# IFT 615 – Intelligence artificielle

## Recherche locale

Hugo Larochelle
Département d'informatique
Université de Sherbrooke
http://www.dmi.usherb.ca/~larocheh/cours/ift615.html

# Objectifs

- Comprendre:
  - la différence entre une recherche complète et une recherche locale
  - la méthode *hill-climbing*
  - la méthode *simulated-annealing*
  - la méthode *beam-search*
  - les algorithmes génétiques et de la programmation génétique

# Motivations pour une recherche locale

- Rappel de quelques faits saillants de A*:
  - un état final (le but) à atteindre est donné comme entrée
  - la solution est un chemin et non juste l'état final
  - idéalement on veut un chemin optimal
  - 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:
  - il y une **fonction objective** à optimiser (possiblement avec un état final)
  - la solution recherchée est juste l'état optimal (ou proche) et non le chemin qui y mène
  - l'espace d'états est trop grand pour enregistrer les états visités
- 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'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

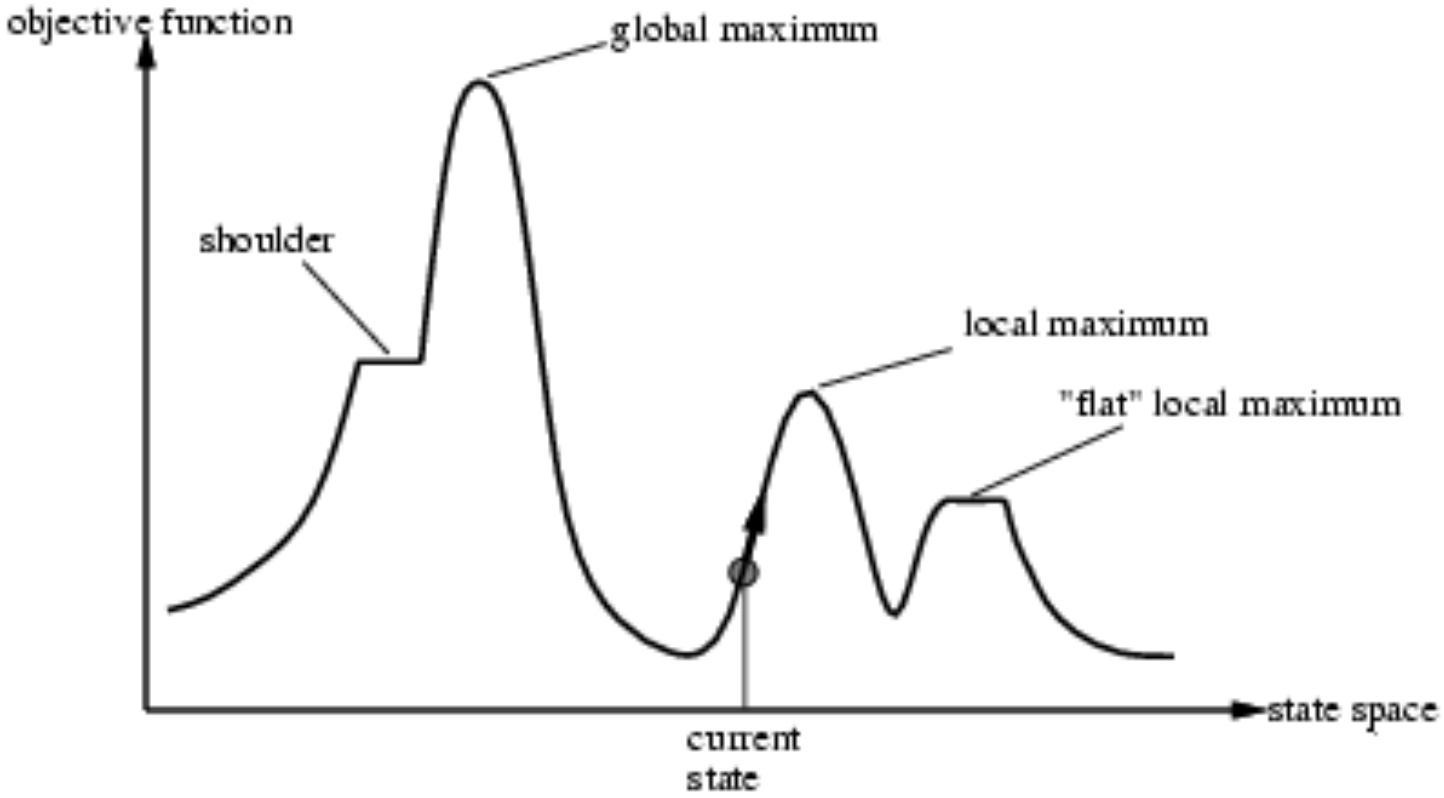
# Méthode Hill-Climbing

- Entrées:
  - état initial.
