INF4230 – Intelligence Artificielle

Recherche locale

Hiver 2017

Objectifs

- Comprendre :
  - La pertinence de la **recherche locale** par rapport à la recherche globale générant un chemin.
  - La méthode de l'**escalade** (*hill-climbing*).
  - La méthode du **recuit simulé** (*simulated-annealing*).
  - La méthode d'**exploration en faisceau** (*beam-search*).
  - Les **algorithmes génétiques**.

Rappel recherche globale pour extraire un chemin

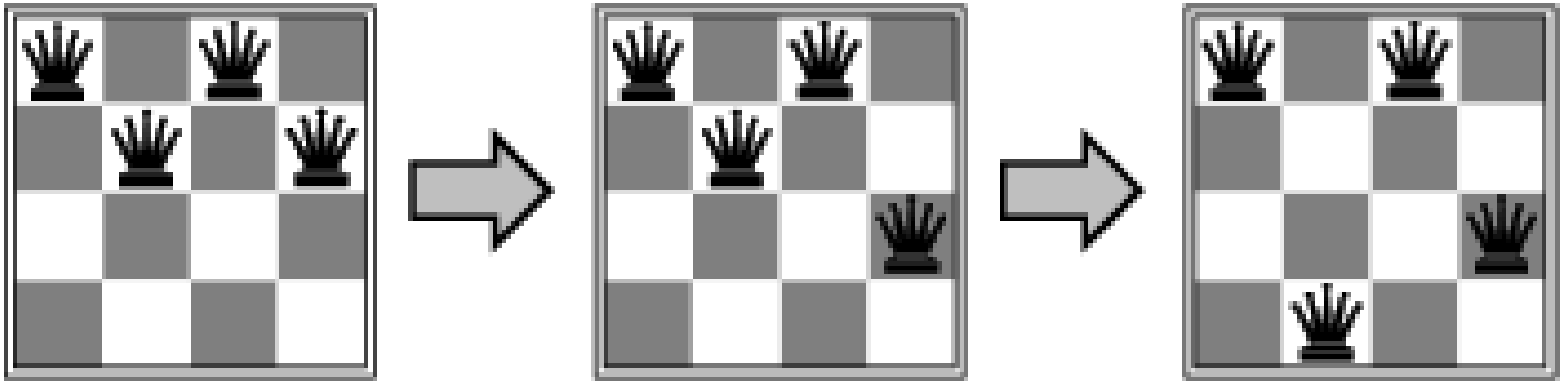
- Problème = État initial + But.
- Solution = **chemin** reliant état initial est un étant satisfaisant le but.
- Algorithmes = recherche d'un chemin dans un graphe.
- Généralement gourmand en mémoire pour mémoriser les états à visiter (*open*) et ceux visités (*closed*).

Recherche locale

- Pour certains problèmes, le **chemin** vers le but n'est pas pertinent ou utile .
- La **solution est seulement un état (nœud)**.
- Espace d'états = **espace de configurations**.
- Problème = trouver une **configuration satisfaisant des contraintes** (ex: *n*-reines / *n*-queens).
- Optionnel : une **Fonction objectif** à optimiser, exclusivement basée sur les caractéristiques de l'état. Le chemin n'est pas considéré par cette fonction.

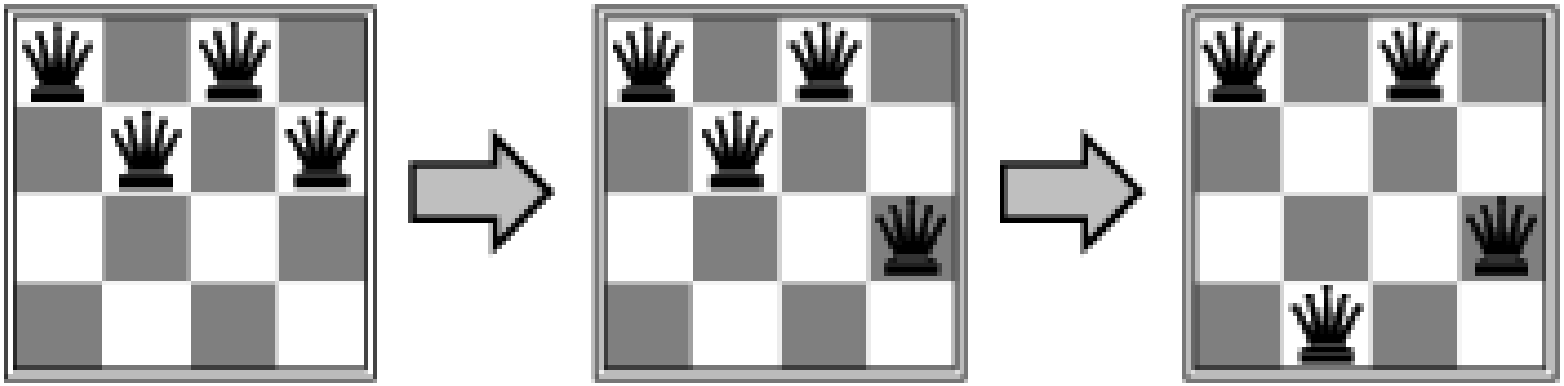
Exemple : *n*-reines (*n-queens*)

