
        child  $\leftarrow$  REPRODUCE(parent1, parent2)
        if (small random probability) then child  $\leftarrow$  MUTATE(child)
        add child to population2
    population  $\leftarrow$  population2
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to fitness

function REPRODUCE(parent1, parent2) returns an individual
    n  $\leftarrow$  LENGTH(parent1)
    c  $\leftarrow$  random number from 1 to n
    return APPEND(SUBSTRING(parent1, 1, c), SUBSTRING(parent2, c + 1, n))
```

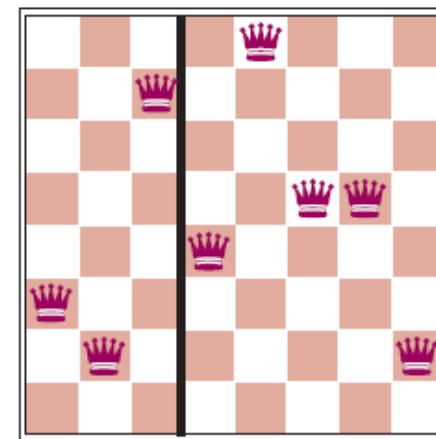
Croisement: exemple avec 8 reines



+



=



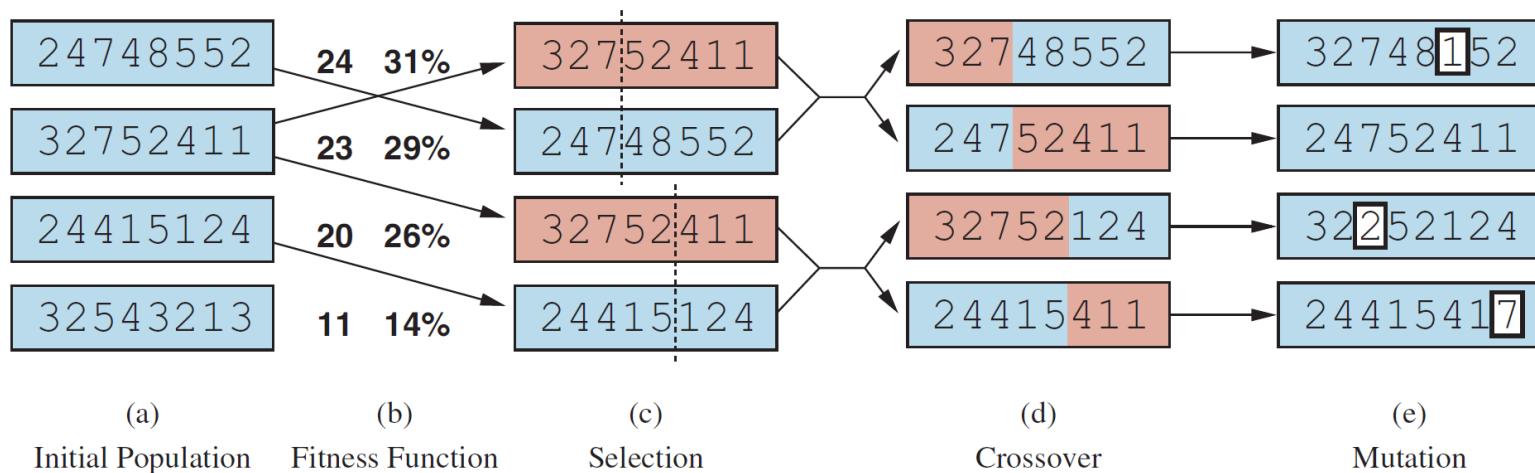
67247588

75251447

= 67251447

Exemple avec 8 reines

- Fonction de fitness: nombre de pairs de reines qui ne s'attaquent pas (min = 0, max = $(8 \times 7)/2 = 28$)
- Pourcentage de fitness (c-à-d., probabilité de sélection du chromosome):
 - ◆ $24/(24+23+20+11) = 31\%$
 - ◆ $23/(24+23+20+11) = 29\%$
 - ◆ $20/(24+23+20+11) = 26\%$
 - ◆ $11/(24+23+20+11) = 14\%$



Autre Exemple

[Michael Negnevitsky. Artificial Intelligence. Addison-Wesley, 2002. Page 222.]

