Search Algorithms: Object-Oriented Implementation (Part A)

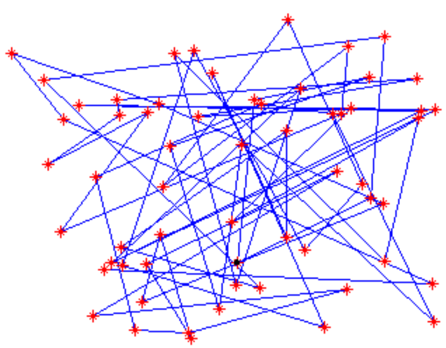
Contents

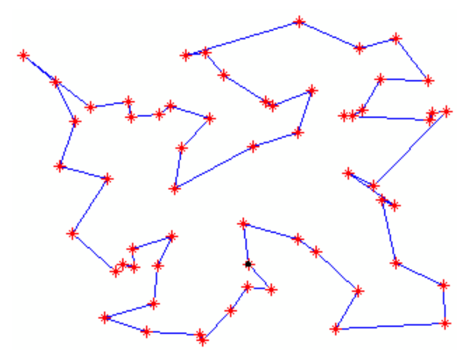
Conventional vs. AI Algorithms

- Intractable problems:
  - There are many optimization problems that require a lot of time to solve but no efficient algorithms have been identified
  - Exponential algorithms are useless except for very small problems

<u>Example</u>: Traveling Salesperson Problem:

The salesperson wants to minimize the total traveling cost required to visit all the cities in a territory, and return to the starting point



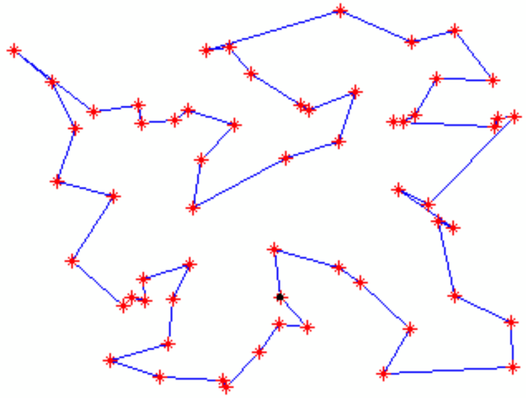


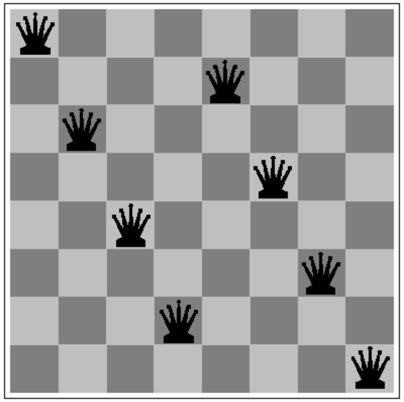
Conventional vs. AI Algorithms

- Combinatorial Explosion: there are $n!$ routes to investigate
  $\rightarrow$ Efficiency is the issue!
- Nearest neighbor heuristic: always goes to the nearest city
  - Given a starting city, there are $(n-1)+(n-2)+\cdots+1=n(n-1)/2$ cases to consider
  - Since there are n different ways to start, the total number of cases is $n^2(n-1)/2$
  - Can find a near optimal solution in a much shorter time
- Conventional algorithms (e.g., sorting algorithms) are often called exact algorithms because they always find a correct or an optimal solution
- AI algorithms use heuristics or randomized strategies to find a near optimal solution quickly

Local Search Algorithms

- Iterative improvement algorithms:
  – State space = set of “complete” configurations (e.g., complete tours in TSP)
  – Find optimal configuration according to the objective function
  – Find configuration satisfying constraints (e.g., n-queens Problem)
- Start with a complete configuration and make modifications to improve its quality


TSP


8-queens

Local Search Algorithms

Example: TSP

– Current configuration:

B	A	C	E	D

– Candidate neighborhood configurations:

