Genetic Algorithm

Unit-11

What is Evolutionary Computation?

- An abstraction from the theory of biological evolution that is used to create optimization procedures or methodologies, usually implemented on computers, that are used to solve problems.
- Evolution has optimized biological processes; therefore
- Adoption of the evolutionary paradigm to computation and other problems can help us find optimal solutions

Components of Evolutionary Computing

- Genetic Algorithms
  - invented by John Holland (University of Michigan) in the 1960's
- Evolution Strategies
  - invented by Ingo Rechenberg (Technical University Berlin) in the 1960's
- Started out as individual developments, but have begun to converge in the last few years

Genetic Algorithm (Holland)

- heuristic method based on ‘ survival of the fittest ’
- useful when search space very large or too complex for analytic treatment
- in each iteration (generation) possible solutions or individuals represented as strings of numbers

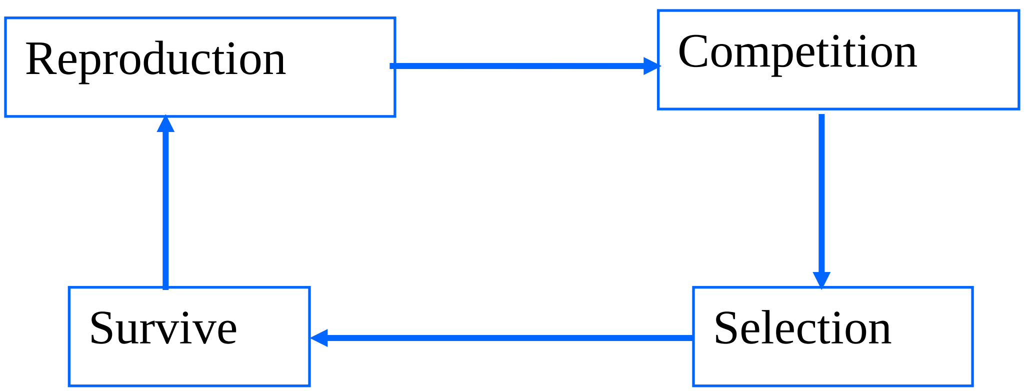
3021 3058 3240

00010101 00111010 11110000
00010001 00111011 10100101
00100100 10111001 01111000

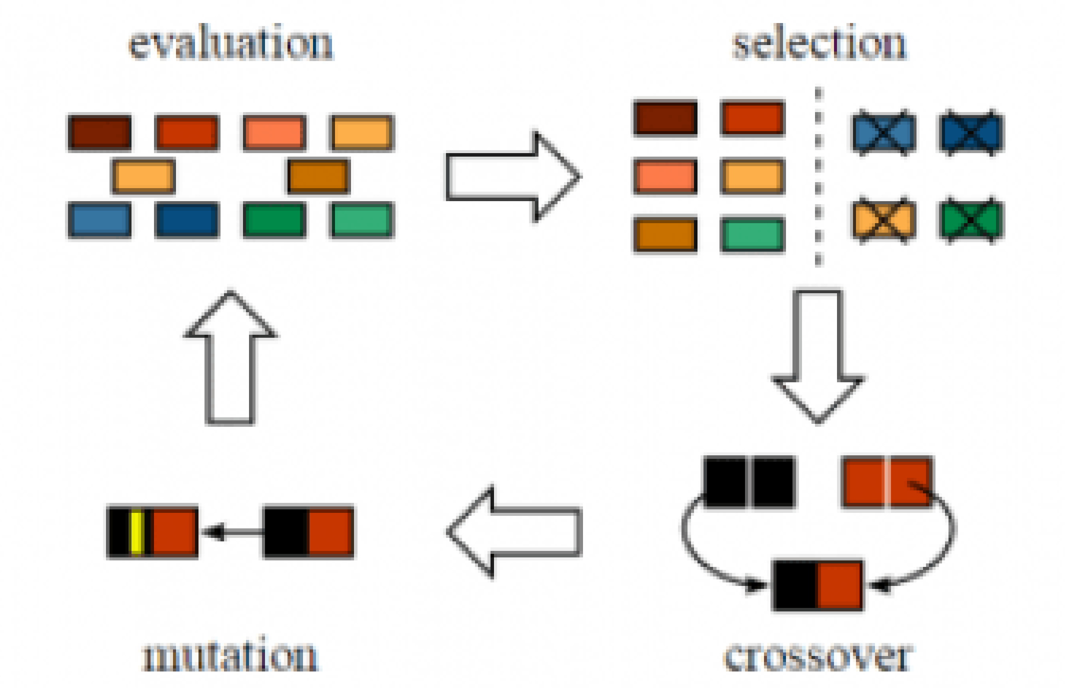
11000101 01011000 01101010

Darwinian Paradigm

- Intrinsically a robust search and optimization mechanism

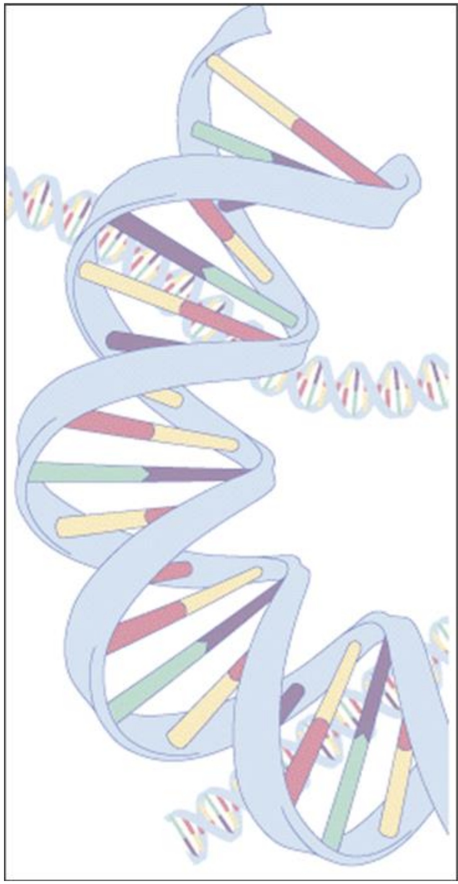


Darwinian Genetic Paradigm



What is Genetic Algorithm?

- Follows steps inspired by the biological processes of evolution.
- Follow the idea of SURVIVAL OF THE FITTEST- Better and better solutions evolve from previous generations until a near optimal solution is obtained.
- Genetic Algorithms are often used to improve the performance of other AI methods.
- The method learns by producing offspring that are better and better as measured by a fitness function.



