Genetic Algorithms

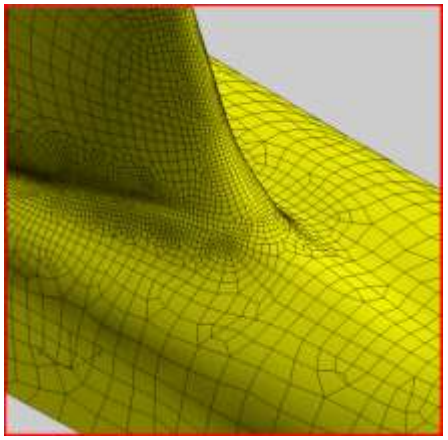
Credits:
Richard Frankel
Stanford University
And
Anas S. To’meh

Outline

- Motivations/applicable situations
- What are genetic algorithms?
- Example
- Pros and cons

Motivation

- Searching some search spaces with traditional search methods would be intractable. This is often true when states/candidate solutions have a large number of successors.
  - Example: Designing the surface of an aircraft.



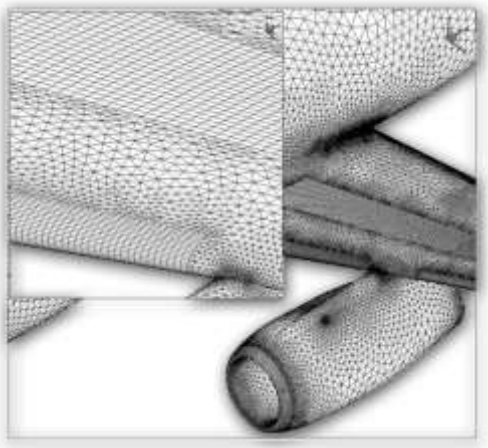


Image source: http://www.centaursoft.com/cms/index.php?section=25

Applicable situations

- Often used for optimization (scheduling, design, etc.) problems, though can be used for many other things as well, as we'll see a bit later.
  - Good problem for GA: Scheduling air traffic
  - Bad problems for GA: Finding large primes (why?)

Applicable situations

- Genetic algorithms work best when the "fitness landscape" is continuous (in some dimensions). This is also true of standard search, e.g. A*.
  - Intuitively, this just means that we can find a heuristic that gives a rough idea of how close a candidate is to being a solution.

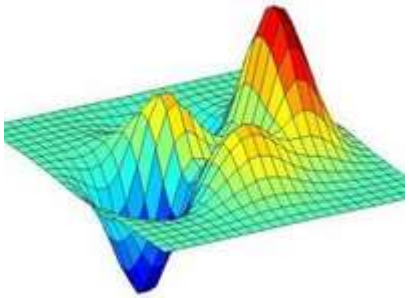


Image source: scholarpedia.org

So what *is* a genetic algorithm?

- Genetic algorithms are a randomized heuristic search strategy.
- Basic idea: Simulate natural selection, where the population is composed of *candidate solutions*.
- Focus is on evolving a population from which strong and diverse candidates can emerge via mutation and crossover (mating).

Basic algorithm

- Create an initial population, either random or “blank”.
- While the best candidate so far is not a solution:
  - Create new population using successor functions.
  - Evaluate the fitness of each candidate in the population.
- Return the best candidate found.

Simple example – alternating string

- Let's try to evolve a length 4 alternating string
- Initial population: C1=1000, C2=0011
- We roll the dice and end up creating C1' = cross (C1, C2) = 1011 and C2' = cross (C1, C1) = 1000.
- We mutate C1' and the fourth bit flips, giving 1010. We mutate C2' and get 1001.
- We run our solution test on each. If C1' is a solution, so we return it and are done.