B	E	C	A	D
B	A	E	C	D
E	C	A	B	D
B	A	C	D	E
C	A	B	E	D

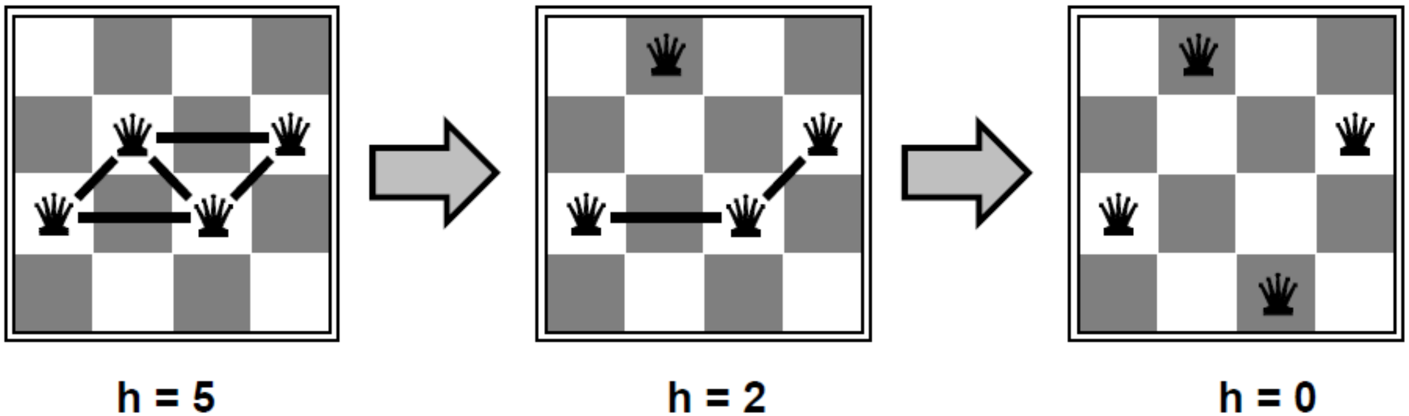
Mutation by inversion:
A random subsequence is inverted

Variants of this approach get within 1% of optimal very quickly with thousands of cities

Local Search Algorithms

<u>Example</u>: n-queens

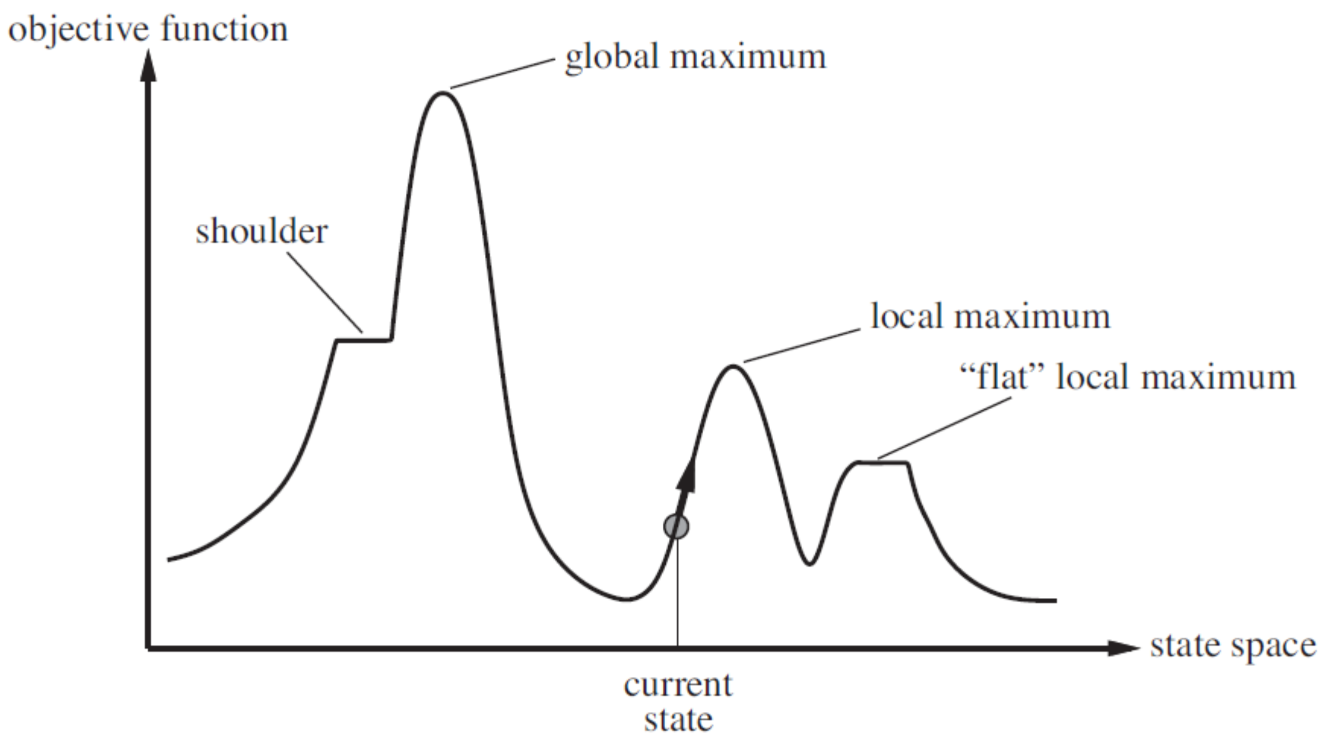
- Move a queen to reduce number of conflicts