Genetic Algorithm

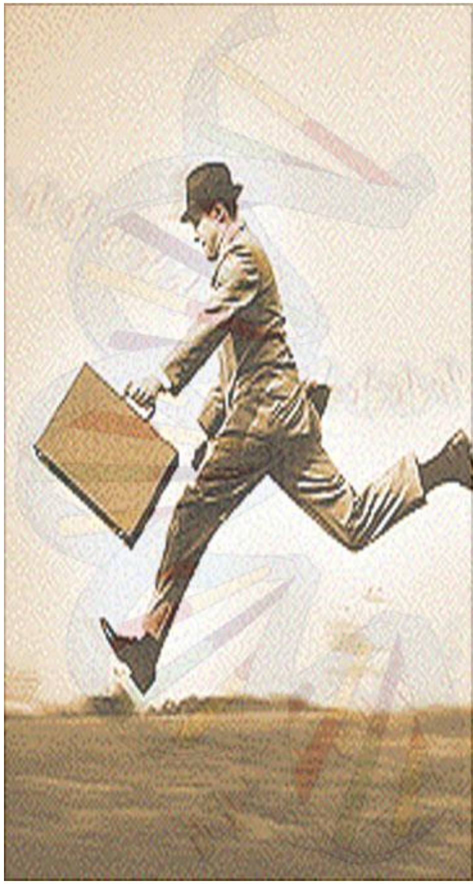
- all individuals in population evaluated by fitness function
- individuals allowed to reproduce (selection), crossover, mutate

Representations

- Genetic programming can be used to evolve S-expressions, which can be used as LISP programs to solve problems.
- A string of bits is known as a **chromosome.**
- Each bit is known as a **gene.**
- Chromosomes can be combined together to form **creatures**.
- We will see how genetic algorithms can be used to solve mathematical problems.

Knapsack Problem

- You are going on a picnic.
- And have a number of items that you could take along.
- Each item has a weight and a benefit or value.
- You can take one of each item at most.
- There is a capacity limit on the weight you can carry.
- You should carry items with max. values.



Example: (Knapsack Problem)

• **Item:**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
• **Benefit:**	**5**	**8**	**3**	**2**	**7**	**9**	**4**
• **Weight:**	**7**	**8**	**4**	**10**	**4**	**6**	**4**

- **Knapsack holds a maximum of 22 pounds**
- **Fill it to get the maximum benefit**

Outline of Basic Genetic Algorithm

1. **[Start]**
   - ✓ Encoding: represent the individual.
   - ✓ Generate random population of *n* chromosomes (suitable solutions for the problem).
2. **[Fitness]** Evaluate the fitness of each chromosome.
3. **[New population]** repeating following steps until the new population is complete.
4. **[Selection]** Select the best two parents.
5. **[Crossover]** cross over the parents to form a new offspring (children).
6. **[Mutation]** With a mutation probability.
7. **[Accepting]** Place new offspring in a new population.
8. **[Replace]** Use new generated population for a further run of algorithm.
9. **[Test]** If the end condition is satisfied, then **stop**.
10. **[Loop]** Go to step **2** .

Basic Steps

- Encoding: 0 = not exist, 1 = exist in the Knapsack

  Chromosome: 1010110

Item.	1	2	3	4	5	6	7
Chro	1	0	1	0	1	1	0
Exist?	y	n	y	n	y	y	n

  => Items taken: 1, 3 , 5, 6.

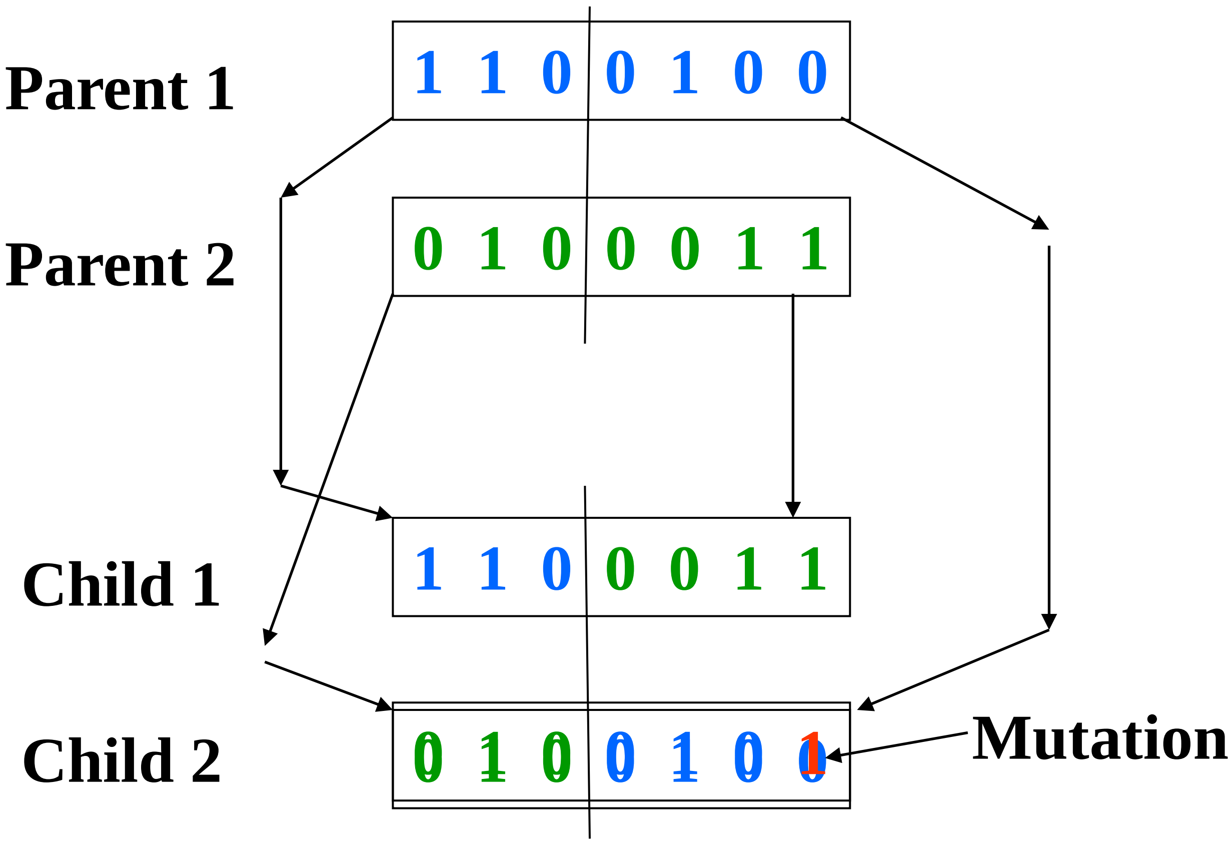
- Generate random population of *n* chromosomes:

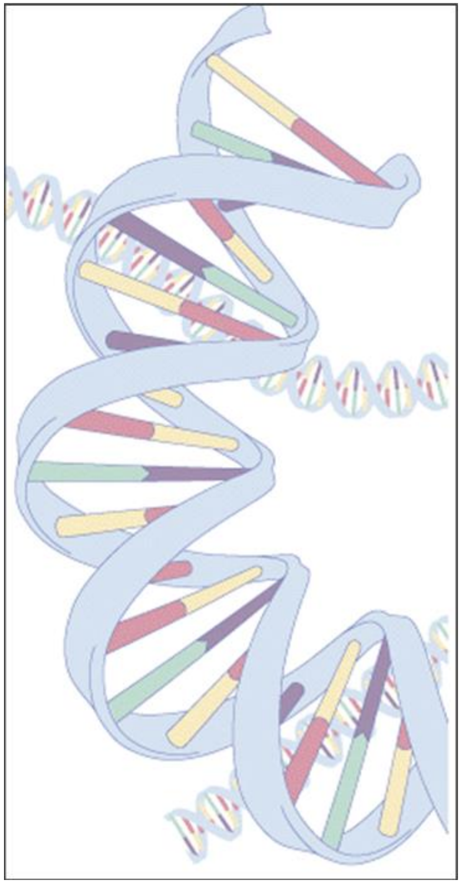
  a) 0101010

  b) 1100100

  c) 0100011

Crossover & Mutation





Accepting, Replacing & Testing

✓Place new offspring in a new population.

✓Use new generated population for a further run of algorithm.

✓If the end condition is satisfied, then **stop**. End conditions:

- Number of populations.
- Improvement of the best solution.

✓Else, return to step 2 **[Fitness].**

The Algorithm

GA(*Fitness, Fitness_threshold, p, r, m*)

- *Initialize*: $P \leftarrow p$ random hypotheses
- *Evaluate*: for each *h* in *P*, compute *Fitness*(h)
- While [$\max_h$ *Fitness*(*h*)] < *Fitness_threshold*
  1. *Select*: Select $(1 - r)$ members of *P* to add to P_S based on fitness

$$\Pr(h_i) = \frac{Fitness(h_i)}{\sum_{j=1}^{P} Fitness(h_j)}$$

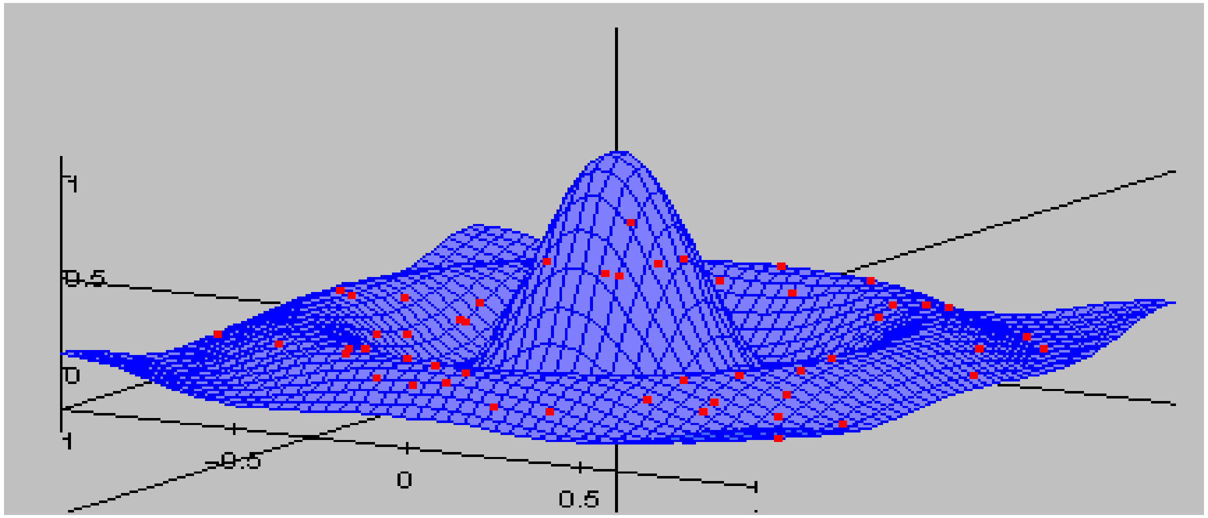
  2. *Crossover*: Probabilistically select pairs of hypotheses from *P*. For each pair, $\langle h_1, h_2 \rangle$, produce two offspring by applying the Crossover operator. Add all offspring to P_S
  3. *Mutate*: Invert a randomly selected bit in $m \cdot p$ random members of P_S
  4. *Update*: $P \leftarrow P_S$
  5. *Evaluate*: for each *h* in *P*, compute *Fitness*(*h*)
- Return the hypothesis from *P* that has the highest fitness

Fitness Function

- Fitness is an important concept in genetic algorithms.
- The fitness of a chromosome determines how likely it is that it will reproduce.
- Fitness is usually measured in terms of how well the chromosome solves some goal problem.
  - E.g., if the genetic algorithm is to be used to sort numbers, then the fitness of a chromosome will be determined by how close to a correct sorting it produces.
- Fitness can also be subjective (aesthetic)
- For each individual in the population, evaluate its relative fitness
- For a problem with m parameters, the fitness can be plotted in an m+1 dimensional space