- Placer *n* reines sur un plateau de *n* x *n* de façon à ce qu'il n'y ait aucune paire de reines en position d'attaque (sur une même ligne, colonne, ou diagonale).
- Ici, le chemin n'est pas important. Ce qui nous intéresse, c'est de trouver une configuration.
- Le concept d'état initial peut ne pas exister.



Exercice : modélisation de l'espace d'états pour le problème des *n*-reines

- Comment modéliser/représenter un état ?
  - Grille (matrice) *n* x *n* indiquant l'état d'occupation de chaque case?
  - Vecteur des positions (ligne,colonne) des reines?
  - **Autres**?
  - >>>>> **Réponse donnée en classe** <<<<



Principe d'une recherche locale

- Une recherche locale garde seulement certains états visités en mémoire:
  - La méthode la plus simple est **l'escalade** (***hill-climbing***). Elle garde **un seul état** («l'état courant») et l'améliore itérativement jusqu'à converger à une solution.
  - Une méthode plus élaborée les **algorithmes génétiques**. Elle gardent **un ensemble d'états** (appelé **individus ou population**) et le fait évoluer jusqu'à obtenir une solution.
- Généralement, une fonction objective doit être optimisée (maximisée ou minimisée)
  - Dans **l'escalade** (***hill-climbing***), cette fonction détermine l'état successeur.
  - Dans les algorithmes génétiques, cette fonction est appelée **fonction de *fitness***. Elle intervient dans le calcul de l'ensemble des états successeurs de l'ensemble 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 de l'escalade (*hill-climbing*)

- Entrée
  - État initial (tiré de l'environnement ou aléatoire).
  - Fonction à optimiser :
    - notée *VALUE* dans l'algorithme;
    - parfois aussi notée h ou h(n).
- Méthode
  - Le nœud **courant** est initialisé à l'état initial.
  - Itérativement, le nœud courant est comparé à ses successeurs (voisins) immédiats.
    - Le meilleur successeur immédiat (celui qui a la plus grande valeur *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 de l'escalade (*hill-climbing*)