- Almost always solves n-queens problems almost instantaneously for very large n, e.g., $n = 1$ million

Local Search Algorithms

- State space landscape
  - Location: state
  - Elevation: heuristic cost function or objective function



Hill-Climbing Search

- "Like climbing Everest in thick fog with amnesia"
  - Continually moves in the direction of increasing value
  - Also called gradient ascent/descent search

[Steepest ascent version]

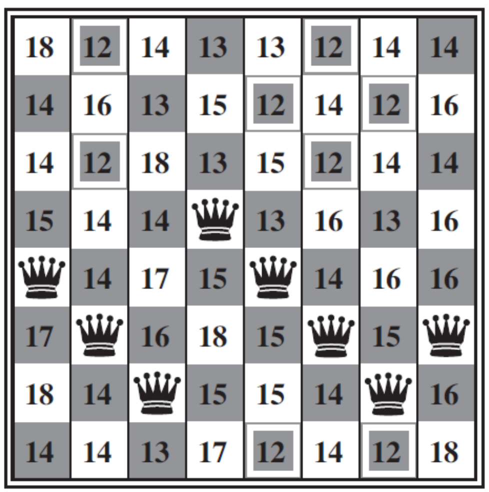
function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum

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

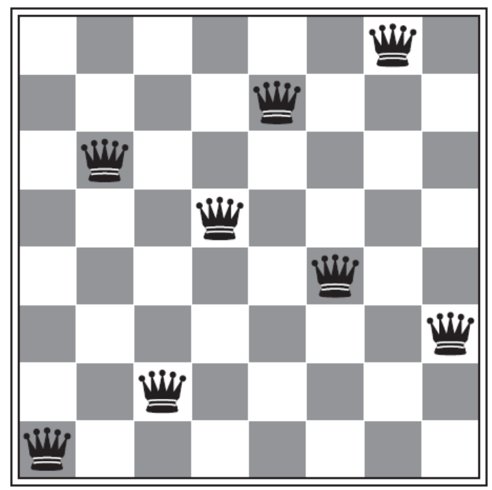
Hill-Climbing Search

Example: 8-queens problem

- Each state has 8 queens on the board, one per column
- Successor function generates 56 states by moving a single queen to another square in the same column
- h is the # of pairs that are attacking each other



A state with $h = 17$



A local minimum

Hill-Climbing Search

Example: Suppose we want to site three airports in Romania:

- 6-D state space defined by (x_1, y_1), (x_2, y_2), (x_3, y_3)
- Objective function:

  $f(x_1, y_1, x_2, y_2, x_3, y_3) =$ sum of squared distances from each city to nearest airport

- We can discretize the neighborhood of each state and apply any local search algorithm
  - Move in each direction by a fixed amount $\pm\delta$ (12 successors)
- We can directly (without discretization) search in continuous spaces
  - Successors are chosen randomly by generating 6-dimensional random vectors of length δ

Hill-Climbing Search

- Drawbacks: often gets stuck to local maxima due to greediness
- Possible solutions:
  - Stochastic hill climbing:
    - Chooses at random from among the uphill moves with probability proportional to steepness
  - First-choice (simple) hill climbing:
    - Generates successors randomly until one is found that is better than the current state
  - Random-restart hill climbing:
    - Conducts a series of hill-climbing searches from randomly generated initial states
    - Very effective for n-queens
      (Can find solutions for 3 million queens in under a minute)