Sample Search Space

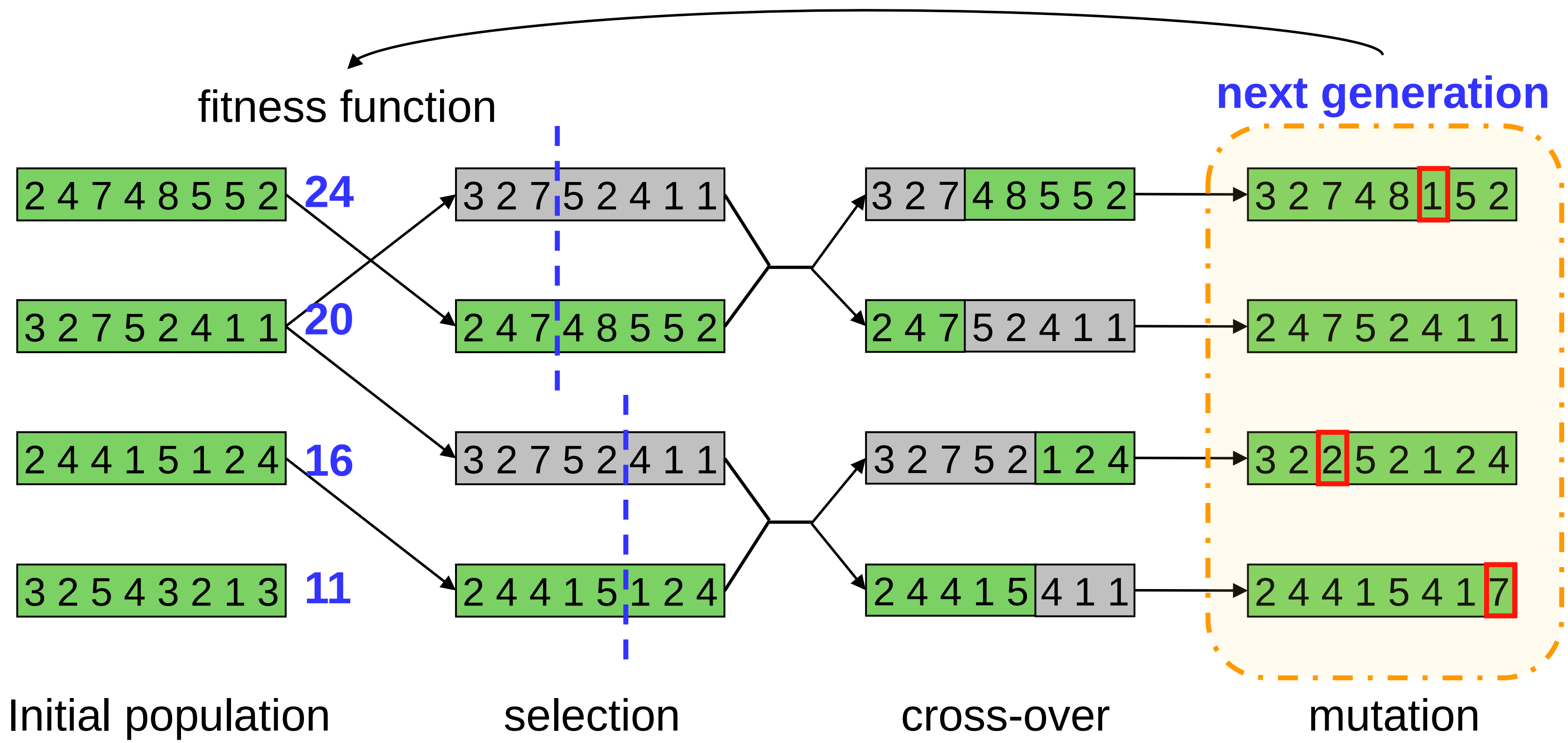
- A randomly generated population of individuals will be randomly distributed throughout the search space



Generations

- As each new generation of *n* individuals is generated, they replace their parent generation
- To achieve the desired results, 500 to 5000 generations are required