  - fonction objective à optimiser:
    - » notée VALUE dans l'algorithme
    - » parfois notée *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

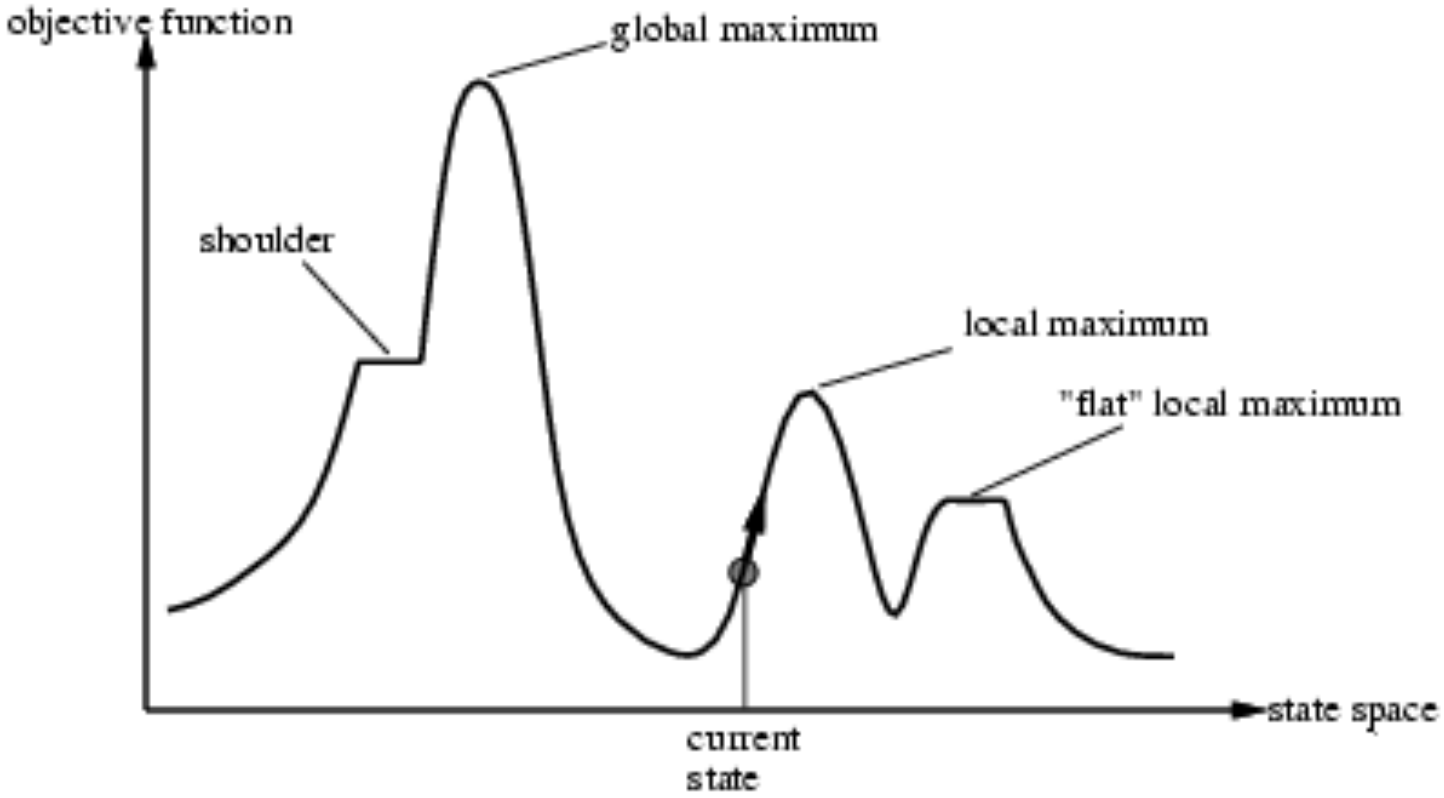
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    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
```

# Illustration de l'algorithme Hill-Climbing




Imaginez ce que vous feriez pour arriver au (trouver le) sommet d'une colline donnée, en plein brouillard et soufrant d'amnésie.