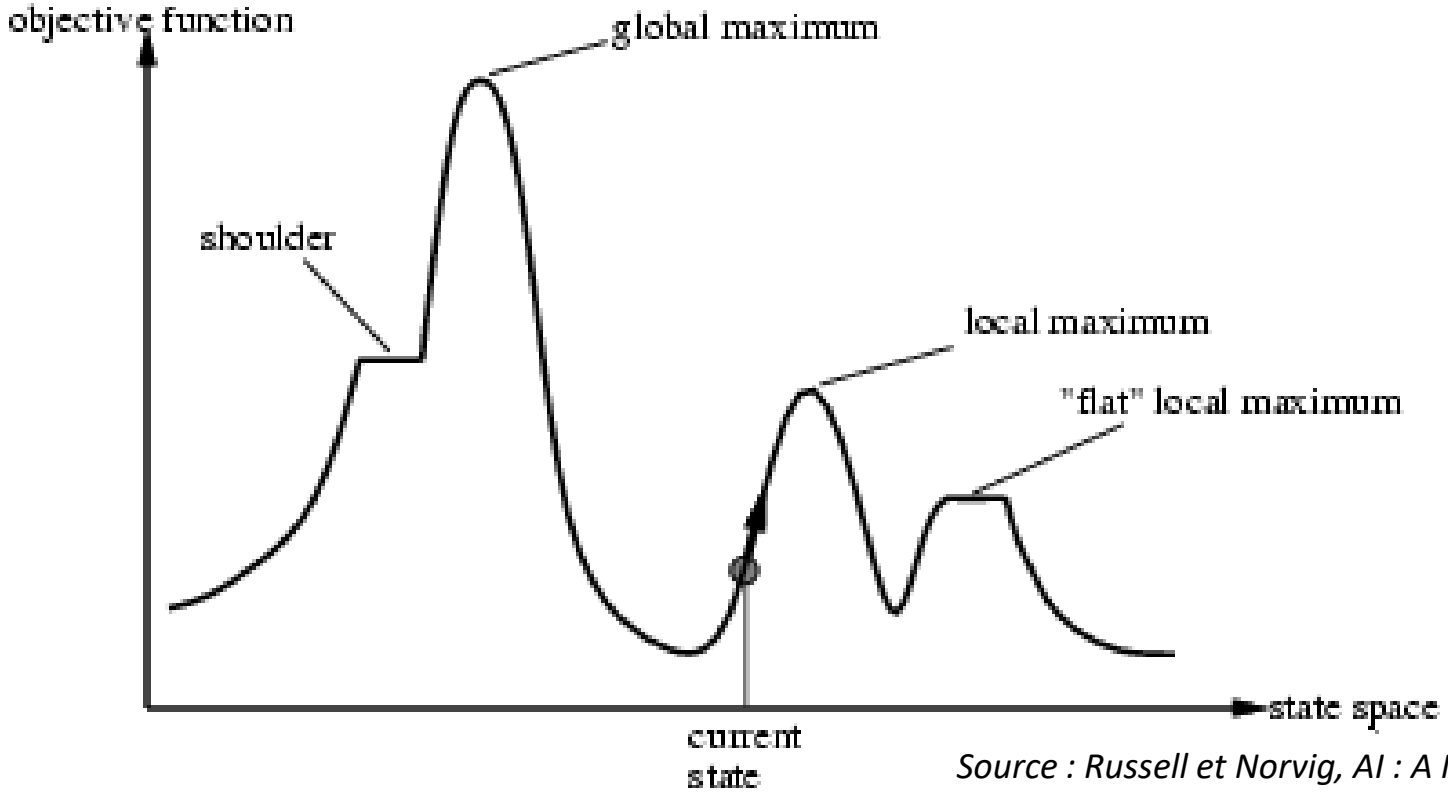
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Source : Russell et Norvig, AI : A Modern Approach.

Illustration de l'escalade (*hill-climbing*)

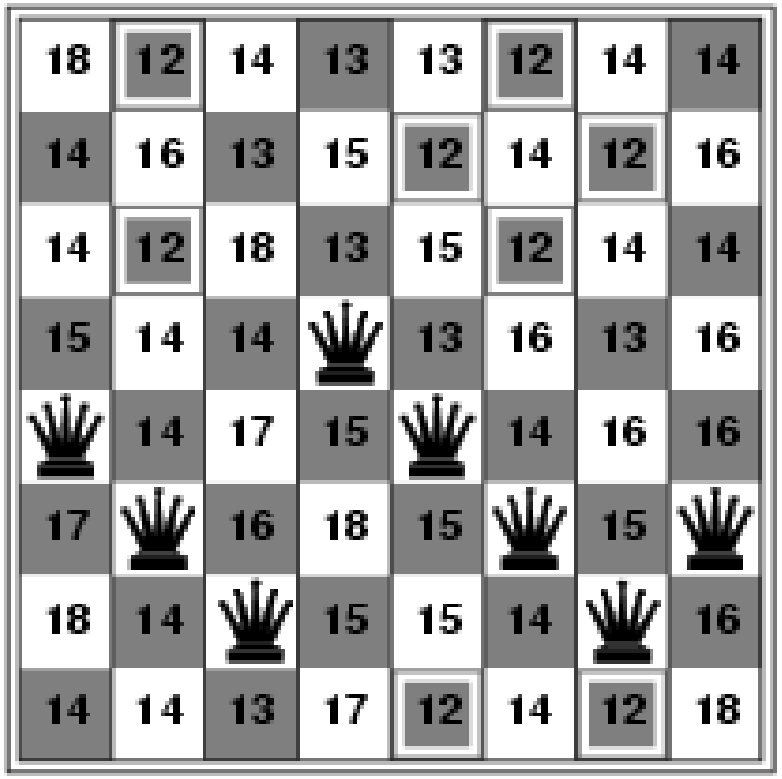
- Limite : on peut être piégé dans un optimum (maximum / minimum) local.



Source : Russell et Norvig, AI : A Modern Approach.

L'escalade appliquée aux *n*-reines.