Generation

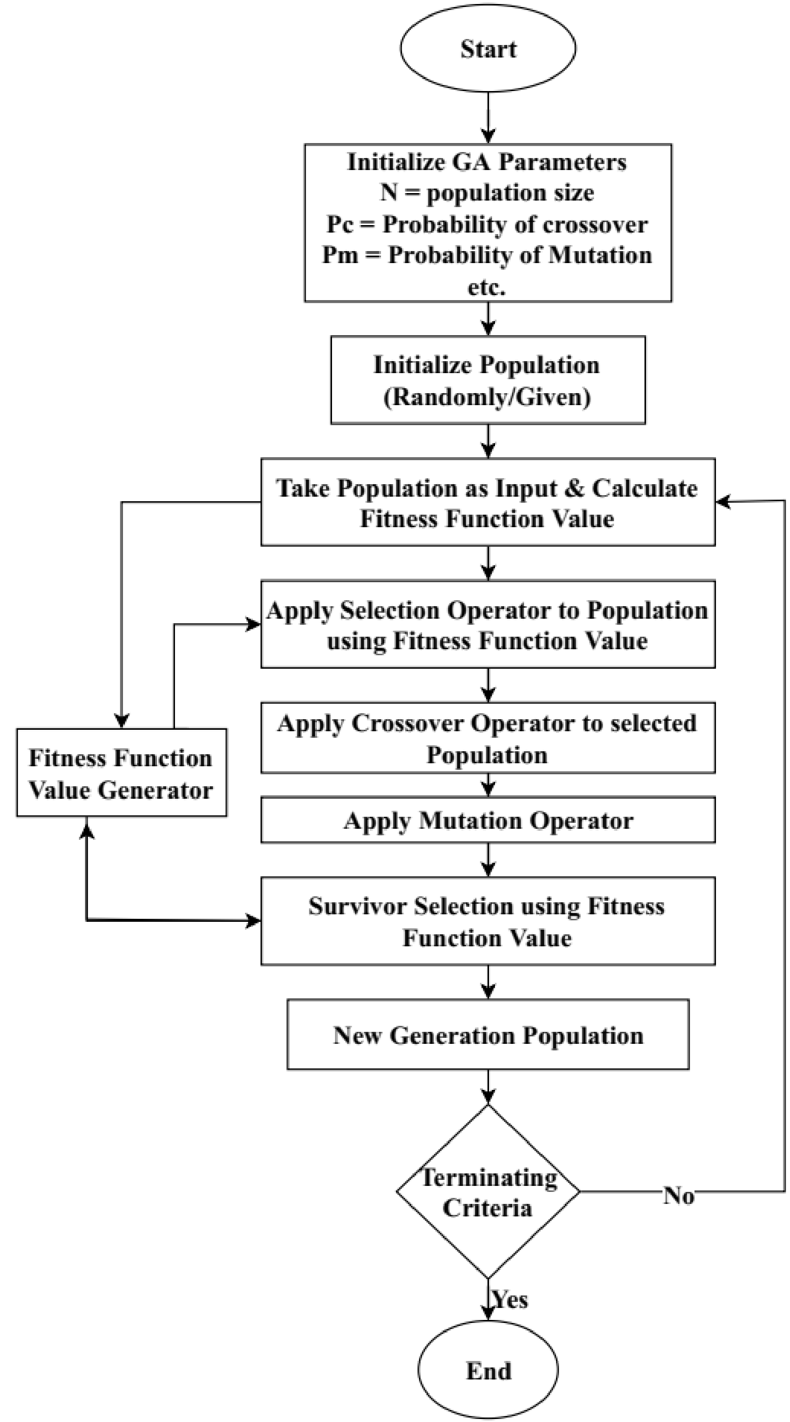


fitness function
next generation
2 4 7 4 8 5 5 2
24
3 2 7 5 2 4 1 1
20
2 4 4 1 5 1 2 4
16
3 2 5 4 3 2 1 3
11
3 2 7 5 2 4 1 1
2 4 7 4 8 5 5 2
3 2 7 5 2 4 1 1
2 4 4 1 5 1 2 4
3 2 7 4 8 5 5 2
2 4 7 5 2 4 1 1
3 2 7 5 2 1 2 4
2 4 4 1 5 4 1 1
3 2 7 4 8 1 5 2
2 4 7 5 2 4 1 1
3 2 2 5 2 1 2 4
2 4 4 1 5 4 1 7
Initial population
selection
cross-over
mutation

Reproduction

- Crossover
  - Two parents produce two offspring
  - There is a chance that the chromosomes of the two parents are copied unmodified as offspring
  - There is a chance that the chromosomes of the two parents are randomly recombined (crossover) to form offspring
  - Generally the chance of crossover is between 0.6 and 1.0
- Mutation
  - There is a chance that a gene of a child is changed randomly
  - Generally the chance of mutation is low (e.g. 0.001)

GA Flow Chart



★ Uniform Crossover

Mask	0	1	1	0	1	1
parent 1	1	0	0	1	0	0
parent 2	0	1	0	1	1	0
child 1	1	1	0	1	1	0
child 2	0	0	0	1	0	0

Mutation

Types → Bit Flip Mutation
→ Swap Mutation
→ Inversion Mutation

⇒ Bit Flip Mutation

1	0	1	1	(Before)

↓

1	0	0	1	(After)

⇒ Swap Mutation

3	$\boxed{1}$	6	5	$\boxed{4}$	(Before)

↓

3	$\boxed{4}$	6	5	$\boxed{1}$	(After)

⇒ Inversion Mutation

3	1	6	5	4

↓

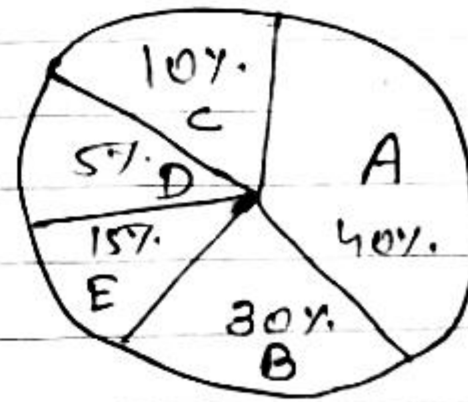
3	5	6	1	4

8. Selection.
↳ Roulette Wheel
↳ Rank based.
↳ Tournament.

A Roulette wheel.

