Local Search & Optimization

CE417: Introduction to Artificial Intelligence
Sharif University of Technology
Spring 2016

Soleymani

"Artificial Intelligence: A Modern Approach", 3rd Edition, Chapter 4

Outline

- Local search & optimization algorithms
  - Hill-climbing search
  - Simulated annealing search
  - Local beam search
  - Genetic algorithms
  - Searching in continuous spaces

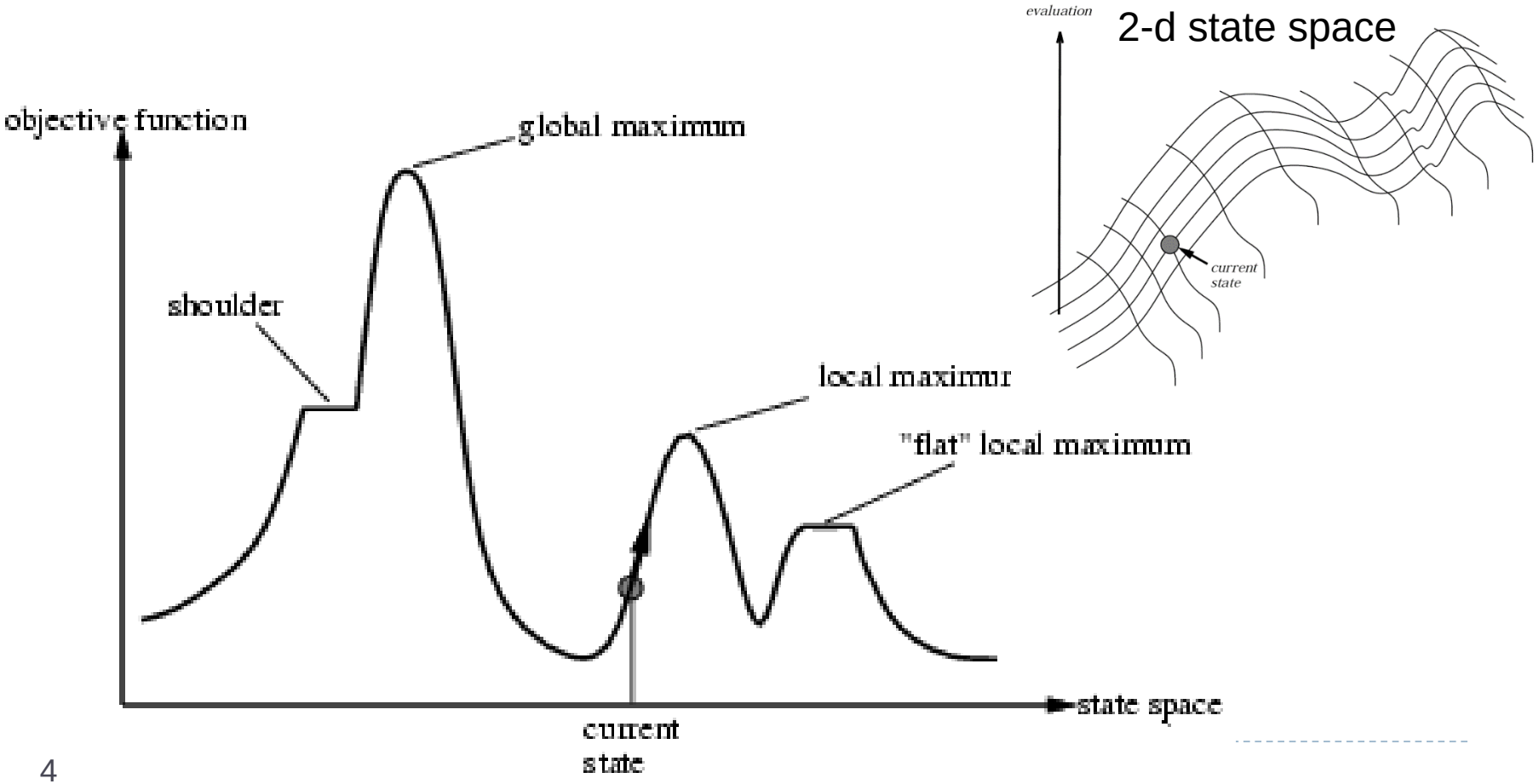
Sample problems for local & systematic search

- Path to goal is important
  - Theorem proving
  - Route finding
  - 8-Puzzle
  - Chess

- Goal state itself is important
  - 8 Queens
  - TSP
  - VLSI Layout
  - Job-Shop Scheduling
  - Automatic program generation

State-space landscape

- Local search algorithms explore the landscape
- Solution: A state with the optimal value of the objective function



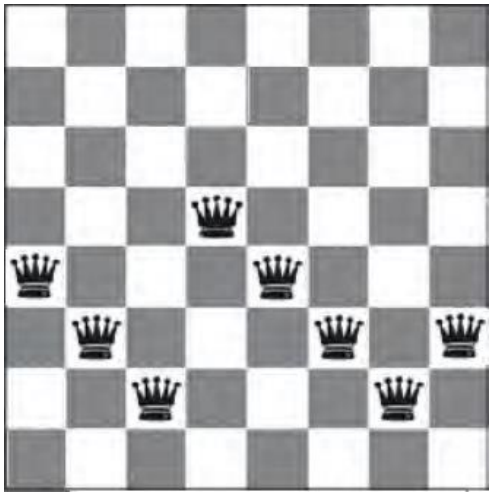
Example: *n*-queens

- Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal
- What is state-space?
- What is objective function?