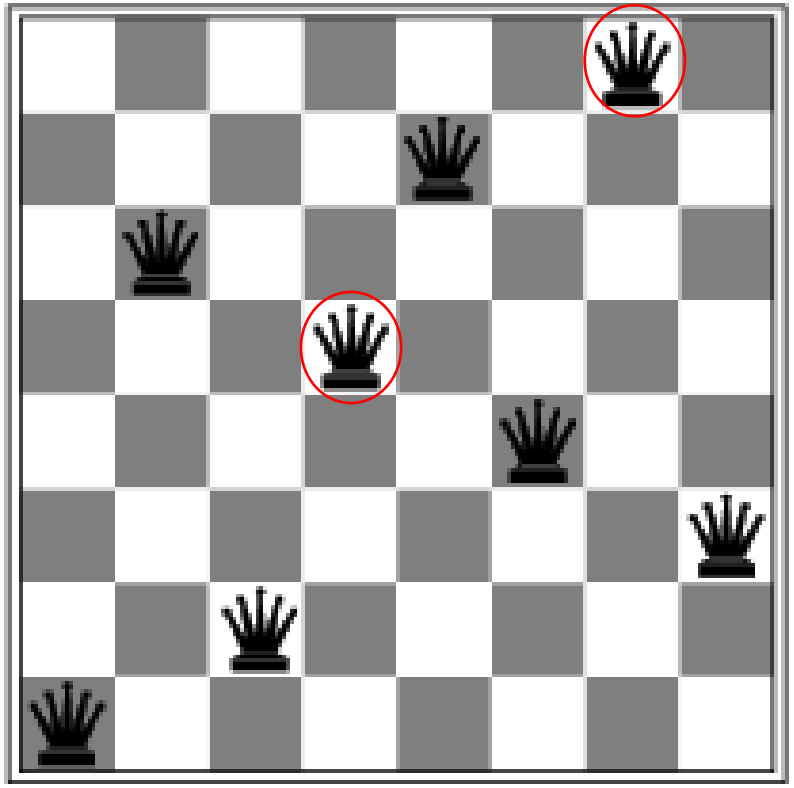
- Fonction objectif : h(*n*) = nombre de reines qui s'attaquent dans l'état *n*.
- On veut minimiser h(.).
- Idéalement, trouver h()=0
- h(état affiché ci-droit) = 17
  - Nombres = h des états succs.
  - Encadrés = meilleurs succs.



Source : Russell et Norvig, AI : A Modern Approach.

Exemple de minimum local avec *n*-reines

- h(état ci-droit)=1
- Aucun meilleur successeur n'existe.
- Nous sommes donc «piégés» dans un minimum local.
- Existe-t-il une meilleure solution ou même une avec h(n)=0 ?
  - L'algorithme *hill-climbing* ne peut pas nous le dire.



Source : Russell et Norvig, AI : A Modern Approach.

Variantes de l'escalade

- **L'escalade stochastique**.
  - Choisi au hasard l'état suivant parmi les successeurs ascendant (pas nécessairement le meilleur).
- **L'escalade du premier choix.**
  - Utile quand il y a un grand nombre de successeurs.
  - Génère les successeurs immédiats aléatoirement et procède à l'escalade dès que l'un d'eux est strictement meilleur que l'état courant.
- **L'escalade avec reprise aléatoire**.
  - Quand on a trouvé un optimum local, on relance la recherche avec un nouvel état tiré aléatoirement.
  - Probabilistiquement complet.

Méthode du recuit simulé (*Simulated Annealing*)

- Amélioration de l'escalade réduisant le risque d'être piégé dans des optimums locaux.
- Méthode:
  - Au lieu de regarder le meilleur voisin immédiat du nœud courant, on a une certaine probabilité d'aller vers un moins bon voisin immédiat.
  - On espère ainsi s'échapper des optimums locaux.
  - Au début de la recherche, la probabilité de prendre un moins bon voisin est plus élevée et diminue graduellement (exponentiellement en fonction de la mauvaise qualité du nœud choisi).
- Méthode inspirée d'un procédé utilisé en métallurgie pour durcir les matériaux (en métal ou en verre): le procédé alterne des cycles de refroidissement lent et de réchauffage (recuit) qui tendent à minimiser l'énergie du matériau.