Hill-Climbing Search

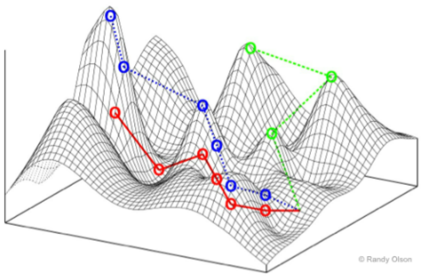
- Complexity:
  - The success of hill climbing depends on the shape of the state-space landscape
  - NP-hard problems typically have an exponential number of local maxima to get stuck on
  - A reasonably good local maximum can often be found after a small number of restarts

Continuous State Spaces

- Gradient methods attempt to use the gradient of the landscape to maximize/minimize f by

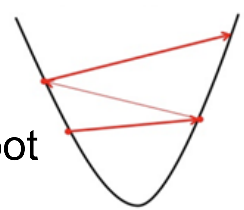
$$x \leftarrow x +/- \alpha \nabla f(x) \quad (\alpha: \text{update rate})$$

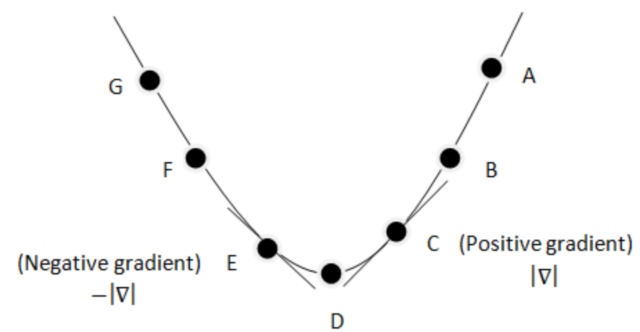
where $\nabla f(x)$ is the gradient vector (containing all of the partial derivatives) of f that gives the magnitude and direction of the steepest slope



  - Too small α: too many steps are needed
  - Too large α: the search could overshoot the target
  - Points where $\nabla f(x) = 0$ are known as critical points

Overshoot





Continuous State Spaces

Example: Gradient descent

– If $f(w) = w^2 + 1$, then $f'(w) = 2w$

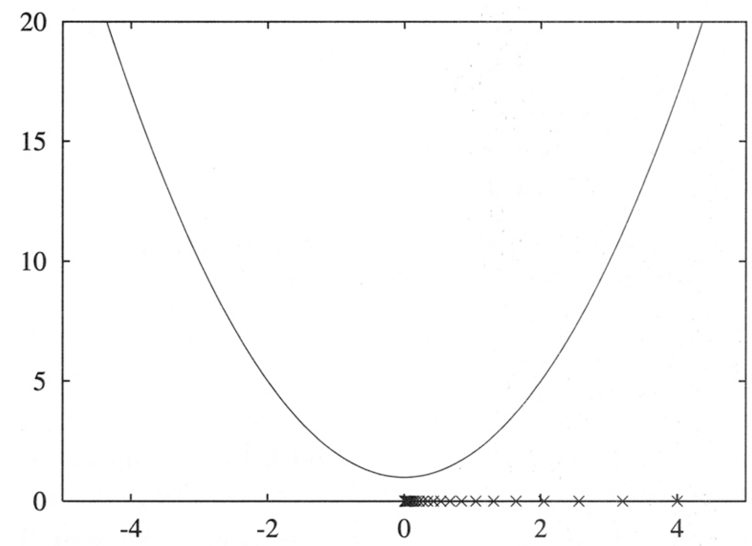
$w \leftarrow w - \alpha f'(w)$

Starting from an initial value $w = 4$, with the step size of 0.1:

- $4 - (0.1 \times 2 \times 4) = 3.2$
- $3.2 - (0.1 \times 2 \times 3.2) = 2.56$
- $2.56 - (0.1 \times 2 \times 2.56) = 2.048$

. . . . . .

– Stops when the change in parameter value becomes too small



Simulated Annealing Search

- Idea:
  - Efficiency of valley-descending + completeness of random walk
  - Escape local minima by allowing some "bad" moves

    But gradually decrease their step size and frequency
- Analogy with annealing:
  - At fixed temperature T, state occupation probability reaches Boltzman distribution $p(x) = \alpha e^{-E(x)/kT}$
  - T decreased slowly enough $\rightarrow$ always reach the best state
  - Devised by Metropolis *et al*., 1953, for physical process modeling
  - Widely used in VLSI layout, airline scheduling, etc.