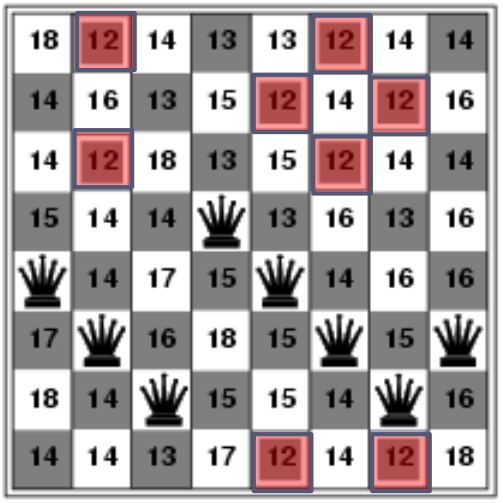
Local search: 8-queens problem

- States: 8 queens on the board, one per column $(8^8 \approx 17\ million)$
- Successors(s): all states resulted from s by moving a single queen to another square of the same column $(8 \times 7 = 56)$
- Cost function $h(s)$: number of queen pairs that are attacking each other, directly or indirectly
- Global minimum: $h(s) = 0$

$h(s) = 17$



successors objective values



Red: best successors

Hill-climbing search

- Node only contains the state and the value of objective function in that state (not path)
- Search strategy: steepest ascent among immediate neighbors until reaching a peak

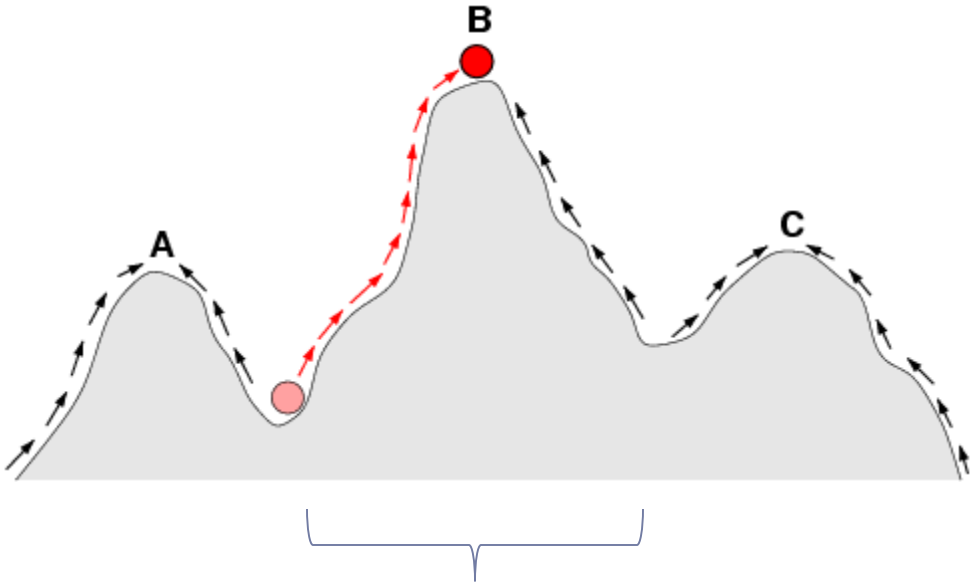
```
function HILL-CLIMBING(problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    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
```

Current node is replaced by the best successor (if it is better than current node)

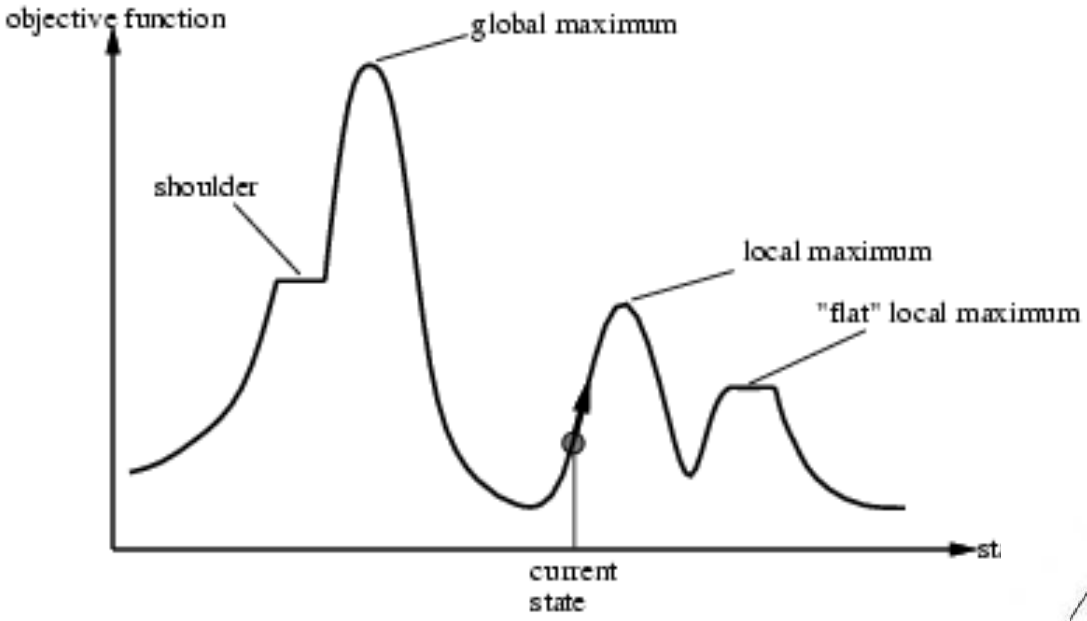
Hill-climbing search is greedy

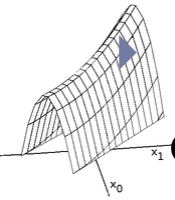
- Greedy local search: considering only one step ahead and select the best successor state (steepest ascent)
  - Rapid progress toward a solution
    - Usually quite easy to improve a bad solution



