Evolutionary Algorithms

Mihai Oltean
moltean@cs.ubbcluj.ro
www.cs.ubbcluj.ro/~moltean

STRUCTURE

- **Representation**
- **Initialization**
- **Crossover / Recombination**
- **Mutation**
- Selection
- **Fitness function**
- Elitism
- Algorithm

REPRESENTATION

- Real $C_r = (x_1, x_2, \ldots, x_n)$, x_i □ □
  - Function optimization, Solving equations
- Binary $C_b = (b_1, b_2, \ldots, b_n)$, b_i □ $\{0, 1\}$
  - Knapsack problem, Function optimization
- Integer $C_I = (I_1, I_2, \ldots, I_n)$, I_i □ $\{0, 9\}$
- Permutations $C_p = (5, 3, 4, 1, 2)$
  - TSP problem
- Trees
  - Prediction, classification, computing primitives

Initialization

- Real
  - x_i = $MinX+rand(0,1)$*($MaxX$-$MinX$)
- Binary
  - b_i = rand() % 2
- Integer
  - I_i = rand() % 10

PERMUTATION INITIALIZATION

- Generate identical permutation

  Apply n times random switch

- Knuth shuffle
  - for i=1 to n do $a[i] = i$;
  - for i=1 to $n-1$ do
    - $swap$[$a[i]$, a[$Random[i,n]$]];

Crossover – real representation

- Two offspring
  - $o_1 = p_1 * \square + p_2 * (1- \square)$
  - $o_2 = p_2 * \square + p_1 * (1- \square)$

- One offspring
  - $o = (p_1+p_2) / 2$

CROSSOVER – BINARY REPRESENTATION

- 1 cutting point

  101 0100   101**1101**

  -----→

  1101101   **110**0100

- 2 cutting points

  10 1010 0   10**0110**0

  -----→

  1101101   **11**1010**1**

CROSSOVER – BINARY REPRESENTATION

- Uniform crossover

  1010100   10**011**0**1**

  -----→

  1101101   **11**101**0**0

Crossover – Integer representation

- The same as in the case of binary representation

Crossover - permutations

- With corrections
  - Apply an operator from binary representation and then correct the solution

- Without corrections
  - Extract the common edges and link the other nodes randomly

Crossover probability p_c

- Applied outside of the operator
- $p_c = 0.9$ (a big value)

- If ($p <= p_c$)
  - Apply crossover
- else
  - Offspring = Parents

Mutation – real encoding

- $o_i = x_i \pm r$
  - *r – small, fixed value*
  - *r – Gaussian value*
- Check for overflow

Mutation – binary encoding

- Weak $b_i = \text{rand}()\ \%\ 2$
- Strong $b_i = 1 - b_i$

Mutation – integer encoding (b>2)

- I_i = rand() % 10

Mutation - permutations

- Switch 2 positions

Mutation probability p_M

- Applied inside
- $p_m = 0.01$ (a small value)

- If ($p <= p_m$)
  - Mutate gene
- else
  - Gene not mutated

Fitness

- Problem dependent

- TSP – length of the road
- 8-queens – number of attacks
- Function minimization – $f(x)$

Selection

- Selection for reproduction
- Selection for survival

Random Selection

- Randomly choose an individual

Tournament selection

- Binary selection
  - randomly select two individuals and choose the best one

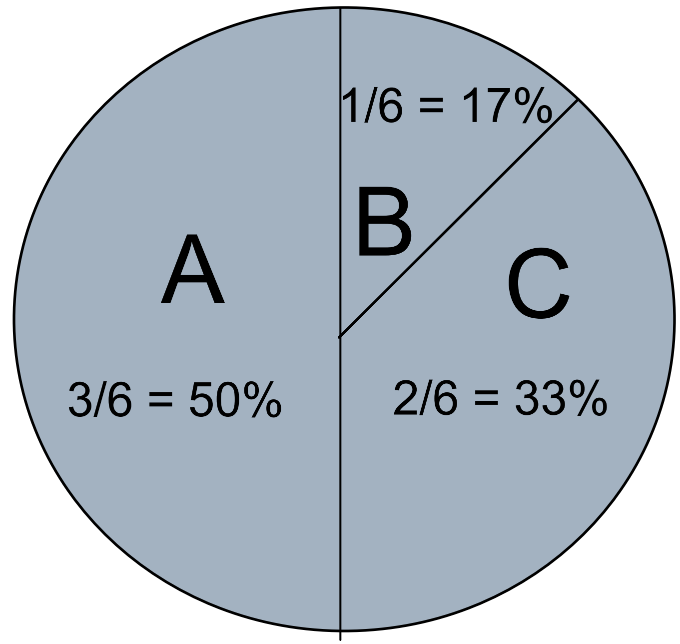
- *q*-tournament selection
  - randomly select *q* individuals and choose the best one