Algorithme du recuit simulé

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(INITIAL-STATE[problem])
   for t ← 1 to ∞ do
       T ← schedule[t]
       if T = 0 then return current
       next ← a randomly selected successor of current
       ΔE ← VALUE[next] – VALUE[current]
       if ΔE > 0 then current ← next
       else current ← next only with probability e^(Δ E/T)
```

Source : Russell et Norvig, AI : A Modern Approach.

Propriétés du recuit simulé

- On peut prouver que si *T* diminue très lentement, alors l'algorithme du recuit simulé va trouver un optimum globale (la solution optimale) avec une probabilité très proche de 1 (probabilistiquement complet).
- La probabilité de converger vers la solution optimale peut converger vers 1 quand le nombre d'itérations tend vers l'infini.
- Exemples d'applications :
  - Le *scheduling* des opérations dans l'industrie aérienne.
  - «L'intégration à très grande échelle» (*VLSI layout*) dans l'industrie des semi-conducteurs.

Recherche *Tabou* (*Tabu Search*)

- L'algorithme du recuit simulé (*simulated-annealing*) minimise le risque d'être piégé dans des minimums locaux.
  - Il n'élimine pas la possibilité d'osciller indéfiniment en revenant à un état antérieurement visité.
- Méthode recherche tabou (*Tabu Search*) :
  - Enregistrement des états visités;
  - Comme l'espace d'états est grand, on garde seulement les *n* derniers états visités.
  - L'ensemble *tabou* est l'ensemble contenant les n états.
  - Le paramètre *n* est choisi empiriquement et/ou selon la mémoire disponible.
  - Cela n'élimine pas les oscillations, mais tend à les réduire.

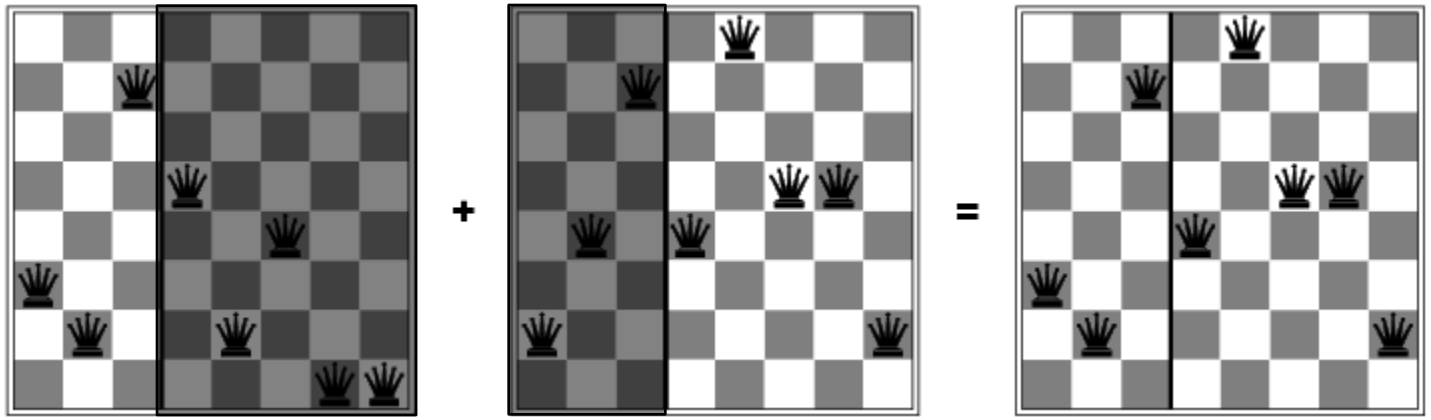
Recherche en faisceau (*beam searh*)