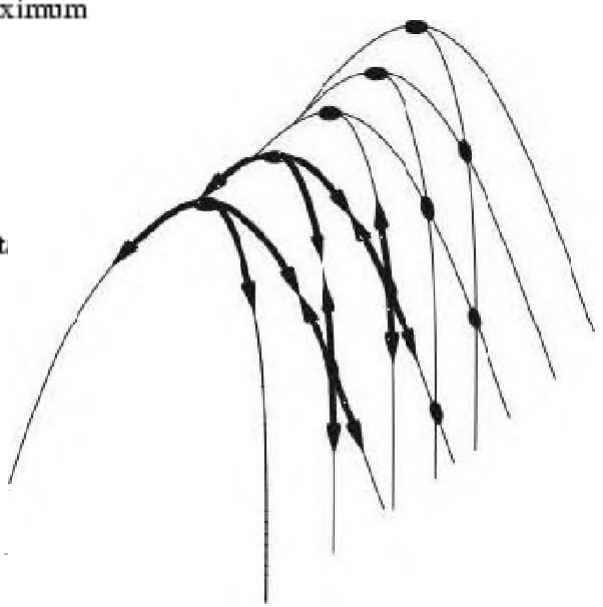
Hill-climbing search problems

- Local maxima: a peak that is not global max
- Plateau: a flat area (flat local max, shoulder)





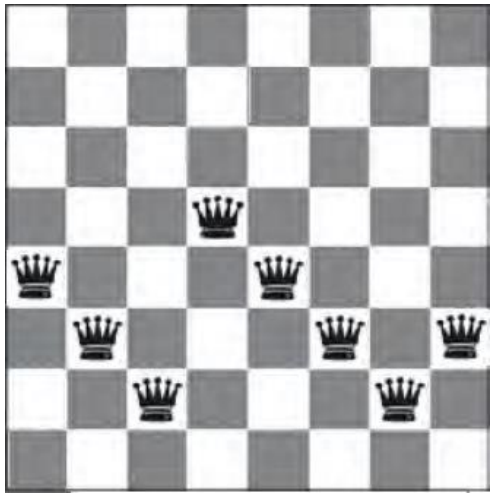
Ridges: a sequence of local max that is very difficult for greedy algorithm to navigate



Hill-climbing search problem: 8-queens

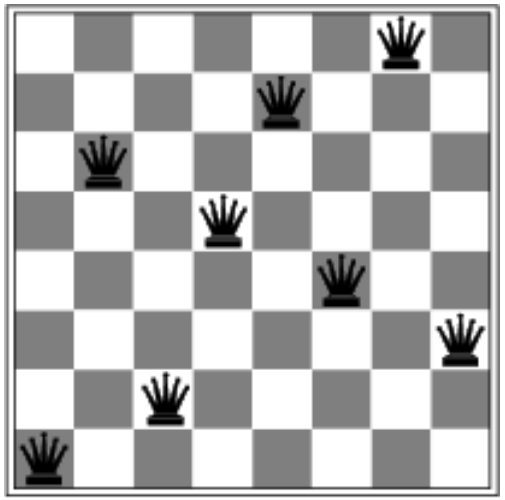
- From random initial state, 86% of the time getting stuck
  - on average, 4 steps for succeeding and 3 steps for getting stuck

$h(s) = 17$



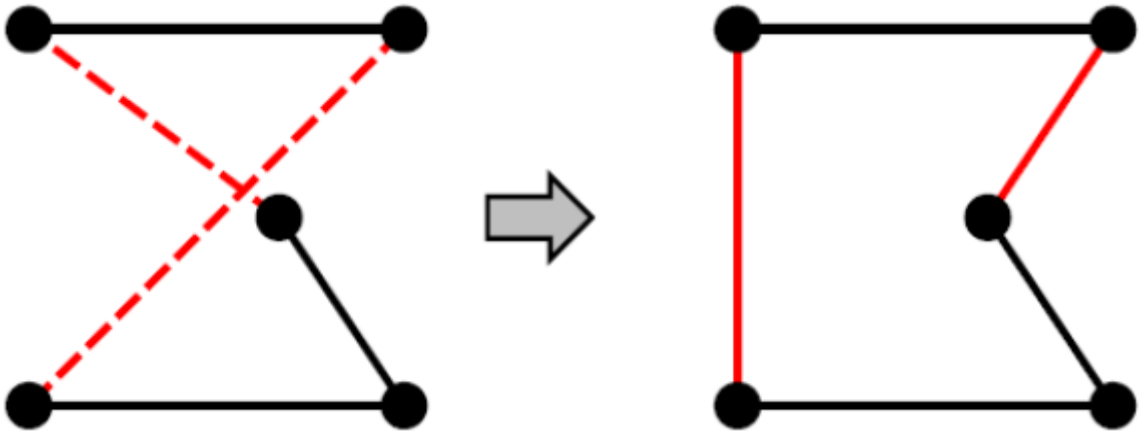
Five steps

$h(s) = 1$



Hill-climbing search problem: TSP

- Start with any complete tour, perform pairwise exchanges
  - Variants of this approach get within 1% of optimal very quickly with thousands of cities