- Calculer le maximum de la fonction $f(x) = 15x - x^2$
- Supposons x entre $[0, 15]$:
 - ◆ on a besoin de seulement 4 bits pour représenter la population.

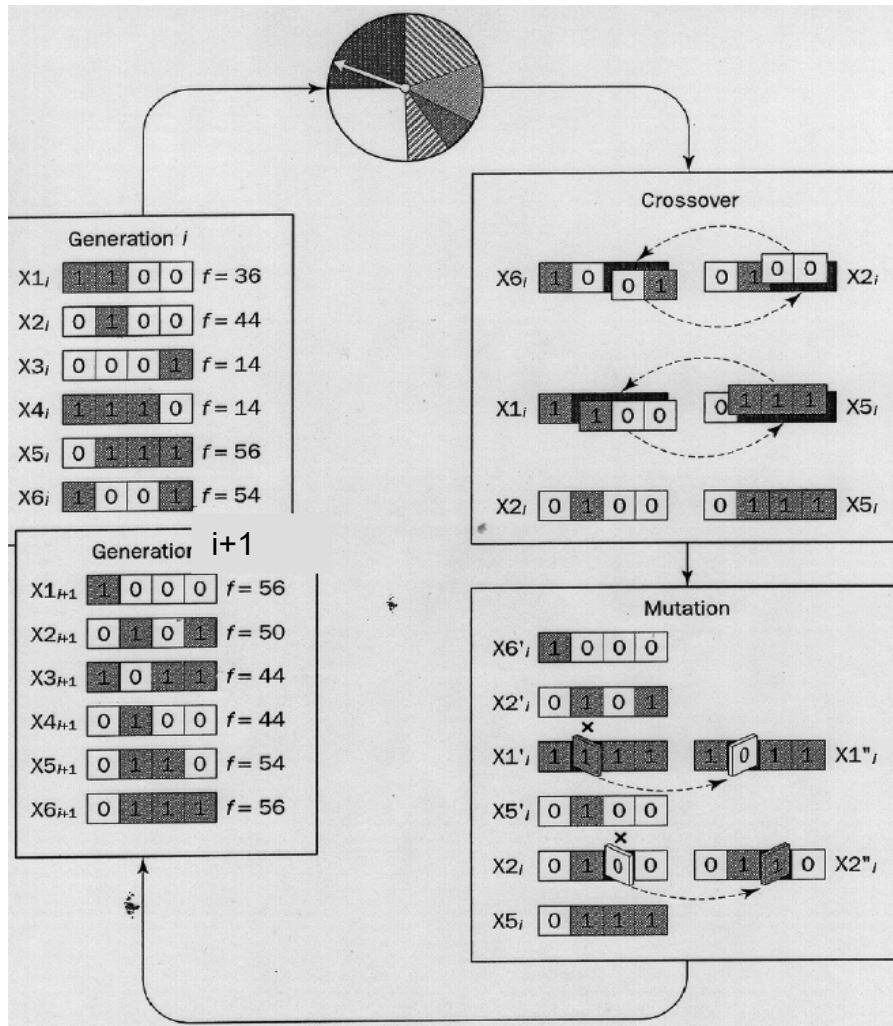
Integer	Binary code	Integer	Binary code	Integer	Binary code
1	0 0 0 1	6	0 1 1 0	11	1 0 1 1
2	0 0 1 0	7	0 1 1 1	12	1 1 0 0
3	0 0 1 1	8	1 0 0 0	13	1 1 0 1
4	0 1 0 0	9	1 0 0 1	14	1 1 1 0
5	0 1 0 1	10	1 0 1 0	15	1 1 1 1

Autre Exemple (suite)

- Fixons la taille de la population à 6.