A	2.0	40%
B	1.5	30%
C	0.5	10%
D	0.25	5%
E	0.75	15%
	$\Sigma f_i = 5$	

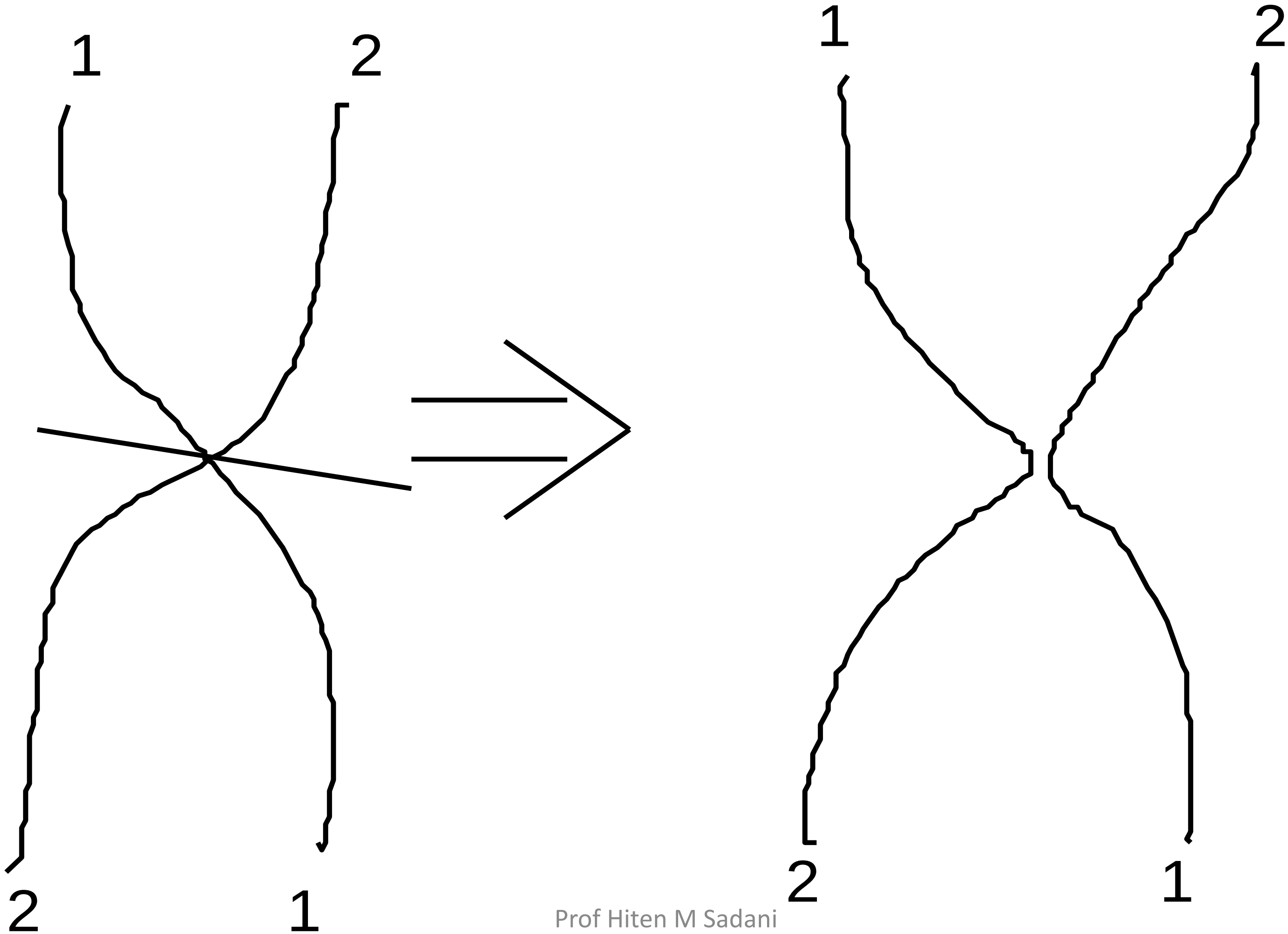
$$P_i = \frac{F_i}{\Sigma F_i}$$



Crossover

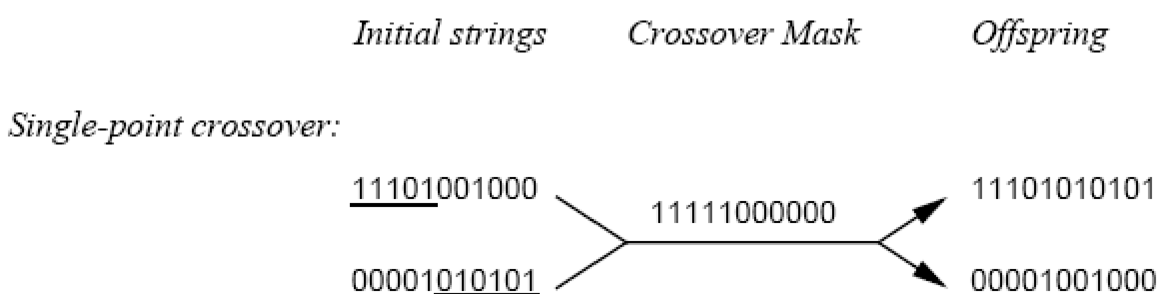
- Crossover
  - Generating offspring from two selected parents
    - Single point crossover
    - Two point crossover (Multi point crossover)
    - Uniform crossover

One-point crossover - Nature



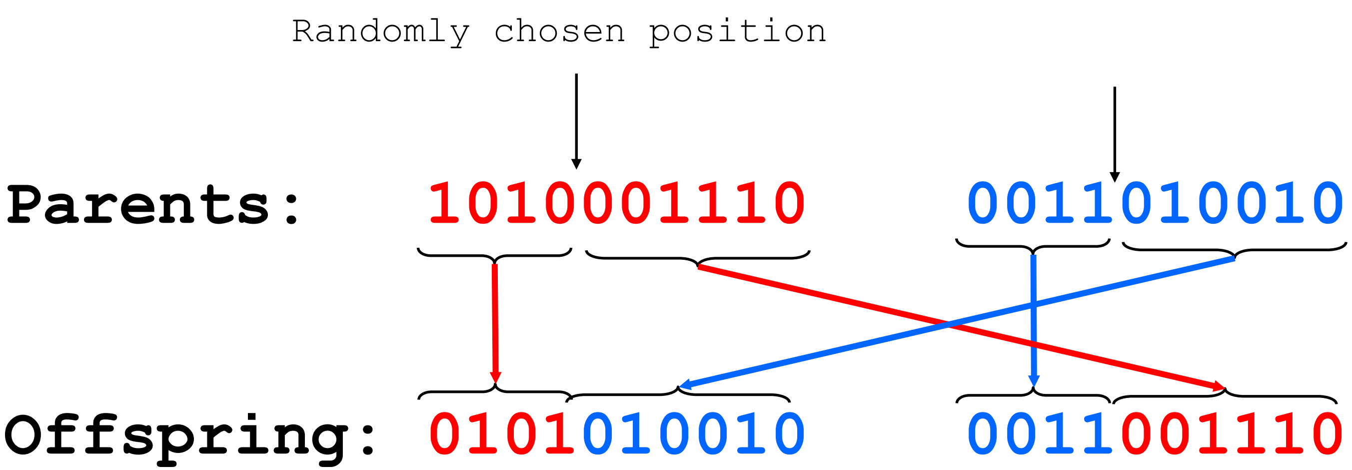
Single-Point Crossover (One-point crossover)

- **Crossover is applied as follows:**
  1) Select a random crossover point.
  2) Break each chromosome into two parts, splitting at the crossover point.
  3) Recombine the broken chromosomes by combining the front of one with the back of the other, and vice versa, to produce two new chromosomes.



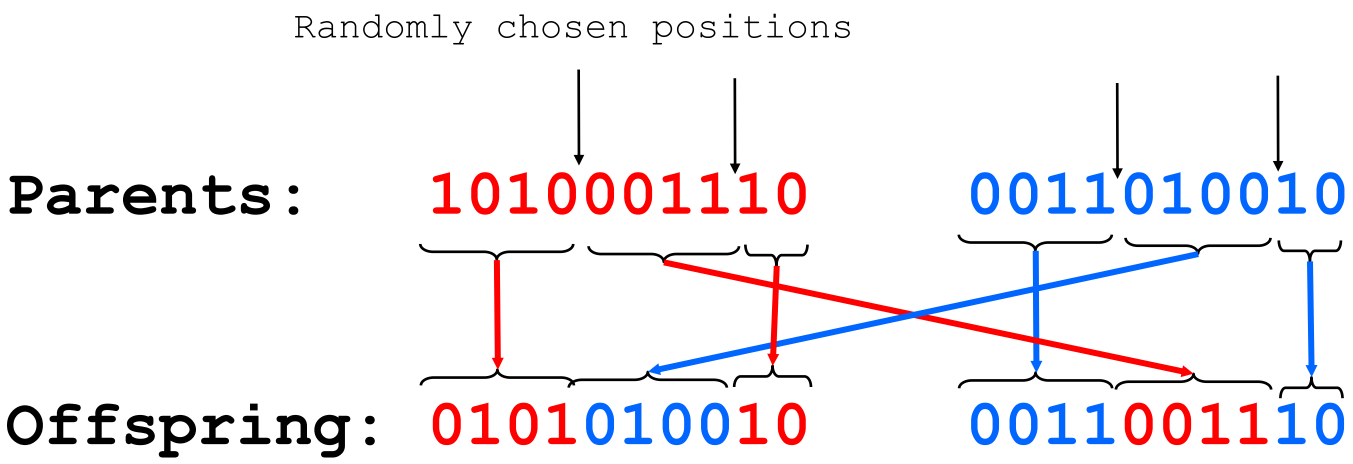
One-point crossover

- Randomly one position in the chromosomes is chosen
- Child 1 is head of chromosome of parent 1 with tail of chromosome of parent 2
- Child 2 is head of 2 with tail of 1