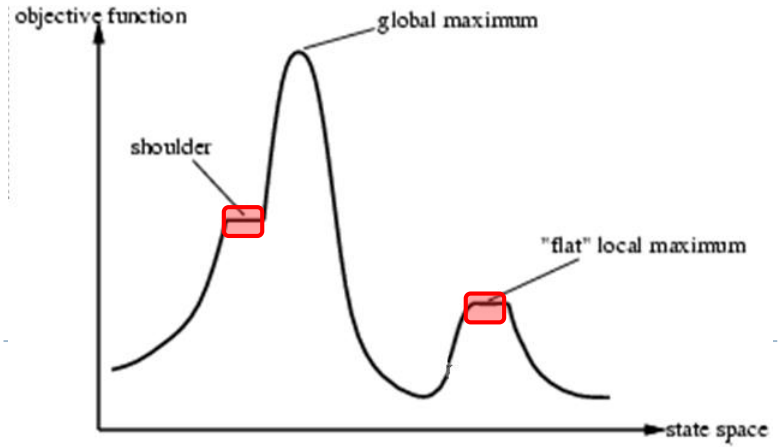


Variants of hill-climbing

- Trying to solve problem of hill-climbing search
  - Sideways moves
  - Stochastic hill climbing
    - First-choice hill climbing
  - Random-restart hill climbing

Sideways move

- Sideways move: plateau may be a shoulder so keep going sideways moves when there is no uphill move
  - Problem: infinite loop where flat local max
    - Solution: upper bound on the number of consecutive sideways moves
- Result on 8-queens:
  - Limit = 100 for consecutive sideways moves
    - 94% success instead of 14% success
      - on average, 21 steps when succeeding and 64 steps when failing



Stochastic hill climbing

- Randomly chooses among the available uphill moves according to the steepness of these moves
  - $P(S')$ is an increasing function of $h(s') - h(s)$

- First-choice hill climbing: generating successors randomly until one better than the current state is found
  - Good when number of successors is high

Random-restart hill climbing

- All previous versions are incomplete
  - Getting stuck on local max
- **while** state ≠ goal **do**

  run hill-climbing search from a random initial state
- p: probability of success in each hill-climbing search
  - Expected no of restarts = $1/p$
  - Expected no of steps = $(1/p - 1) \times n_f + n_s$
    - n_f: average number of steps in a failure
    - n_s: average number of steps for the success
- Result on 8-queens:
  - $p \approx 0.14 \Rightarrow 1/p \approx 7$ iterations
    - $(7 - 1) \times 3 + 4 \approx 22$ steps
    - For 3×10^6 queens needs less than 1 minute
  - Using also sideways moves: $p \approx 0.94 \Rightarrow 1/p \approx 1.06$ iterations
    - $(1.06 - 1) \times 64 + 21 \approx 26$ steps

Effect of land-scape shape on hill climbing

- Shape of state-space land-scape is important:
  - Few local max and platea: random-restart is quick
  - Real problems land-scape is usually unknown a priori
  - NP-Hard problems typically have an exponential number of local maxima
    - Reasonable solution can be obtained after a small no of restarts

Simulated Annealing (SA) Search

- **Hill climbing**: move to a better state
  - Efficient, but incomplete (can stuck in local maxima)
- **Random walk**: move to a random successor
  - Asymptotically complete, but extremely inefficient

- Idea: Escape local maxima by allowing some "bad" moves but gradually decrease their frequency.
  - More exploration at start and gradually hill-climbing become more frequently selected strategy

SA relation to annealing in metallurgy

- In SA method, each state *s* of the search space is analogous to a **state of some physical system**
- *E*(*s*) to be minimized is analogous to the **internal energy of the system**
- The goal is to bring the system, from an arbitrary *initial state*, to an equilibrium state with the minimum possible energy.

```
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)
```

- Pick a random successor of the current state
- If it is better than the current state go to it
- Otherwise, accept the transition with a probability

$T(t) = schedule[t]$ is a decreasing series

E(s): objective function

Probability of state transition

A successor of s

$$P(s, s', t) = \alpha \times \begin{cases} 1 & if\ E(s') > E(s) \\ e^{(E(s')-E(s))/T(t)} & o.w. \end{cases}$$

- Probability of "un-optimizing" ($\Delta E = E(s') - E(s) < 0$) random movements depends on badness of move and temperature
  - Badness of movement: worse movements get less probability
  - Temperature
    - High temperature at start: higher probability for bad random moves
    - Gradually reducing temperature: random bad movements become more unlikely and thus hill-climbing moves increase

SA as a global optimization method

- Theoretical guarantee: If T decreases slowly enough, simulated annealing search will converge to a global optimum (with probability approaching 1)

- Practical? Time required to ensure a significant probability of success will usually exceed the time of a complete search

Local beam search

- Keep track of k states
  - Instead of just one in hill-climbing and simulated annealing

Start with k randomly generated states

Loop:

All the successors of all k states are generated

If any one is a goal state **then** stop

else select the k best successors from the complete list of successors and repeat.