- La probabilité de croisement à 0.7
- Et la probabilité de mutation à 0.001.
- La fonction d'adaptabilité à $f(x)=15x - x^2$.
- L'algorithme génétique initialise les 6 chromosomes de la population en les choisissant au hasard.

Chromosome label	Chromosome string	Decoded integer	Chromosome fitness	Fitness ratio, %
X1	1 1 0 0	12	36	16.5
X2	0 1 0 0	4	44	20.2
X3	0 0 0 1	1	14	6.4
X4	1 1 1 0	14	14	6.4
X5	0 1 1 1	7	56	25.7
X6	1 0 0 1	9	54	24.8

Autre Exemple (Illustration des étapes)



- Critère d'arrêt : Maximum de la moyenne de fitness de la population.
- Détectée au point où la moyenne d'adaptation commence à décroître.
- Problème de minima locaux.

Programmation génétique

Même principes que les algorithmes génétiques sauf que les populations sont des programmes au lieu des chaînes de bits.

[Michael Negnevitsky. Artificial Intelligence. Addison-Wesley, 2002. Page 247.]

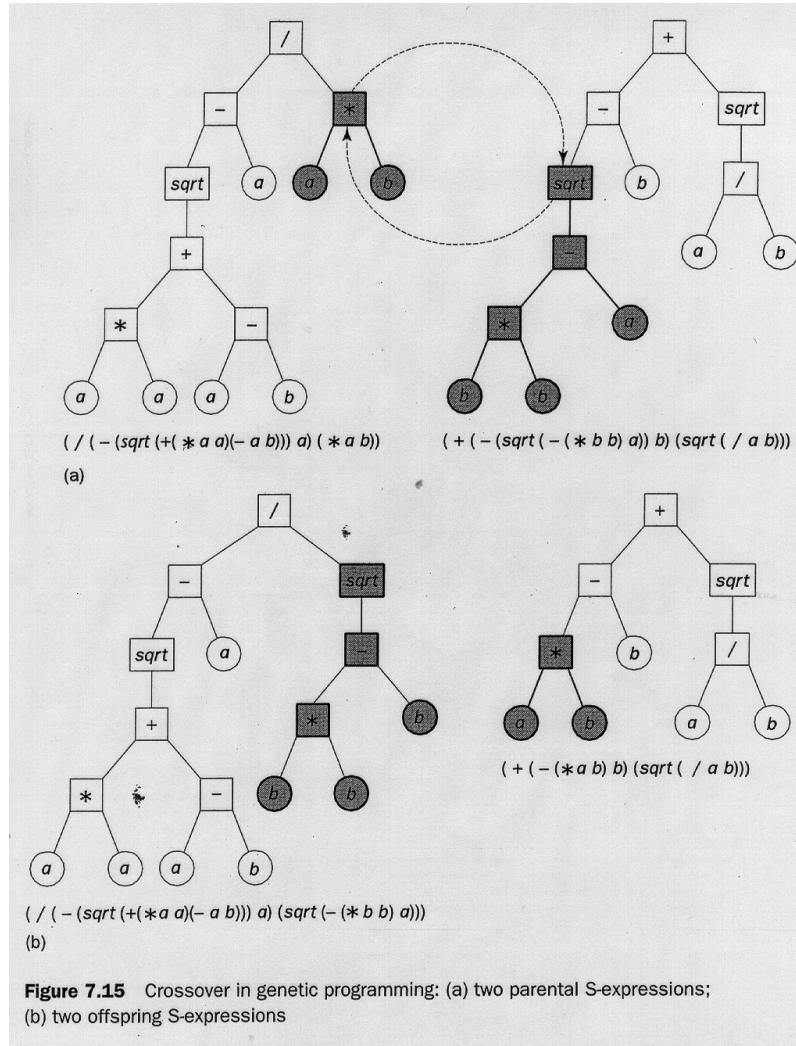


Figure 7.15 Crossover in genetic programming: (a) two parental S-expressions; (b) two offspring S-expressions

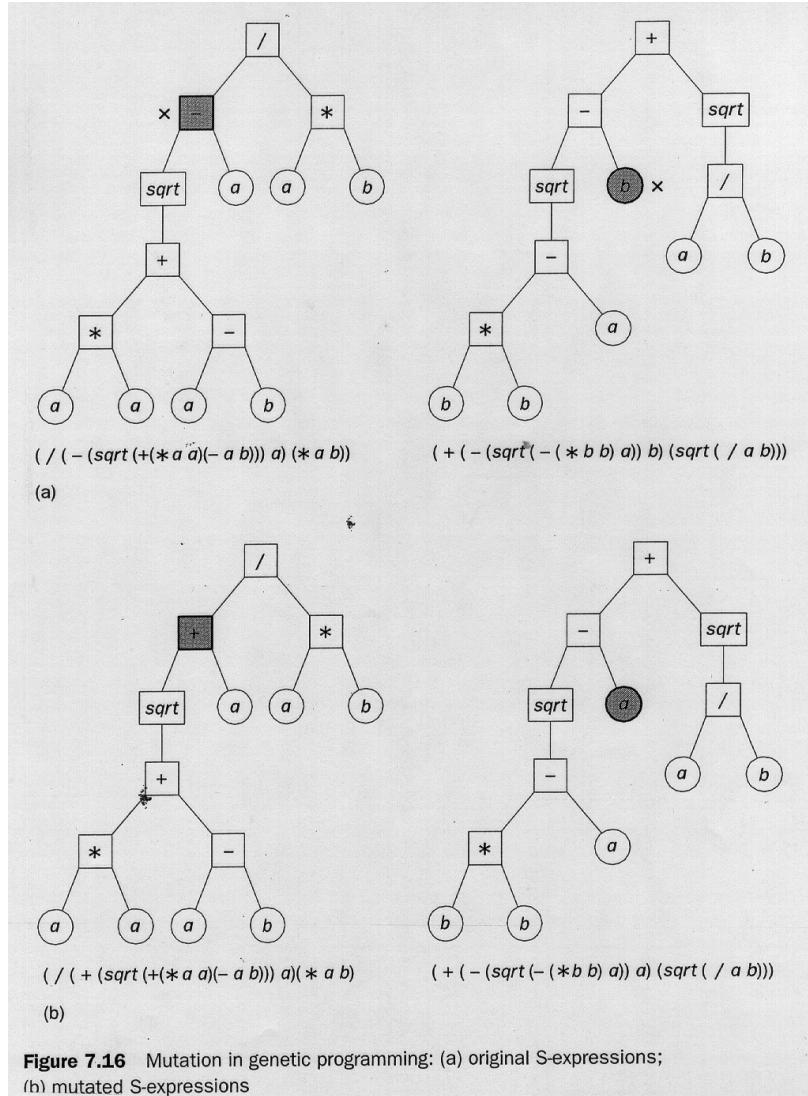


Figure 7.16 Mutation in genetic programming: (a) original S-expressions; (b) mutated S-expressions

Applications

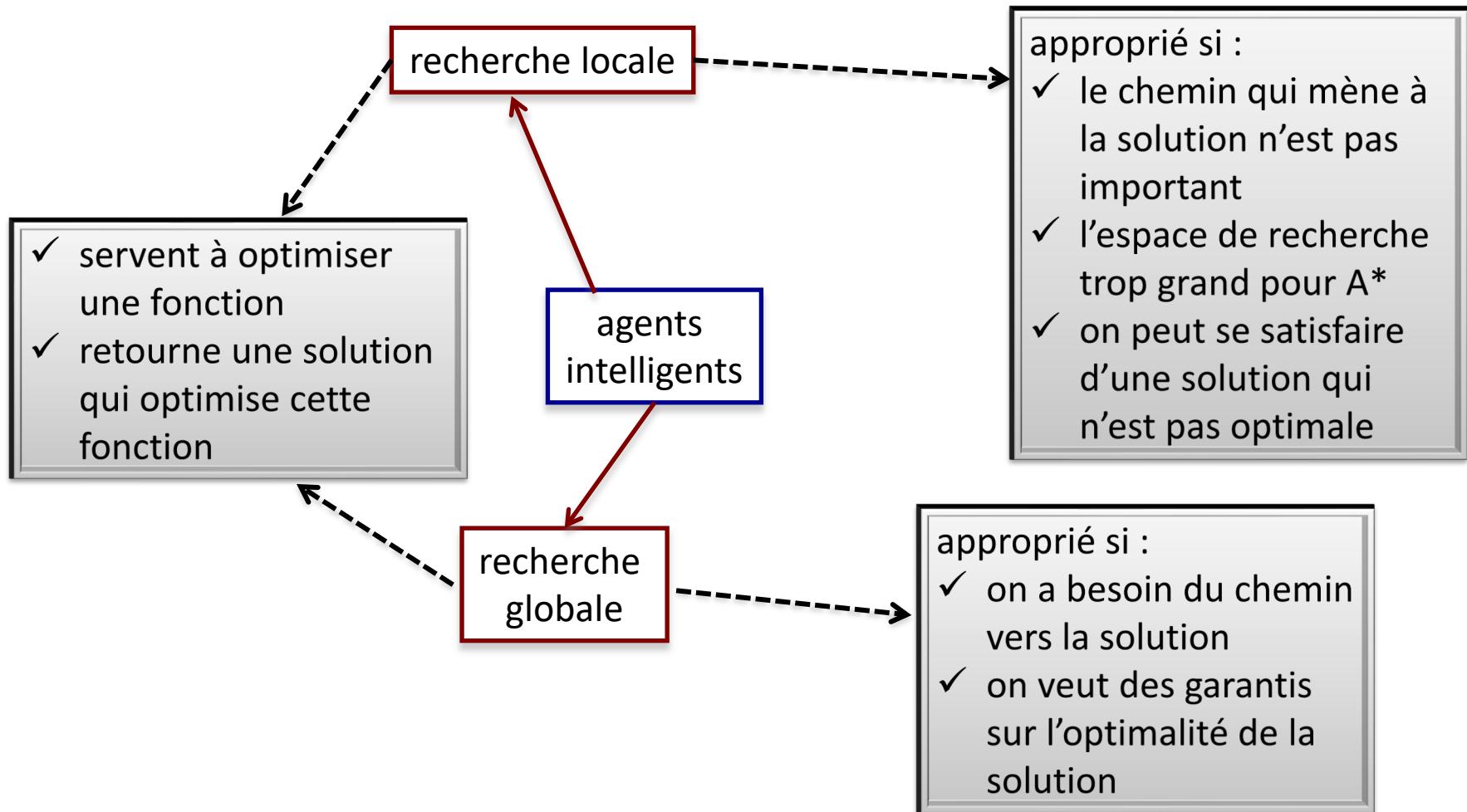
- Problèmes d'optimisation
- Optimization des hyper-paramètres d'un réseau de neurones
 - ◆ https://en.wikipedia.org/wiki/Hyperparameter_optimization
- Entrainement de réseaux de neurones combiné avec l'optimization des hyper-paramètres (opulation-based training)
 - ◆ https://en.wikipedia.org/wiki/Hyperparameter_optimization
 - ◆ Jaderberg et al. Population Based Training of Neural Networks.
<https://arxiv.org/abs/1711.09846>
 - ◆ <https://www.cs.utexas.edu/users/nn/pages/research/neatdemo.html>

Conclusion

- La recherche locale est parfois une alternative plus intéressante que la recherche heuristique
- J'ai ignoré le cas où on a également une fonction $goal(n)$
 - ◆ dans ce cas, lorsqu'on change la valeur de n , on arrête aussitôt que $goal(n)$ est vrai
 - » ex.: $goal(n)$ est vrai si n est un optimum global de $F(n)$
- Il y a d'autres algorithmes de recherche locale que je n'ai pas couvert. Par exemple, l'optimisation bayésienne.