# Illustration de l'algorithme Hill-Climbing



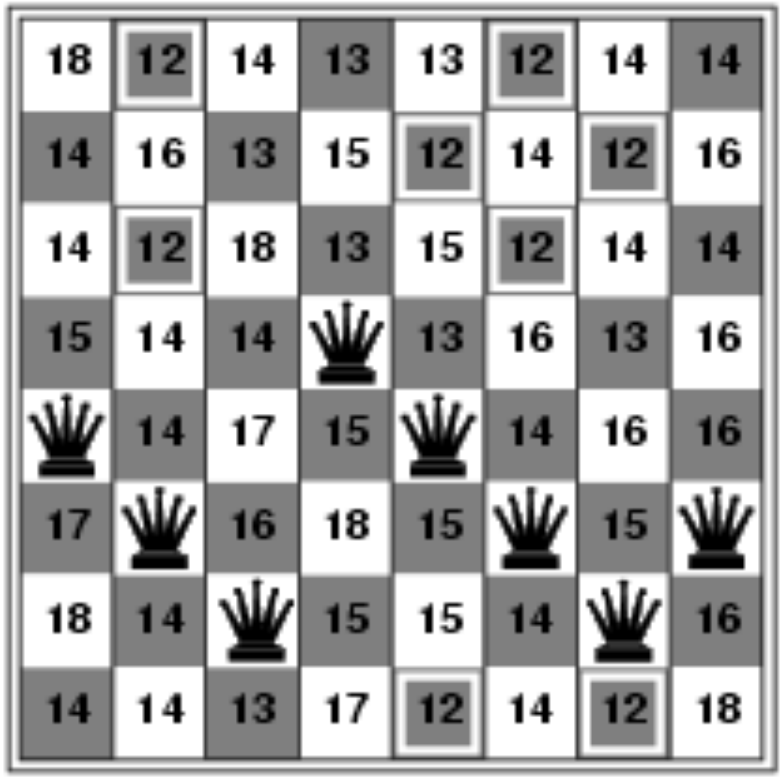

L'algorithme *hill-climbing* risque d'être piégé dans des optimums locaux: s'il atteint un nœud dont ses voisins immédiats sont moins bons, il arrête!

# Exemple: N-Queen

- Problème: Placer *N* reines sur un échiquier de taille *N*× *N* de sorte que deux reines ne s'attaquent mutuellement:
  - c-à-d., jamais deux reines sur la même diagonale, ligne ou colonne

# Hill-Climbing avec 8 reines

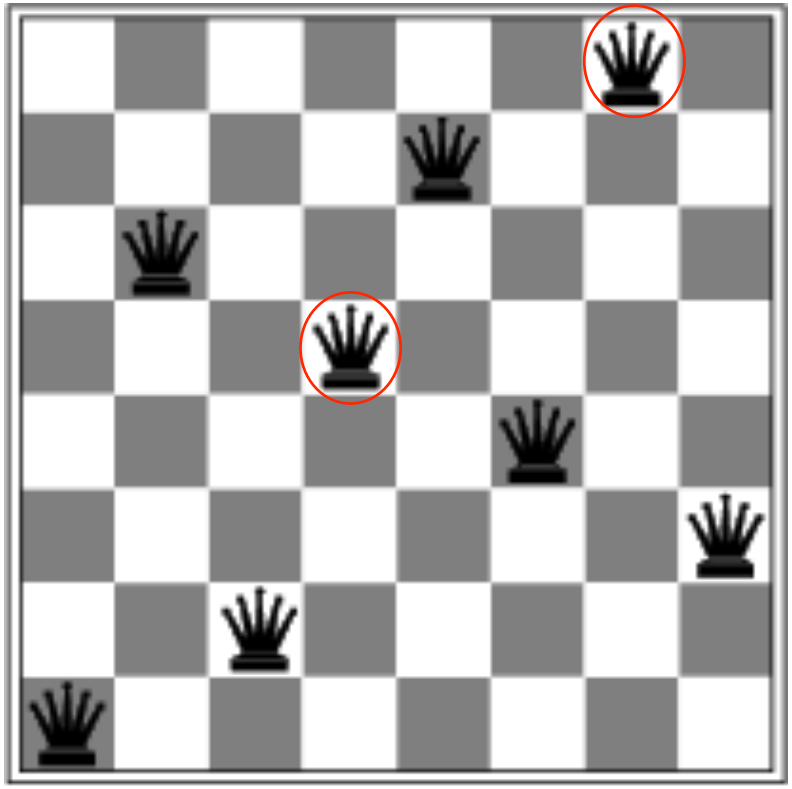
- h (VALUE): nombre de paires de reines qui s'attaquent mutuellement directement ou indirectement
- On veut le minimiser




- h pour l'état affiché: 17
- Encadré: les meilleurs successeurs, si on bouge une reine dans sa colonne

# 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 minimums locaux
- Approche:
  - au lieu de regarder le **meilleur** voisin immédiat du nœud courant, avec une certaine probabilité regarder 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)
- La méthode est 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 Simulated Annealing

**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* $\leftarrow$ MAKE-NODE(INITIAL-STATE[*problem*])
  **for** $t \leftarrow$ 1 **to** $\infty$ **do**
    $T \leftarrow$ *schedule*[$t$]
    **if** $T = 0$ **then return** *current*
    *next* $\leftarrow$ a randomly selected successor of *current*
    $\Delta E \leftarrow$ VALUE[*next*] – VALUE[*current*]
    **if** $\Delta E > 0$ **then** *current* $\leftarrow$ *next*
    **else** *current* $\leftarrow$ *next* only with probability $e^{\Delta E/T}$

# Tabu-Search

- L'algorithme *simulated-annealing* minimise le risque d'être piégé dans des minima locaux