Local beam search

- Is it different from running high-climbing with k random restarts in parallel instead of in sequence?
  - Passing information among parallel search threads

- Problem: Concentration in a small region after some iterations
  - Solution: Stochastic beam search
    - Choose k successors at random with probability that is an increasing function of their objective value

Genetic Algorithms

- A variant of stochastic beam search
  - Successors can be generated by combining two parent states rather than modifying a single state

- Inspired by
  - Natural Selection: "Variations occur in reproduction and will be preserved in successive generations approximately in proportion to their effect on reproductive fitness"
  - Reproduction in the nature usually combines two parents to generate off springs

Evolution theory

- There is competition among living things
  - More are born than survive and reproduce
- Selection determines which individuals enter the adult breeding population
  - This selection is done by the environment
  - Those which are best suited or fitted reproduce
  - They pass these well suited characteristics on to their young
- Reproduction occurs with variation
  - This variation is heritable
- "Survival of the fittest" means those who have the most offspring that reproduce

Genetic Algorithms: inspiration by natural selection

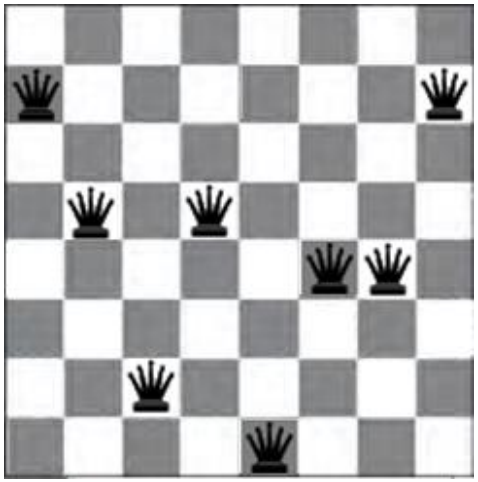
- State: organism
- Objective value: fitness (populate the next generation according to its value)
- Successors: offspring (they may be located long away from parents)

Genetic Algorithm (GA)

- A state (solution) is represented as a string over a finite alphabet
  - Like a chromosome containing genes
- Start with *k* randomly generated states (population)
- Evaluation function to evaluate states (fitness function)
  - Higher values for better states
- Combining two parent states and getting offsprings (cross-over)
  - Cross-over point can be selected randomly
- Reproduced states can be slightly modified (mutation)
- The next generation of states is produced by selection (based on fitness function), crossover, and mutation

Chromosome & Fitness: 8-queens

- Describe the individual (or state) as a string

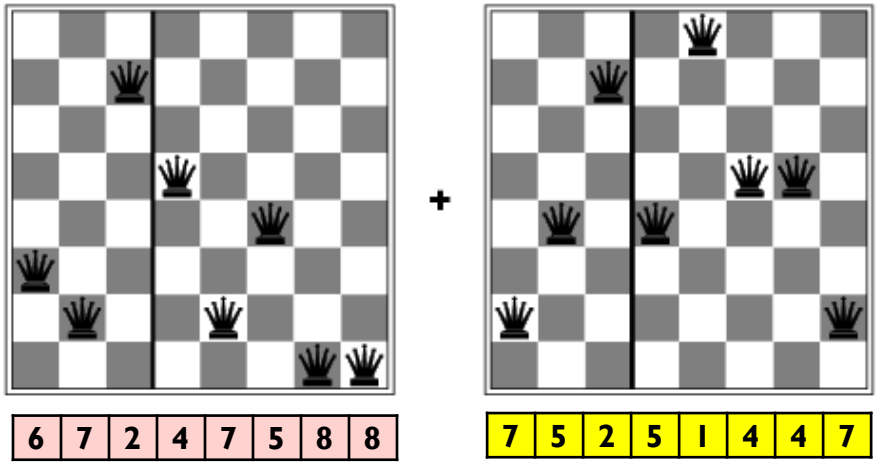


2	4	7	4	8	5	5	2

- Fitness function: number of non-attacking pairs of queens
  - 24 for above figure

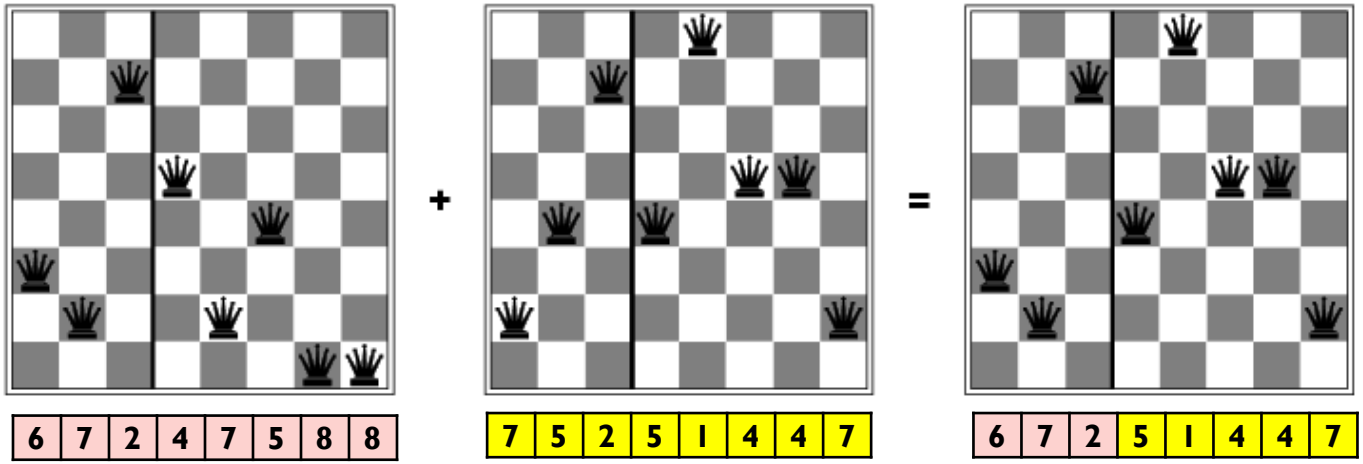
Genetic operators: 8-queens

- Cross-over: To select some part of the state from one parent and the rest from another.