Two-point crossover

- Randomly two positions in the chromosomes are chosen
- Avoids that genes at the head and genes at the tail of a chromosome are always split when recombined



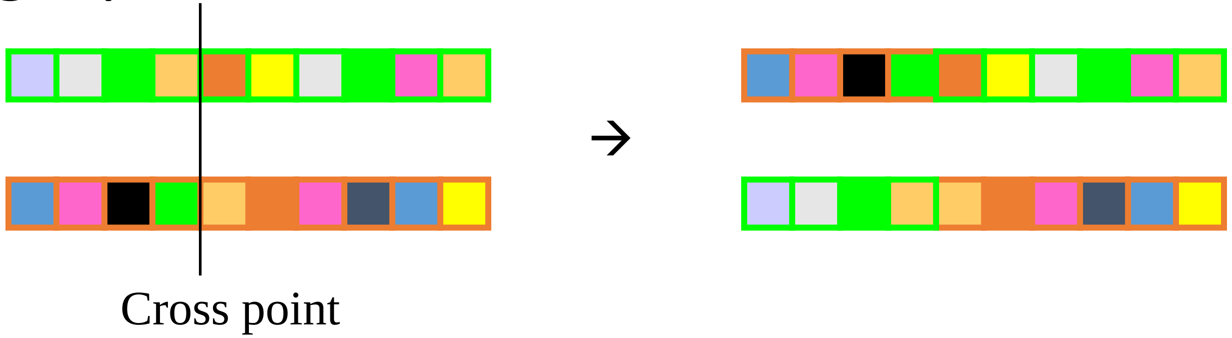
Uniform crossover

- A random mask is generated
- The mask determines which bits are copied from one parent and which from the other parent
- Bit density in mask determines how much material is taken from the other parent (takeover parameter)

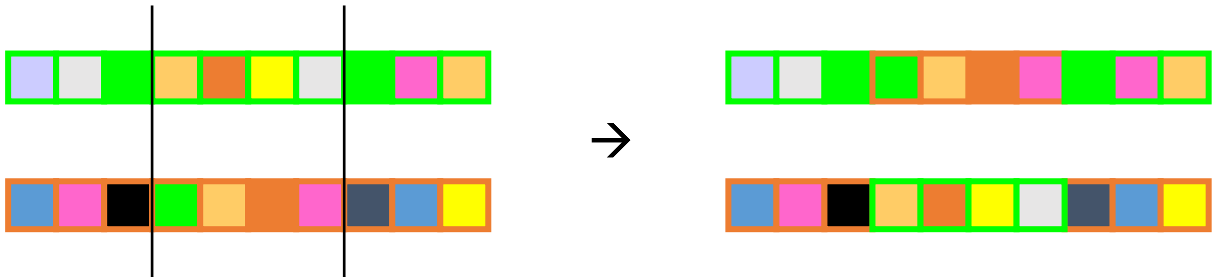
```
Mask:       0110011000     (Randomly generated)
Parents:    1010001110     0011010010

Offspring:  0011001010     1010010110
```

Operators comparison

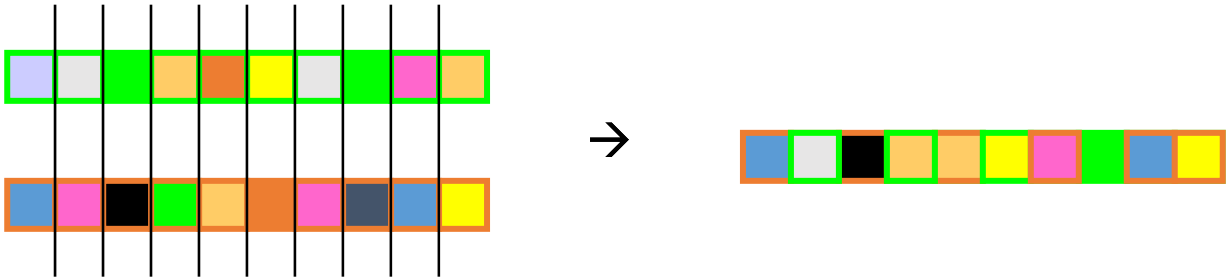
- Single point crossover



- Two point crossover (Multi point crossover)



- Uniform crossover
- Is uniform crossover better than single crossover point?
  - Trade off between
    - Exploration: introduction of new combination of features
    - Exploitation: keep the good features in the existing solution

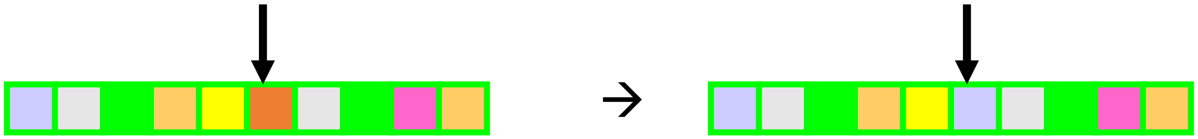


Problems with crossover

- Depending on coding, simple crossovers can have high chance to produce illegal offspring
  - E.g. in TSP with simple binary or path coding, most offspring will be illegal because not all cities will be in the offspring and some cities will be there more than once
- Uniform crossover can often be modified to avoid this problem
  - E.g. in TSP with simple path coding:
    - Where mask is 1, copy cities from one parent
    - Where mask is 0, choose the remaining cities in the order of the other parent

Mutation

- Generating new offspring from single parent



- Maintaining the diversity of the individuals
  - Crossover can only explore the combinations of the current gene pool
  - Mutation can “generate” new genes

Reproduction Operators

- Control parameters: **population size, crossover/mutation probability**
  - Problem specific
  - Increase population size
    - Increase diversity and computation time for each generation
  - Increase crossover probability
    - Increase the opportunity for recombination but also disruption of good combination
  - Increase mutation probability
    - Closer to randomly search
    - Help to introduce new gene or reintroduce the lost gene
- Varies the population
  - Usually using crossover operators to recombine the genes to generate the new population, then using mutation operators on the new population

Parent/Survivor Selection

- Strategies
  - Survivor selection
    - Always keep the best one
    - Elitist: deletion of the K worst
    - Probability selection : inverse to their fitness
    - Etc.