Simulated Annealing Search

function SIMULATED-ANNEALING(*problem, schedule*) **returns** a solution state
  inputs: *problem*, a problem
    schedule, a mapping from time to "temperature"

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

Simulated Annealing Search

- A random move is picked instead of the best move
- If the move improves the situation, it is always accepted
- Otherwise, the move is accepted with probability $e^{-\Delta E/T}$
  - ΔE : the amount by which the evaluation is worsened
    - The acceptance probability decreases exponentially with the "badness" of the move
  - T : temperature, determined by the annealing schedule (controls the randomness)
    - Bad moves are more likely at the start when T is high
    - They become less likely as T decreases

Simulated Annealing Search

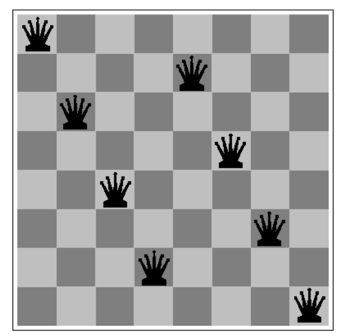
- $T \to 0$: simple hill-climbing (first-choice hill-climbing)
- If the annealing schedule lowers T slowly enough, a global optimum will be found with probability approaching 1
- The initial temperature is often heuristically set to a value so that the probability of accepting bad moves is 0.5

Genetic Algorithm

- Starts with a population of individuals
  - Each individual (state) is represented as a string over a finite alphabet (called chromosome)—most commonly, a string of 0s and 1s
- Each individual is rated by the fitness function
  - An individual is selected for reproduction by the probability proportional to the fitness score

1	3	5	7	2	4	6	8

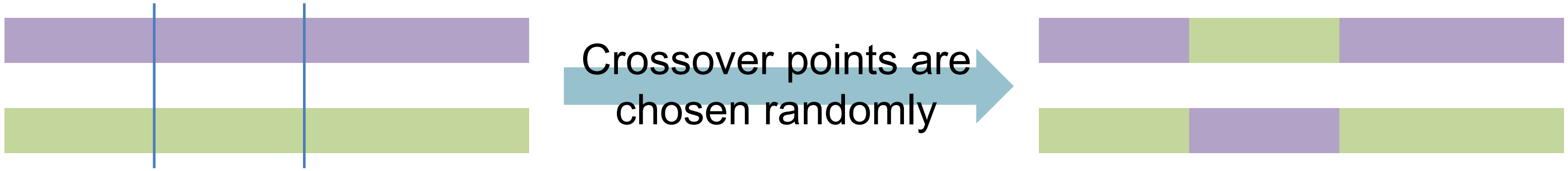
Column-by-column integer representation



Local search algorithms do not use any problem-specific heuristics
Simulated annealing and GA use meta-level heuristic $\rightarrow$ metaheuristic algorithms

Genetic Algorithm

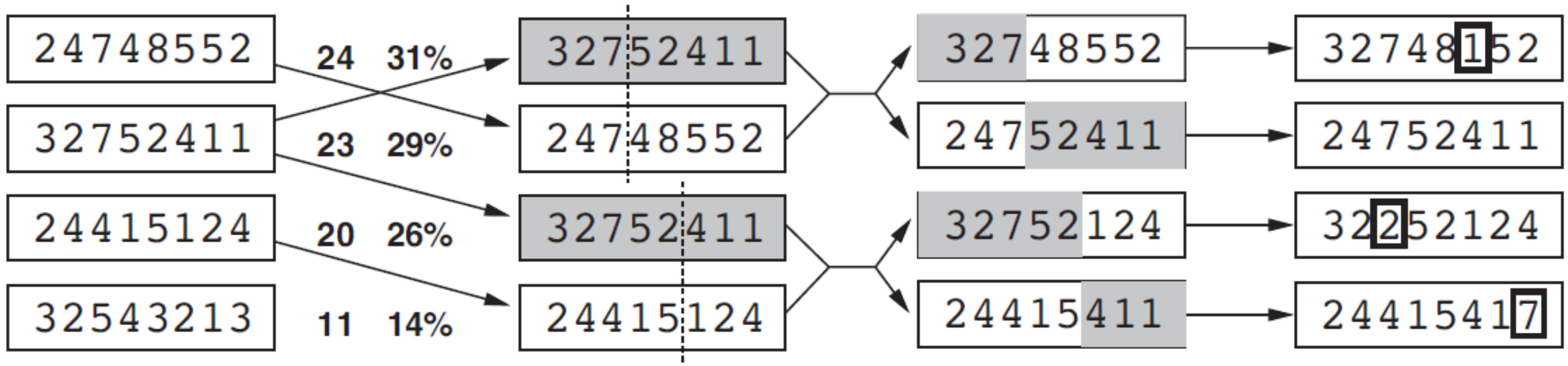
- Selected pair are mated by a crossover
  – Crossover frequently takes large steps in the state space early in the search process when the population is quite diverse, and smaller steps later on when most individuals are quite similar