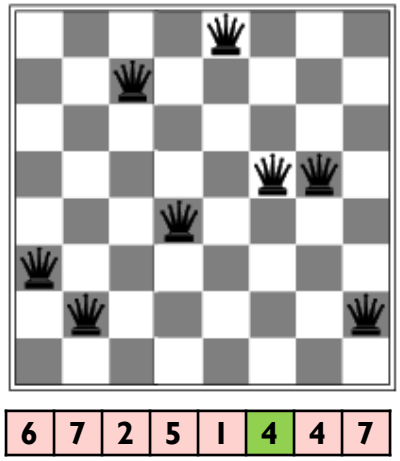
Genetic operators: 8-queens

- Cross-over: To select some part of the state from one parent and the rest from another.



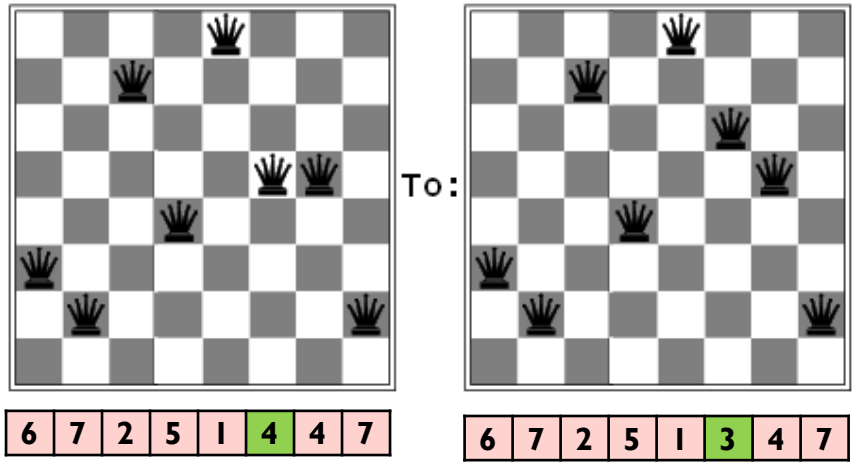
Genetic operators: 8-queens

- Mutation: To change a small part of one state with a small probability.



Genetic operators: 8-queens

- Mutation: To change a small part of one state with a small probability.



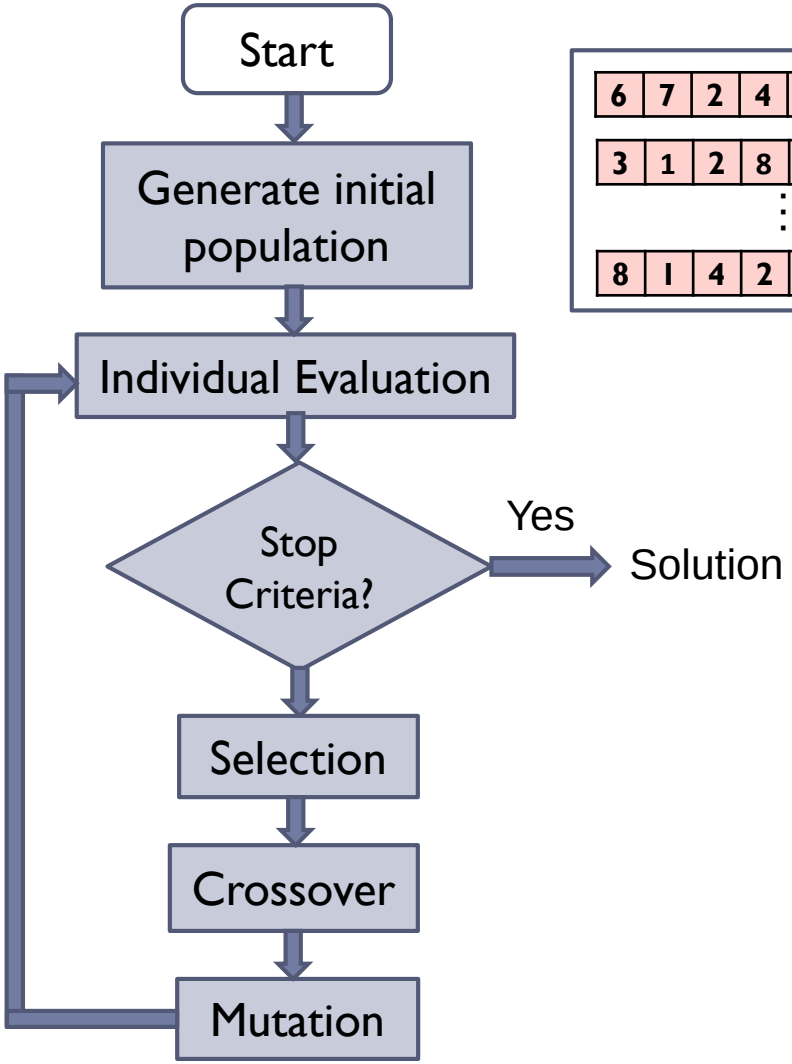
```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns an individual
  inputs: population, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
    new-population ← empty set
    loop for i from 1 to SIZE(population) do
      x ← RANDOM-SELECTION(population, FITNESS-FN)
      y ← RANDOM-SELECTION(population, FITNESS-FN)
      child ← REPRODUCE(x, y)
      if (small random probability) then child ← MUTATE(child)
      add child to new-population
    population ← new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN
```