- La recherche locale est utilisée entre autre pour l'optimisation des hyperparamètres dans l'apprentissage automatique

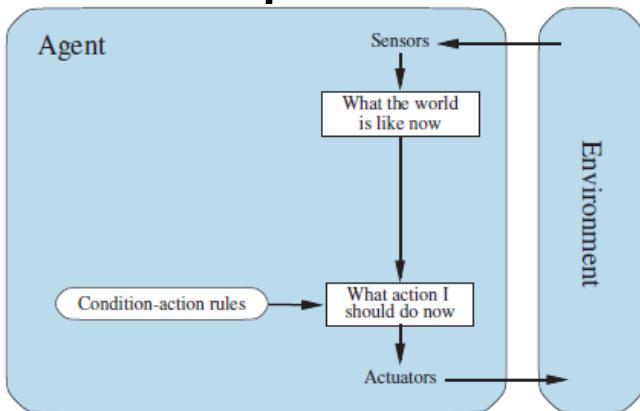
Recherche local vs Recherche global

Concepts et algorithmes

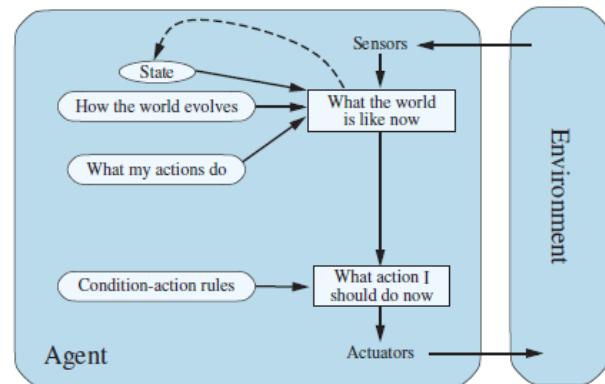


Recherche locale pour quel type d'agents?

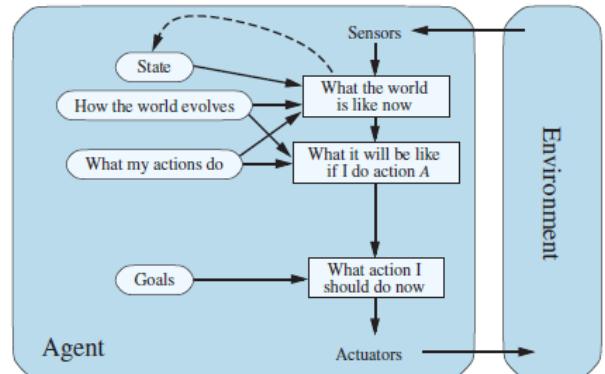
Simple reflex



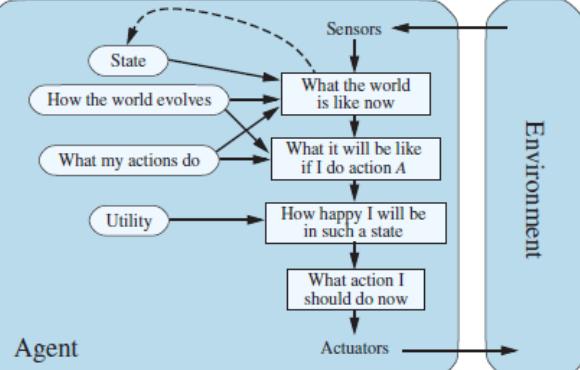
Model-based reflex



Goal-based



Utility-based



Vous devriez être capable de...