- On fait progresser un ensemble de *n* états plutôt qu'un seul état.
  1. On commence avec un ensemble de *n* 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 parmi ces états et on recommence.

Algorithmes génétiques

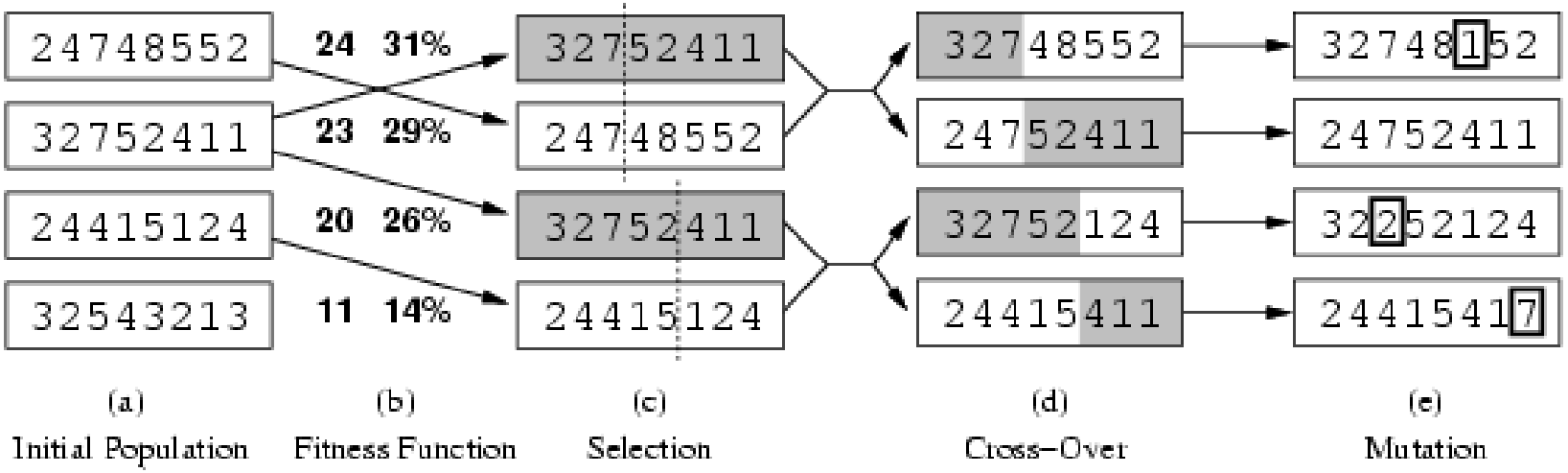
- Méthode inspirée de la **théorie de l'évolution** de Darwin.
- On commence avec une **population** de *n* états aléatoires (individus). **Première génération**.
- Un état est représenté à l'aide d'**une chaîne de symboles dans un alphabet fini** (souvent {0,1}).
- Fonction d'évaluation appelé **fonction de *fitness***. On cherche à maximiser la fonction de fitness.
- La prochaine génération est produite à l'aide de **sélections**, **croisements** et **mutations**.

Exemple de croisement avec *n*-reines.



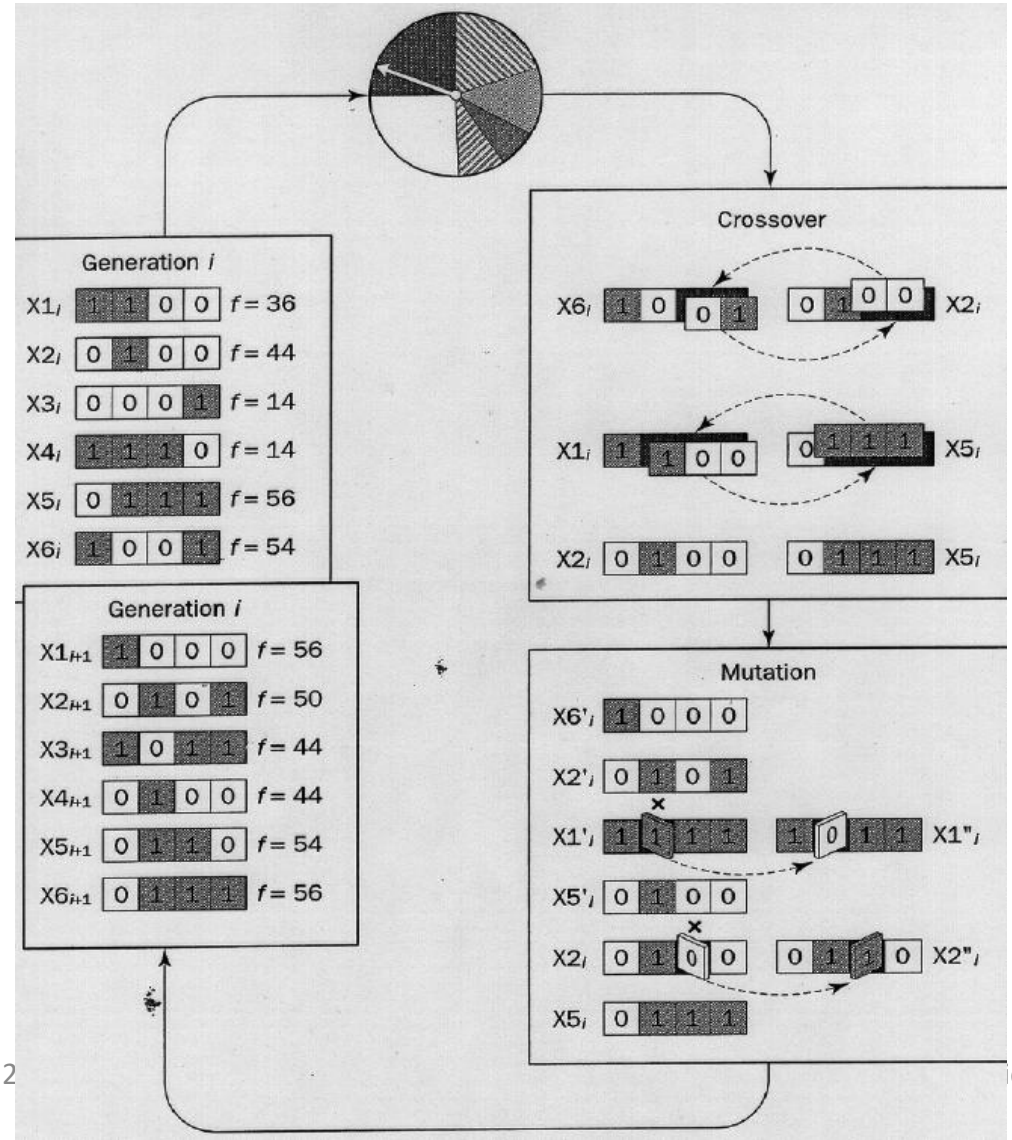