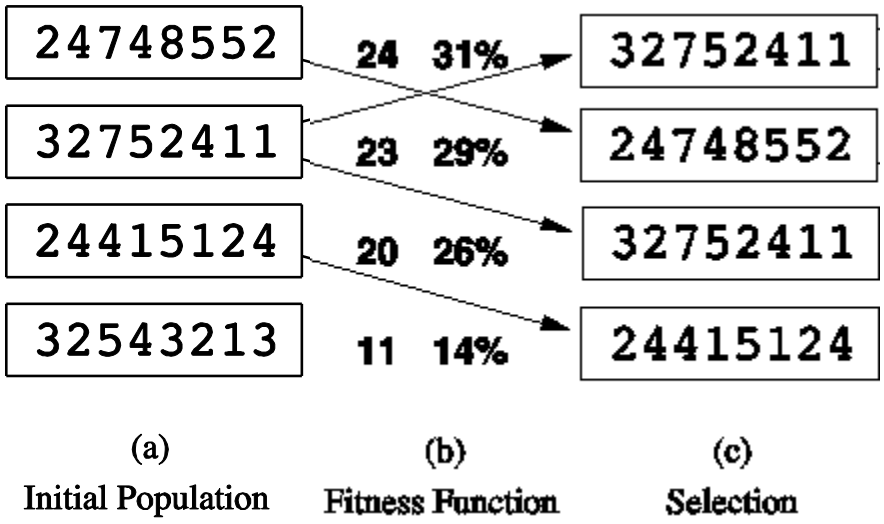
```
function REPRODUCE(x, y) returns an individual
  inputs: x, y, parent individuals

  n ← LENGTH(x)
  c ← random number from 1 to n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

A Genetic algorithm diagram

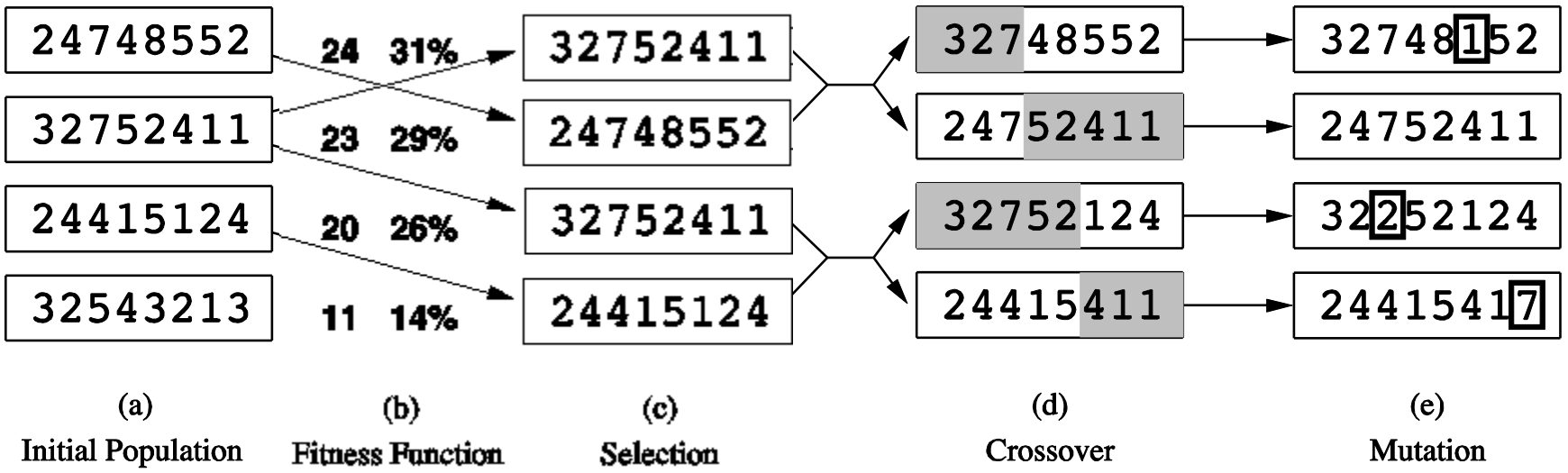


A variant of genetic algorithm: 8-queens



- Fitness function: number of non-attacking pairs of queens
  - min = 0, max = 8 × 7/2 = 28
  - Reproduction rate(i) = $fitness(i) / \sum_{k=1}^{n} fitness(k)$
    - e.g., 24/(24+23+20+11) = 31%

A variant of genetic algorithm: 8-queens



- Fitness function: number of non-attacking pairs of queens
  - min = 0, max = 8 × 7/2 = 28
  - Reproduction rate(*i*) = $fitness(i) / \sum_{k=1}^{n} fitness(k)$
    - e.g., 24/(24+23+20+11) = 31%

Genetic algorithm properties

- Why does a genetic algorithm usually take large steps in earlier generations and smaller steps later?
  - Initially, population individuals are diverse
    - Cross-over operation on different parent states can produce a state long a way from both parents
  - More similar individuals gradually appear in the population

- Cross-over as a distinction property of GA
  - Ability to combine large blocks of genes evolved independently
    - Representation has an important role in benefit of incorporating crossover operator in GA

Local search in continuous spaces

- Infinite number of successor states
  - E.g., select locations for 3 airports such that sum of squared distances from each city to its nearest airport is minimized
    - (x_1, y_1) , (x_2, y_2) , (x_3, y_3)