- Par contre, il n'élimine pas la possibilité d'osciller indéfiniment en revenant à un état antérieurement visité
- On pourrait enregistrer les états visités (on revient à A* et approches similaires!) mais c'est impraticable si l'espace d'états est trop grand
- L'algorithme *tabu-search* enregistre seulement les *k* derniers états visités
  - l'ensemble *tabu* est l'ensemble contenant les *k* états
- Le paramètre *k* est choisi empiriquement
- Cela n'élimine pas les oscillations, mais les réduit

# Local beam-search

- On fait progresser un ensemble de *k* états plutôt qu'un seul état
  1. on commence avec un ensemble de *k* états choisis aléatoirement
  2. à chaque itération, tous les successeurs des *k* états sont générés
  3. si un d'eux satisfait le but, on arrête
  4. sinon on choisit les *k* meilleurs parmi ces états et on recommence

- À ne pas confondre avec *tabu-search*

# Algorithmes génétiques

- Idée très similaire à *local beam-search*
- Algorithme génétique
  - on commence aussi avec un ensemble *k* 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 **fonction de *fitness*** (fonction d'adaptabilité, de survie)
  - la prochaine génération est produite par **sélection, croisement et mutation**

# Algorithmes génétiques

- On peut aussi voir les algorithmes génétiques comme un modèle de calcul inspiré 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

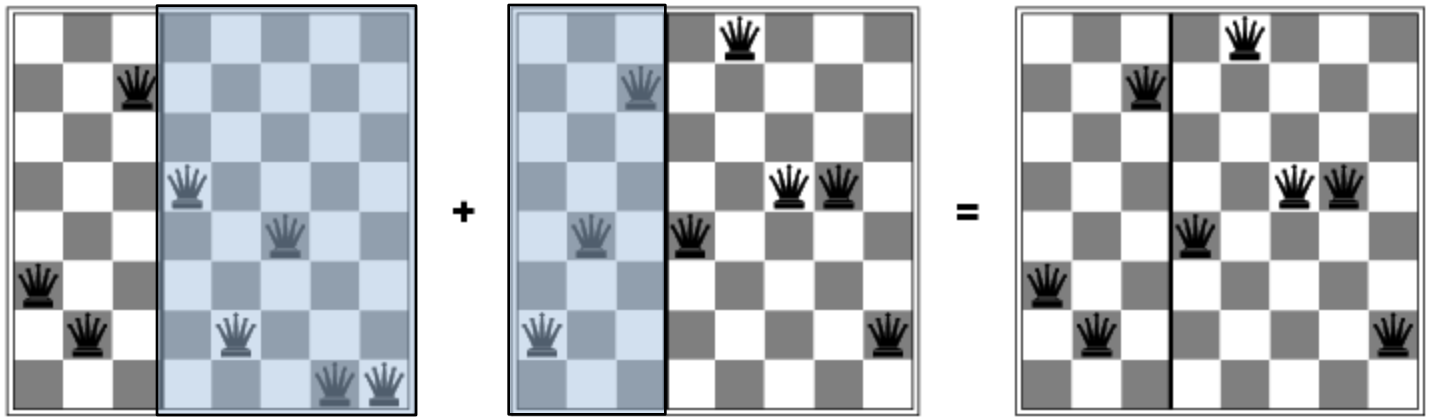
# Algorithmes génétiques

- 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 enfants sont conservés en fonction de leur **adaptabilité** (*fitness*) déterminée par une fonction d'adaptabilité donnée *f(x)*

# Algorithmes génétiques