Parent/Survivor Selection

- Too strong fitness selection bias can lead to sub-optimal solution
- Too little fitness bias selection results in unfocused and meandering search

Parent selection

Chance to be selected as parent proportional to fitness

- Roulette wheel

To avoid problems with fitness function

- Tournament

Not a very important parameter

Parent/Survivor Selection

- Strategies
  - Parent selection
    - Uniform randomly selection
    - Probability selection : proportional to their fitness
    - Tournament selection (Multiple Objectives)
      - Build a small comparison set
      - Randomly select a pair with the higher rank one beats the lower one
        - Non-dominated one beat the dominated one
        - **Niche count**: the number of points in the population within certain distance, higher the niche count, lower the rank.
    - Etc.

Roulette wheel

- Sum the fitness of all chromosomes, call it T
- Generate a random number N between 1 and T
- Return chromosome whose fitness added to the running total is equal to or larger than N
- Chance to be selected is exactly proportional to fitness

Chromosome:	1	2	3	4	5	6
Fitness:	8	2	17	7	4	11
Running total:	8	10	27	34	38	49
N ($1 \leq N \leq 49$):			23			
Selected:			3			

Tournament

- Binary tournament
  - Two individuals are randomly chosen; the fitter of the two is selected as a parent
- Probabilistic binary tournament
  - Two individuals are randomly chosen; with a chance p, $0.5<p<1$, the fitter of the two is selected as a parent
- Larger tournaments
  - n individuals are randomly chosen; the fittest one is selected as a parent

- By changing n and/or p, the GA can be adjusted dynamically

Termination Criteria

- A genetic algorithm is run over a number of generations until the termination criteria are reached.
- Typical termination criteria are:
  - Stop after a fixed number of generations.
  - Stop when a chromosome reaches a specified fitness level.
  - Stop when a chromosome succeeds in solving the problem, within a specified tolerance.
- Human judgment can also be used in some more subjective cases.

Application of Genetic Algorithm

- Genetic Algorithms can be applied to virtually any problem that has a large search space.
- The military uses GAs to evolve equations to differentiate between different radar returns.
- Stock companies use GA-powered programs to predict the stock market.

Application of Genetic Algorithm

- Feature Selection
- Engineering Design
  - Engineering design has relied heavily on computer modeling and simulation to make design cycle process fast and economical. Genetic algorithm has been used to optimize and provide a robust solution.
- Traffic and Shipment Routing (Travelling Salesman Problem)
  - This is a famous problem and has been efficiently adopted by many sales-based companies as it is time saving and economical. This is also achieved using genetic algorithm.
- Robotics
  - Genetic algorithm is being used to create learning robots which will behave as a human and will do tasks like cooking our meal, do our laundry etc.

Drawbacks of GA

- Difficult to find an encoding for the problem
- Difficult to define a valid fitness function
- May not return the global maximum
- GA is nondeterministic – two runs may end with different results
- There's no indication whether best individual is optimal

Software

- GATSS, by Thomas Pederson, a Genetic Algorithm based solver of the Traveling Salesman problem written in GNU C++ linked to CGI-script.
- GAlib, A C++ Genetic Algorithms Library, see in particular the provided examples, the TSP.
- Maugis TSP solver, a program written in ansi-C, for Symmetric Euclidean TSPs, by Lionnel Maugis.
- tsp_solve, a collection of heuristics and optimal algorithms for the TSP, by Chad Hurwitz. He does not have a web page or ftp site to email him for a copy of the software, it's GNU C so it'll run on most unixes.
- An ATSP code, by Glenn Dicus, Bart Jaworski and Joseph Ou-Yang.
- A TSP program in Parlog, by Steve Gregory.
- The Travelling Spider Problem, by Moshe Sniedovich. We expect this page to include TSP codes based on Dynamic Programming.
- Traveling Salesman Problem solving program (TSPSolver), by Victor V. Miagkikh. TSPSolver is written in Borland C++ and Borlar Some benchmarks are enclosed.
- GRIN, a software package for graphs by Vitali Petchenkine.
- Traveling Salesman Java Applet, by Martin Hagerup. The author comments that the applet has been selected to appear in the book Dummies" (please, send me one) . Requires Netscape 2.0 in 32 bits or other Java-browser.
- Simple Closed Paths, from the book: *Introduction to Programming with Mathematica, 2nd Edition* , 1995, *TELOS*/Springer-V *Mathematica* notebook.
- tsp1.gms, a page related with GAMS, the General Algebraic Modeling System which is a high-level modeling system for mathematic
- A Java TSP demo program using Kohonen's neural network formulation.
- A Simple TSP-Solver: An ABACUS Tutorial, by Stefan Thienel, 1996.
- A Guided Local Search demo for TSP, by Christos Voudouris, (it requires Microsoft Windows 3.1.).
- Another Guided Local search demo, this one requires SunOS (precompiled for SunOS 5.3) and XView, by Christos Voudouris. A 95/NT is now available.
- A heuristic and a brute force method, Java programs by Aaron Passey.
- BOB: Branch-and-bound Optimization liBrary, a general-purpose parallel Branch-and-Bound algorithm library being developed at P University of Versailles-Saint Quentin en Yvelines. They provide examples for QAP, TSP and VCP. It is freely available via anonymou the file pub/software/BoBL1.0.tar.Z. Click here to get a copy of it.
- Concorde, a powerful code by by David Applegate, Robert Bixby, William Cook, and V. Chvatal. (for download, please use anony ftp.caam.rice.edu, change to directory /pub/people/bico/970827/ and download file cc970827.tgz). A must-see !
- David Neto's Lin-Kernighan ``cluster aware'' heuristic, by David Neto, a literate program written using the CWEB toolset. Another

Genetic programming

- A string of bits could represent a *program*
- If you want a program to do something, you might try to *evolve* one
- As a concrete example, suppose you want a program to help you choose stocks in the stock market
  - There is a huge amount of data, going back many years
  - What data has the most predictive value?
  - What's the best way to combine this data?
- A genetic program is possible in theory, but it might take millions of years to evolve into something useful
  - How can we improve this?