67247588 + 752**51448** = **67251448**

Exemple avec *n*-reines.



- *Fonction de fitness*: nombre de pairs de reines qui ne s'attaquent pas (min = 0, max = 8 × 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%

Résumé des étapes



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

Programmation génétique

- Approche similaire aux algorithmes génétiques.
  - Individus : chaînes d'instructions (programmes).

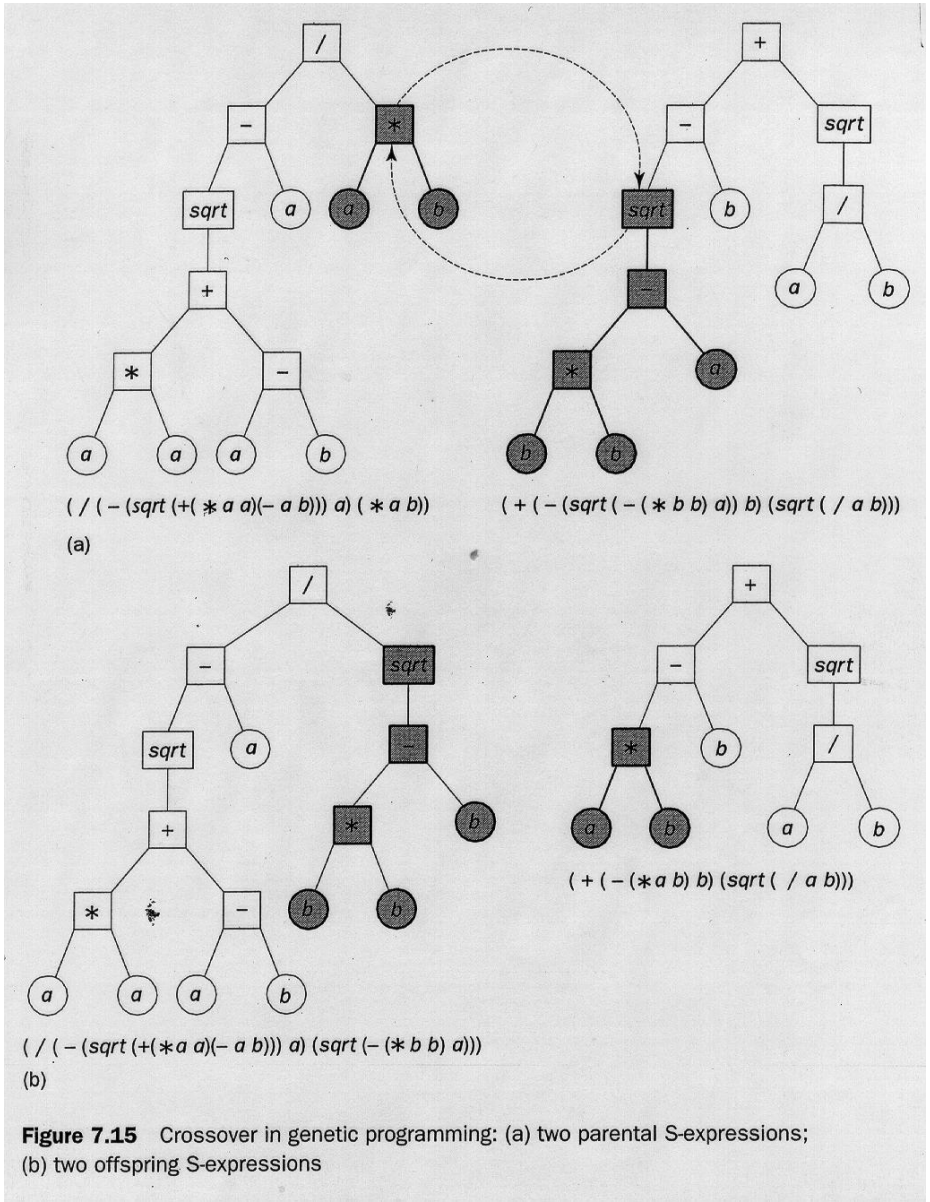


Figure 7.15 Crossover in genetic programming: (a) two parental S-expressions; (b) two offspring S-expressions

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

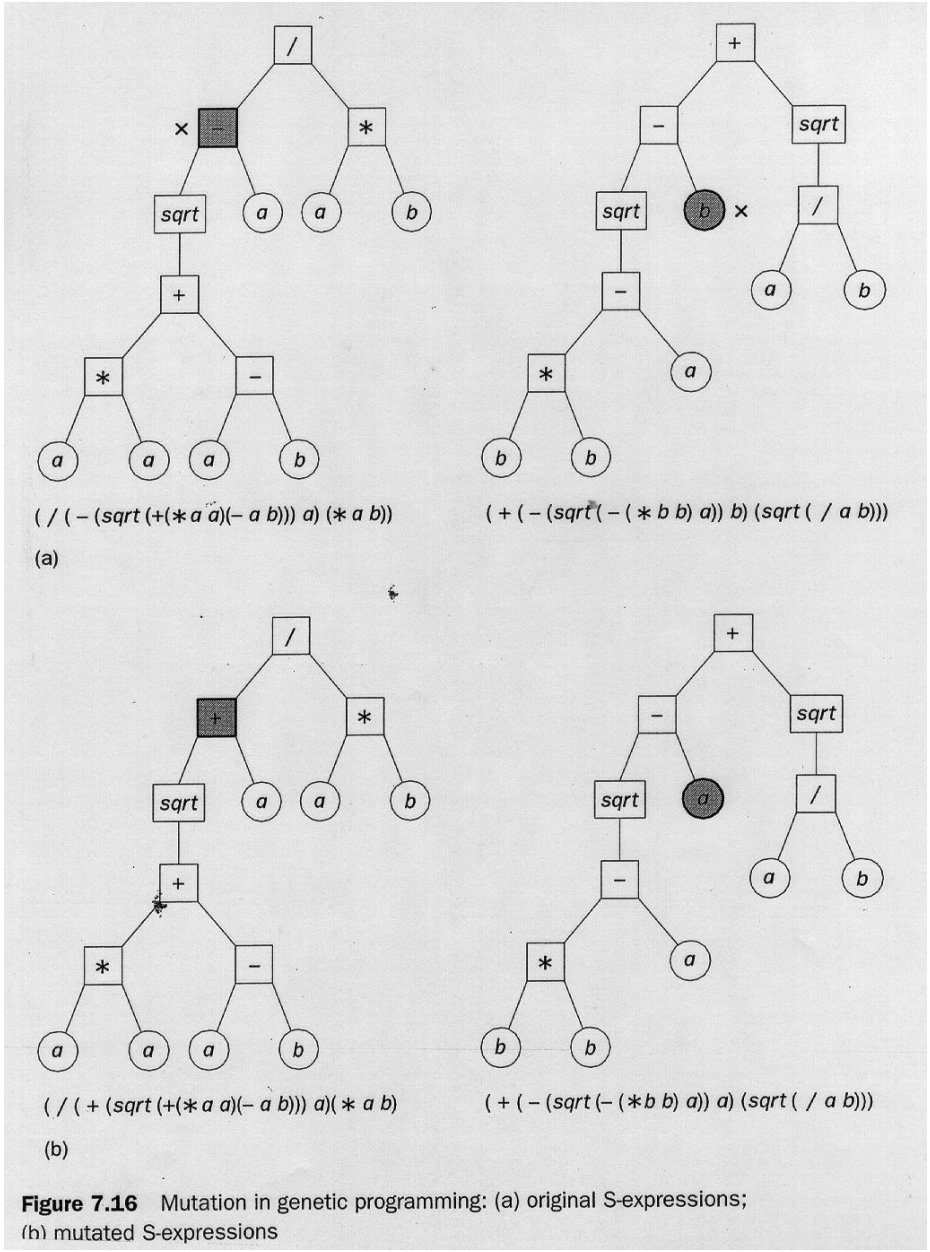


Figure 7.16 Mutation in genetic programming: (a) original S-expressions; (b) mutated S-expressions

[Michael Negnevitsky. Artificial Intelligence. Addison-Wesley, 2002. Page 248.]

Démos