    - $F(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$

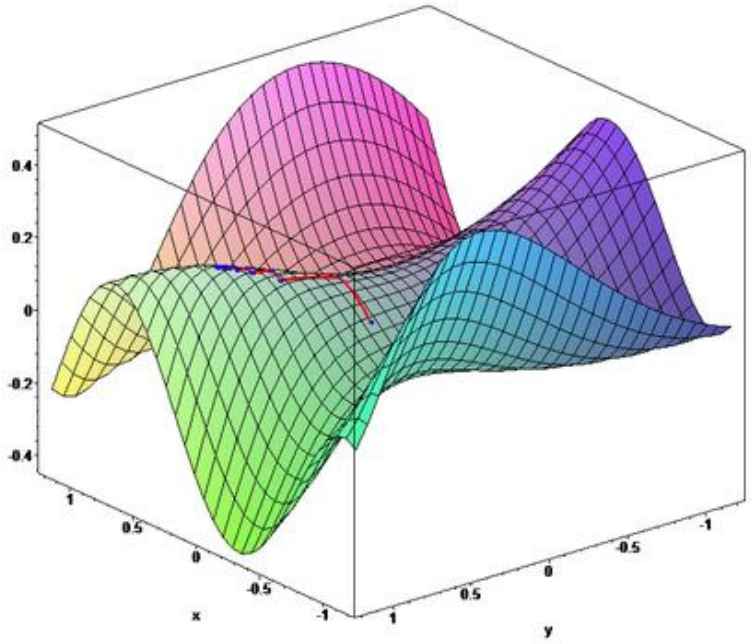
- Approach 1: Discretization
  - Just change variable by $\pm\delta$
    - E.g., 6×2 actions for airport example

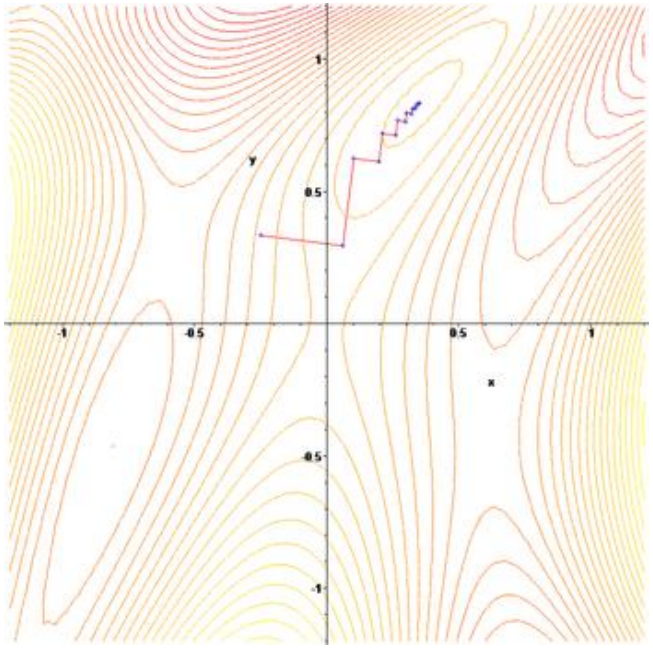
- Approach 2: Continuous optimization
  - $\nabla f = 0$ (only for simple cases)
  - Gradient ascent $\mathbf{x}^{t+1} \leftarrow \mathbf{x}^t + \alpha \nabla f(\mathbf{x}^t)$

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, + \frac{\partial f}{\partial x_d} \right)$$

Gradient ascent

$$\mathbf{x}^{t+1} \leftarrow \mathbf{x}^{t} + \alpha \nabla f(\mathbf{x}^{t})$$





- Local search problems also in continuous spaces
  - Random restart hill-climbing and simulated annealing can be useful
  - Higher dimensions raises the rate of getting lost

Gradient ascent (step size)

- Adjusting α in gradient descent
  - Line search
  - Newton-Raphson

$$\mathbf{x}^{t+1} \leftarrow \mathbf{x}^{t} - \mathbf{H}_{f}^{-1}(\mathbf{x}^{t})\nabla f\ (\mathbf{x}^{t})$$

$$H_{ij} = \partial^2 f \,/\, \partial x_i \partial x_j$$

Local search vs. systematic search

	Systematic search	Local search
Solution	Path from initial state to the goal	Solution state itself
Method	Systematically trying different paths from an initial state	Keeping a single or more "current" states and trying to improve them
State space	Usually incremental	Complete configuration
Memory	Usually very high	Usually very little (constant)
Time	Finding optimal solutions in small state spaces	Finding reasonable solutions in large or infinite (continuous) state spaces
Scope	Search	Search & optimization problems