Roulette selection

fitness(A) = 3

fitness(B) = 1

fitness(C) = 2



Ordered selection

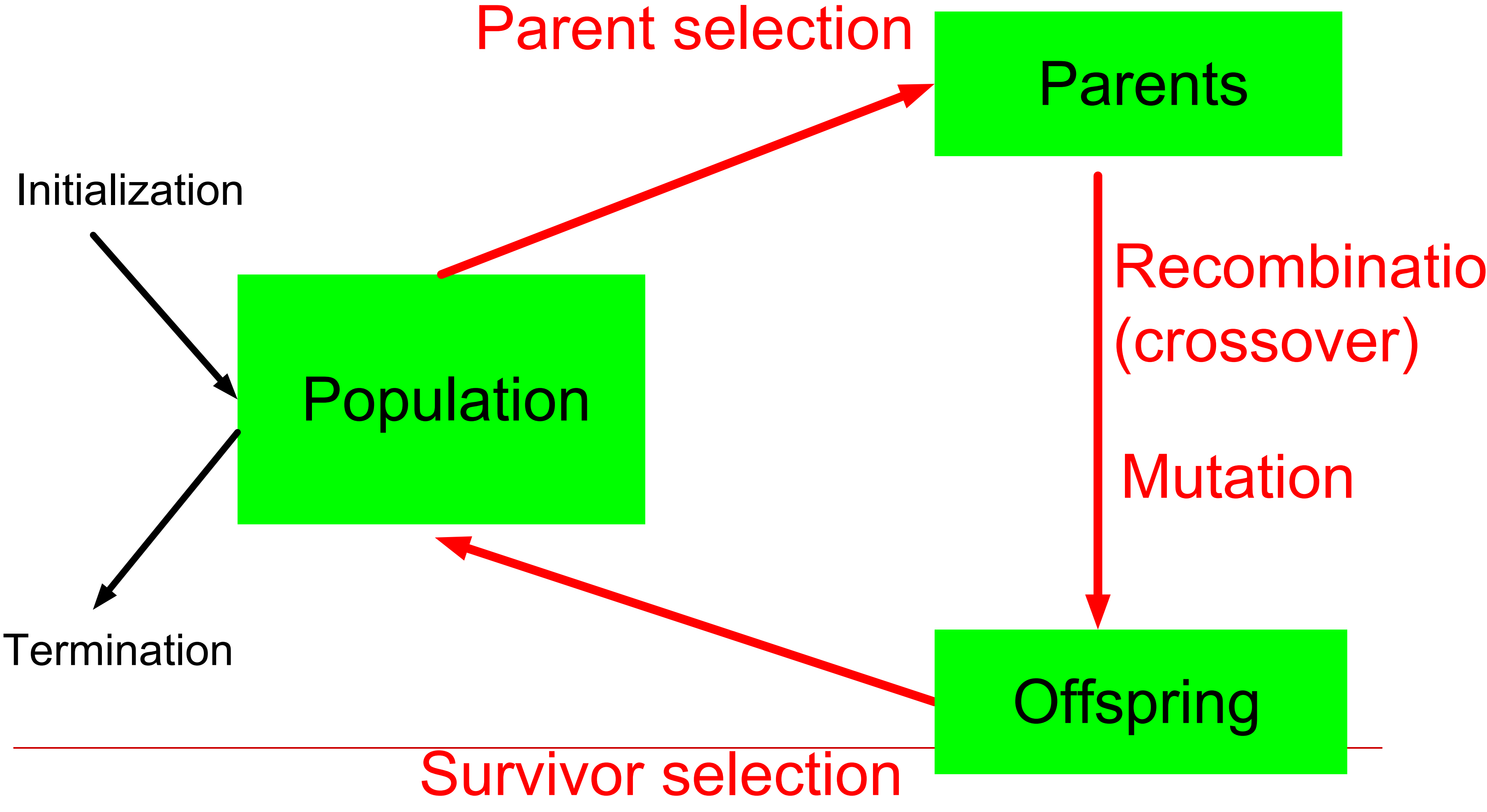
- Why? – sometimes there are big differences between the fitness of individuals.