1. Générer aléatoirement une population de *k* chromosomes.
2. Calculer la valeur d'adaptabilité (*fitness*) de chaque chromosome *x*
3. Créer une nouvelle population en:
   - sélectionnant 2 parents chromosomes, où **chaque parent est sélectionné avec une probabilité proportionnelle à son adaptabilité**
   - mutant l'enfant obtenu avec une certaine probabilité
   - plaçant l'enfant dans la population
4. Répéter l'étape 3 jusqu'à avoir une population de taille N
5. Si la population satisfait le critère d'arrêt, arrêter
6. Sinon, recommencer à l'étape 2

# Croisement: exemple avec 8 reines



67247588 + 75251448 = 67251448

# Exemple avec 8 reines


- Fonction de *fitness*: nombre de paires 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%

# Autre Exemple

- 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 |

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

# Autre Exemple (suite)

- Fixons la taille de la population à 6
- 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 |

# 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.]

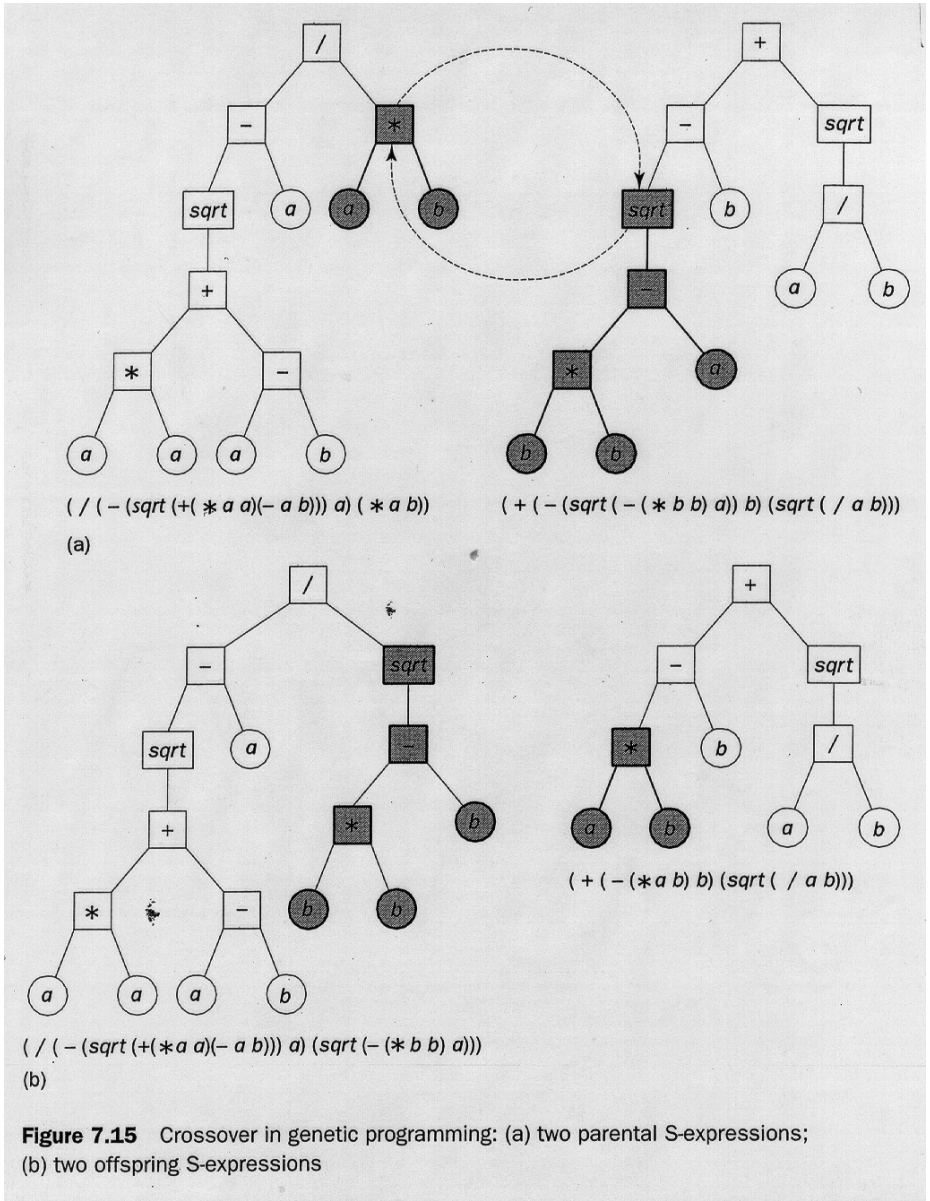



**Figure 7.15** Crossover in genetic programming: (a) two parental S-expressions; (b) two offspring S-expressions

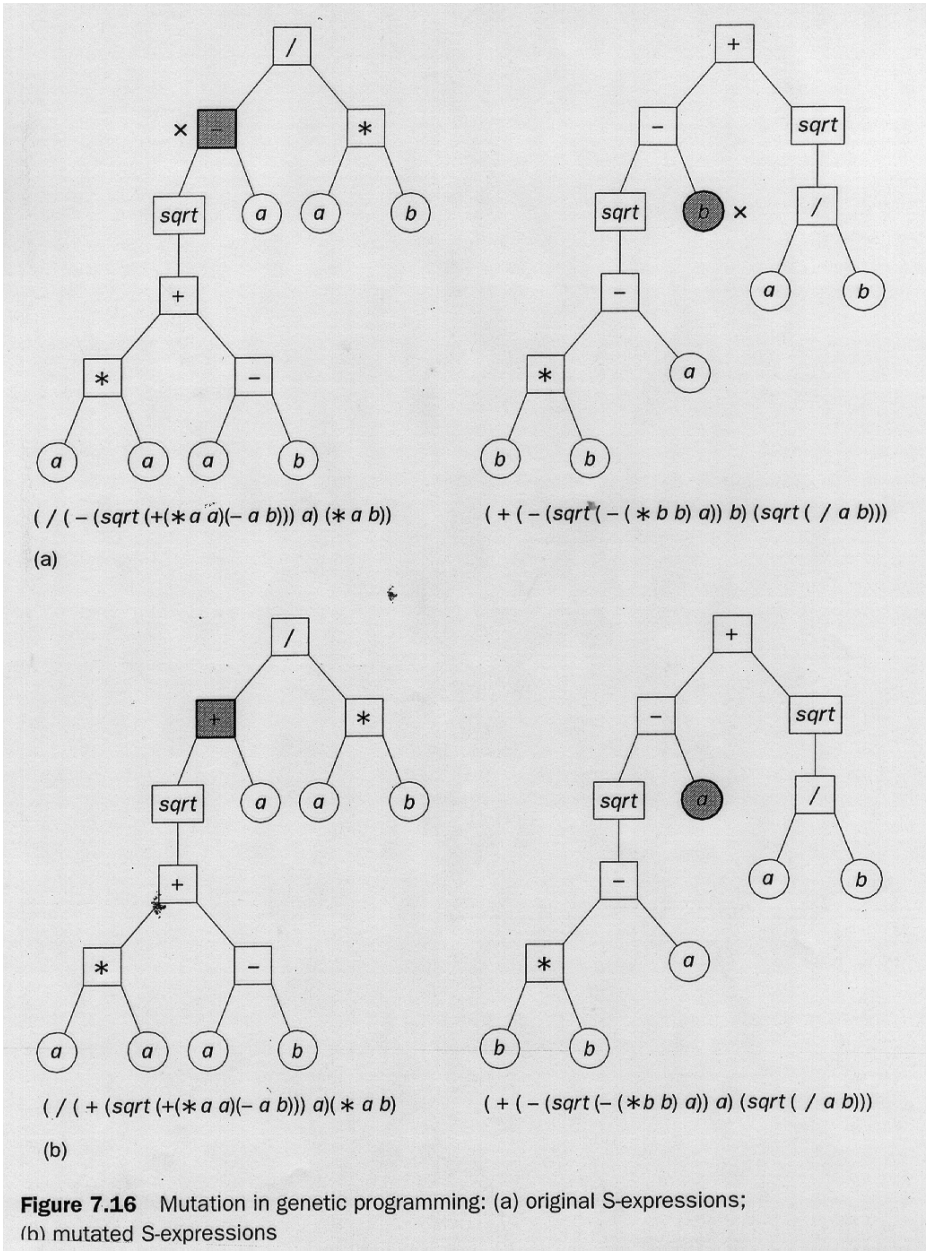



**Figure 7.16** Mutation in genetic programming: (a) original S-expressions; (b) mutated S-expressions