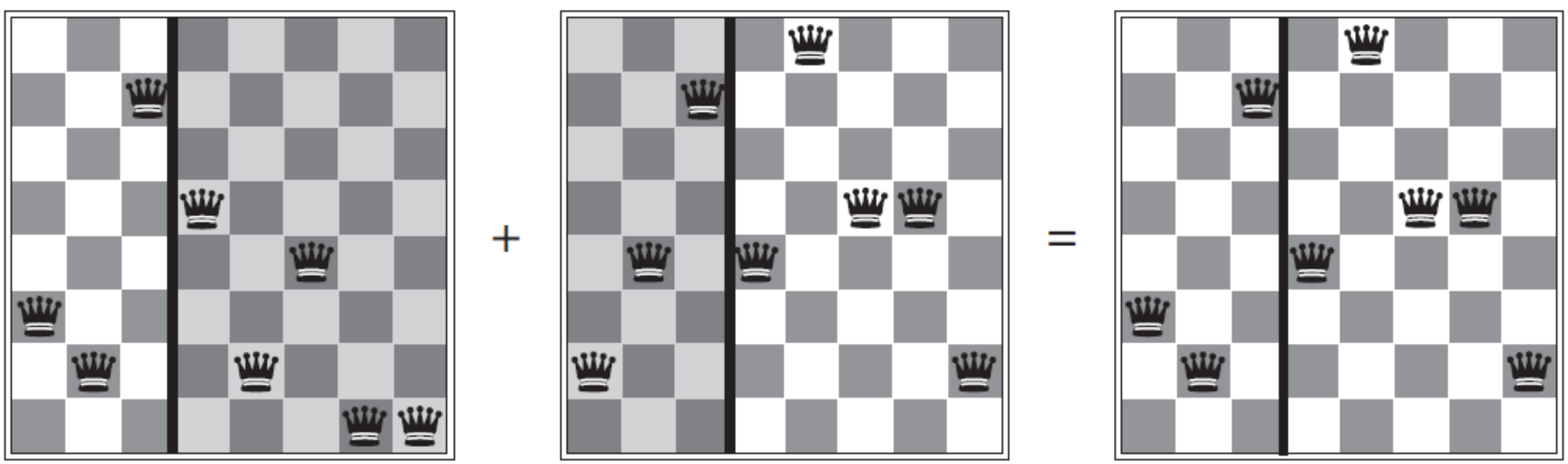
- Each locus is subject to random mutation with a small independent probability
- Advantage of GA comes from crossover:
  – Is able to combine large blocks of letters that have evolved independently to perform useful functions

  ➡ Raises the level of granularity at which the search operates

Genetic Algorithm



(a) Initial Population (b) Fitness Function (c) Selection (d) Crossover (e) Mutation



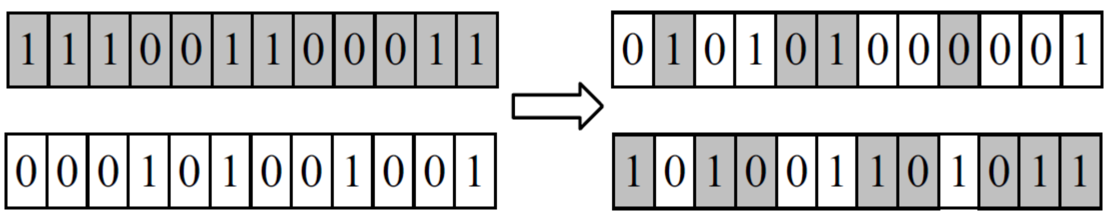
Genetic Algorithm

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
    for *i* = 1 **to** SIZE(*population*) **do**
      (*x*, *y*) ← SELECT-PARENTS(*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

Genetic Algorithm

- Parent selection by binary tournament:
  1. Randomly select 2 individuals with replacement
  2. Select the one with the best fitness as the winner (ties are broken randomly)
- Uniform crossover:
  - Each gene is chosen from either parent stochastically by flipping coin at each locus