- Décrire ce qu'est la recherche locale en général
- Décrire les algorithmes :
 - ◆ *hill-climbing*
 - ◆ *simulated annealing*
 - ◆ *Beam search*
 - ◆ algorithme génétique
- Savoir simuler ces algorithmes
- Connaître leurs propriétés (avantages vs. désavantages)

Sujets couverts par le cours

Concepts et algorithmes

Applications

- ✓ Apprentissage supervisé
- ✓ Apprentissage par renforcement

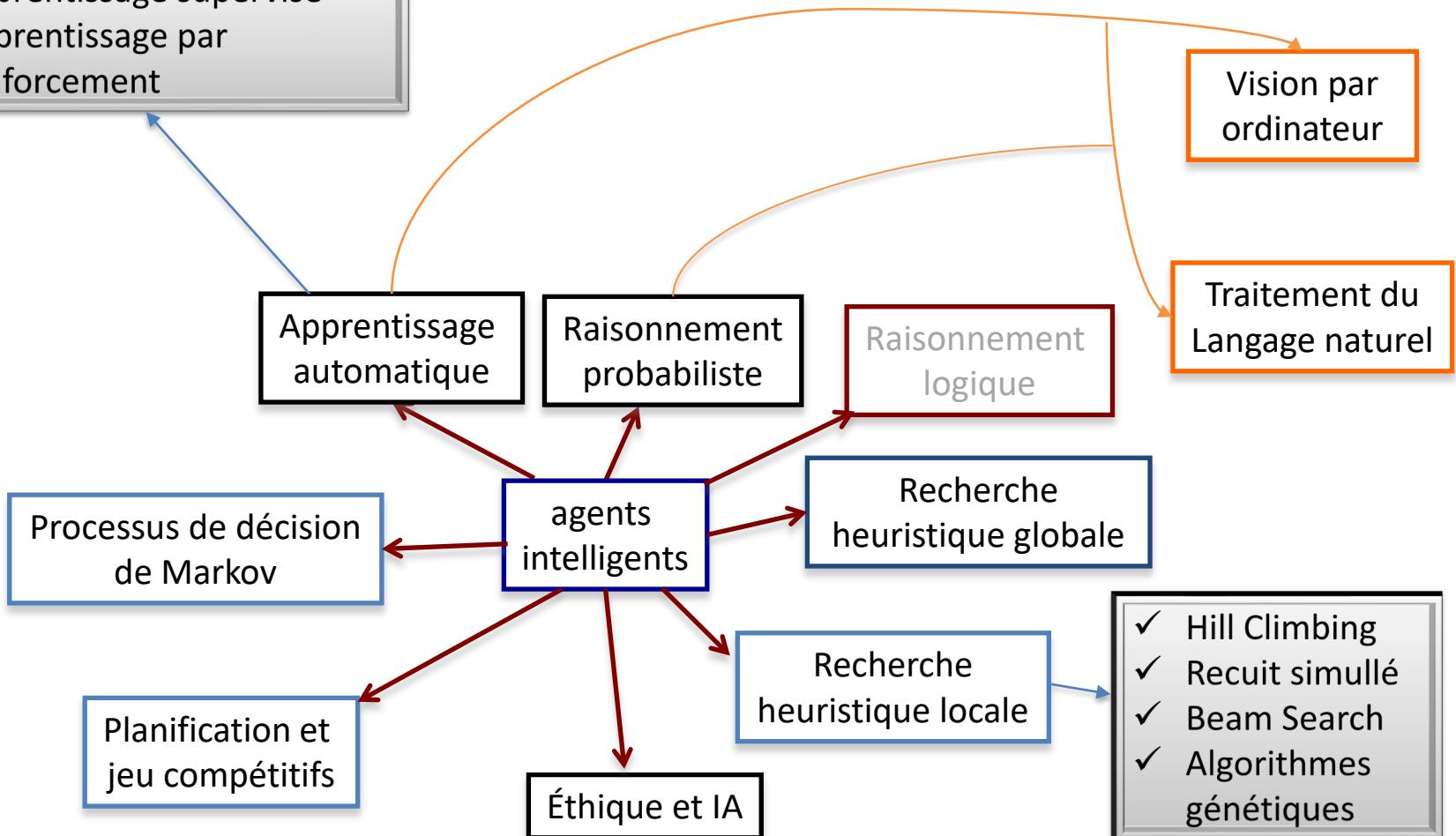


Illustration de Beam Search

Taille de la population =3

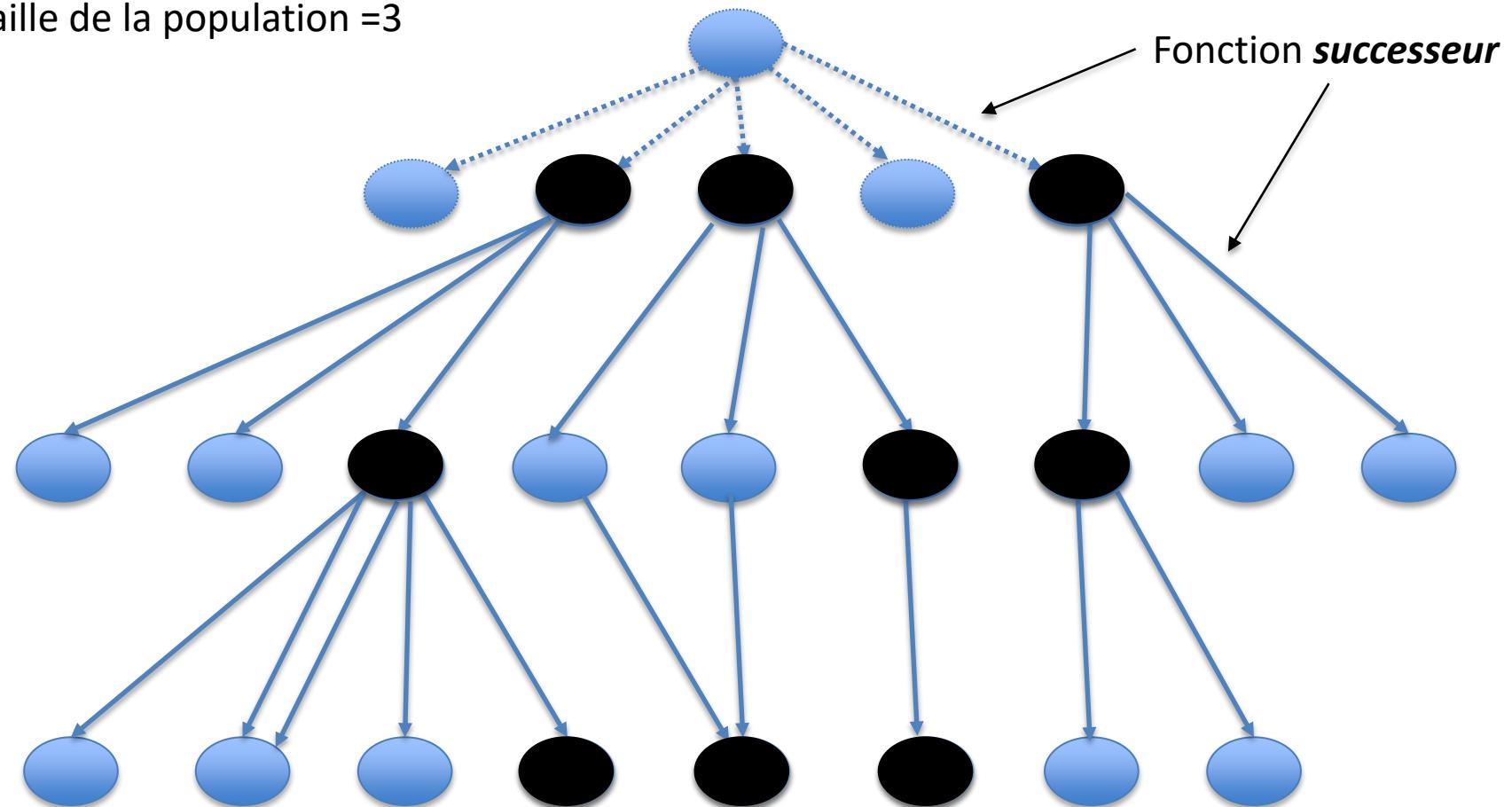


Illustration d'un algorithme génétique

Taille de la population =3