- Sort the individuals based on fitness and apply roulette selection using the positions of individuals instead of their fitness.

Elitism

- Save the best individual found so far

General Scheme



Algorithms

- Random Search
- Evolution Strategy
- Genetic Algorithm
- GA with Steady State
- Evolutionary Programming

Random Search

- Repeat a fixed number of generations
  - Generate a random solution
- Print the best solution found

(1+1) Evolution Strategy

- Randomly generate a solution *p*
- Repeat for a fixed number of generations
  - *q* = Mutate(*p*)
  - If *q* better than *p*
    - *p* = *q*
- End Repeat
- Print the best solution found (*p*)

Repeated (1+1) Evolution Strategy

- Repeat (1+1) ES *n* time
- Print the best solution found

(1+LAMBDA) EVOLUTION STRATEGY

- Randomly generate a solution p
- Repeat for a fixed number of generations
  - q_k = Mutate(p); // k = 1, lambda
  - if q_k better than p
    - $p = q_k$
- End Repeat
- Print the best solution found (p)

(MU+LAMBDA) EVOLUTION STRATEGY

- Randomly generate a population of *mu* individuals
- Repeat for a fixed number of generations
- Repeat until *lambda* offspring have been generated.
- Choose several parents
- Recombine them
- Mutate the offspring
- End Repeat
- Keep the best *mu* individuals out of (*mu*+*lambda*) as parents of the next generation
- End Repeat
- Print the best solution found

(MU, LAMBDA) EVOLUTION STRATEGY

- Randomly generate a population of *mu* individuals
- Repeat for a fixed number of generations
  - Repeat until *lambda* (*lambda* > *mu*) offspring have been generated.
- Choose several parents
- Recombine them
- Mutate the offspring
  - End Repeat
  - Keep the best *mu* individuals out of *lambda* as parents of the next generation
- End Repeat
- Print the best solution found

Evolutionary Programming

- Randomly generate a population of *n* individuals
- Repeat for a fixed number of generations
  - Repeat until *n* offspring have been generated.
- Mutate each individual -> offspring
  - End Repeat
  - Keep the best *n* individuals out of 2**n* as parents of the next generation
- End Repeat
- Print the best solution found

GENETIC ALGORITHM

- Randomly generate a population of *n* individuals
- Repeat for a fixed number of generations
  - Copy the best to the next generation
  - Create a Mating Pool using a selection procedure
  - Repeat until *n-1* offspring have been generated.
    - Randomly choose 2 parents from MP
    - Recombine the parents -> offspring
    - Mutate the offspring
    - Copy the offspring into the next generation
  - End Repeat
- End Repeat
- Print the best solution found

GENETIC ALGORITHM (V2) – NO MP

- Randomly generate a population of *n* individuals
- Repeat for a fixed number of generations
  - Copy the best to the next generation
  - Repeat until *n-1* offspring have been generated.
    - Choose 2 parents from current population by using a selection procedure
    - Recombine the parents -> offspring
    - Mutate the offspring
    - Copy the offspring into the next generation
  - End Repeat
- End Repeat
- Print the best solution found

Genetic Algorithm with Steady-State

- Randomly generate a population of *n* individuals
- Repeat until *n*NumGens* offspring have been created
  - Randomly choose 2 parents from current pop
  - Recombine the parents -> offspring
  - Mutate the offspring
  - Replace the worst individual in the current population with the offspring, only if the offspring is better
- End Repeat
- Print the best solution found

GA with Local Steady-State

- Randomly generate a population of *n* individuals
- Repeat until *n*NumGen* offspring have been created
  - Randomly choose 4 individuals from current pop
  - The best 2 individuals are the parents
  - Recombine the parents -> offspring
  - Mutate the offspring
  - Replace the worst 2 by offspring
- End Repeat
- Print the best solution found

No Free Lunch (NFL) theorems

- There is no "the best" EA for all problems !!!

- For search and for optimization
  - The average performance of any pair of algorithms across all problems is the same.
  - We cannot use the algorithm's behavior so far in order to predict its future behavior.