  - If the probability of head $p = 0.5$, the average number of crossover points is $l/2$, where l is the length of the chromosome
  - $p = 0.2$ is a popular choice

Genetic Algorithm

- Crossover after parent selection is done according to the probability called crossover rate
  - Crossover rate close to 1 is popular
- Bit-flip mutation for binary representation:
  - Each bit is flipped with a small mutation probability called mutation rate
  - Mutation rate of $1/l$ is popular

Genetic Algorithm

Example: Binary encoding/decoding for numerical optimization

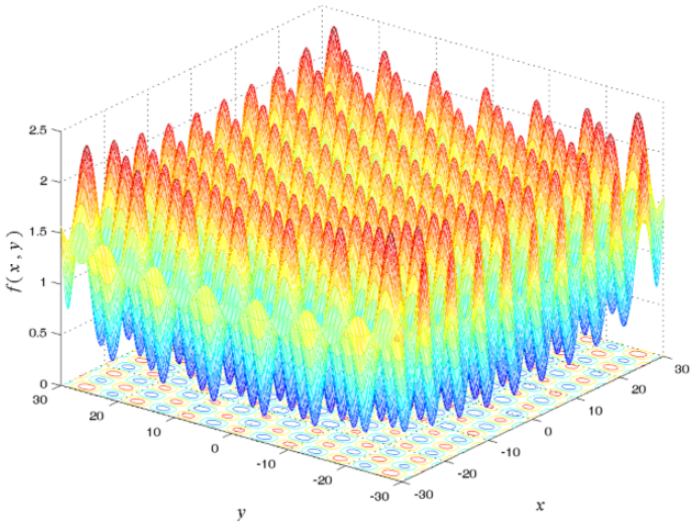
$$\min f(x,y) = \frac{x^2+y^2}{4000} - \cos(x)\cos\left(\frac{y}{\sqrt{2}}\right) + 1 \quad (-30 \le x, y \le 30)$$

– Assuming a 10-bit binary encoding for each variable, the code shown below can be decoded as

$$-30 + \left(30-(-30)\right)\times\frac{1}{2^{10}}(2^8+2^4+2^0)$$

$$= -14.004$$

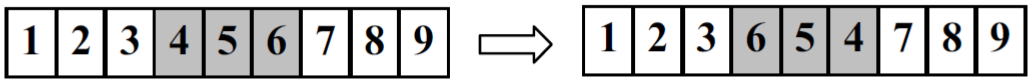
0	1	0	0	0	1	0	0	0	1



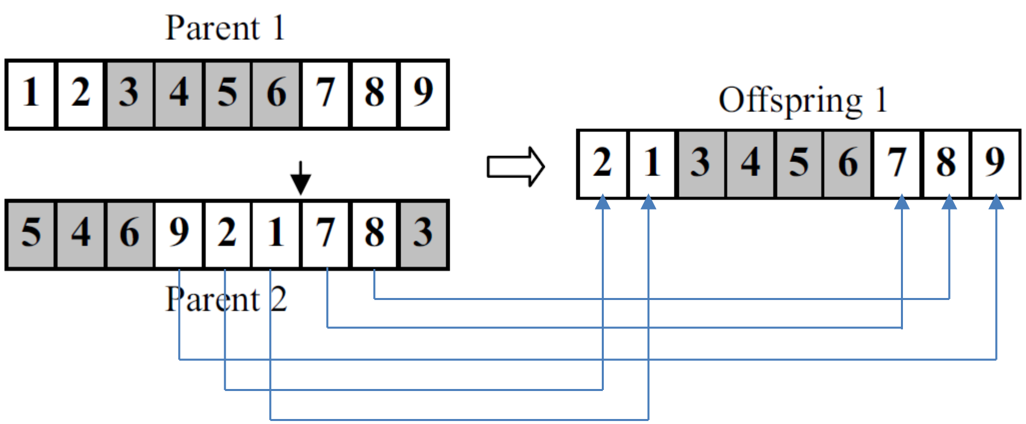
2-D Griewank function

Genetic Algorithm

- Genetic operators for permutation code (e.g., for TSP):
  - Simple inversion mutation:



  - Ordered crossover (OX):



Genetic Algorithm

- In the case of permutation code
  - Crossover rate is the probability of whether or not to perform the ordered crossover
  - Similarly, mutation rate is the probability of whether or not to perform the inversion