Basic components

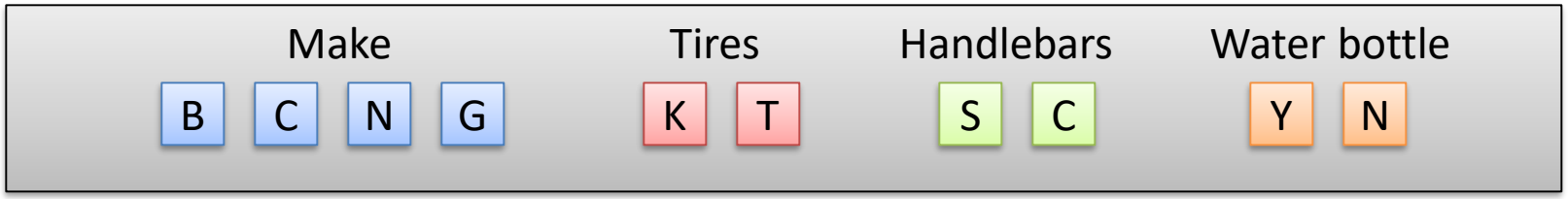
- Candidate representation
  - Important to choose this well. More work here means less work on the successor functions.
- Successor function(s)
  - Mutation, crossover
- Fitness function
- Solution test
- Some parameters
  - Population size
  - Generation limit

Candidate representation

- We want to encode candidates in a way that makes mutation and crossover easy.
- The typical candidate representation is a binary string. This string can be thought of as the genetic code of a candidate – thus the term “genetic algorithm”!
  - Other representations are possible, but they make crossover and mutation harder.

Candidate representation example

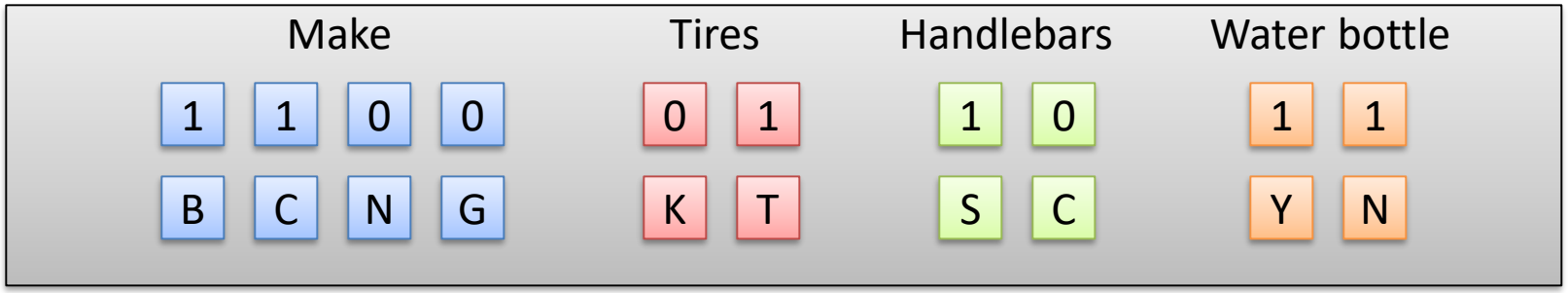
- Let's say we want to represent a rule for classifying bikes as mountain bikes or hybrid, based on these attributes*:
  - Make (Bridgestone, Cannondale, Nishiki, or Gary Fisher)
  - Tire type (knobby, treads)
  - Handlebar type (straight, curved)
  - Water bottle holder (*Boolean*)
- We can encode a rule as a binary string, where each bit represents whether a value is accepted.



*Bikes scheme used with permission from Mark Maloof.

Candidate representation example

- The candidate will be a bit string of length 10, because we have 10 possible attribute values.
- Let's say we want a rule that will match any bike that is made by Bridgestone or Cannondale, has treaded tires, and has straight handlebars. This rule could be represented as 1100011011:



Successor functions

- Mutation – Given a candidate, return a slightly different candidate.
- Crossover – Given two candidates, produce one that has elements of each.
- We don't always generate a successor for each candidate. Rather, we generate a successor *population* based on the candidates in the current population, weighted by fitness.

Successor functions

- If your candidate representation is just a binary string, then these are easy:
  - Mutate(c): Copy c as c'. For each bit b in c', flip b with probability p. Return c'.
  - Cross (c1, c2): Create a candidate c such that c[i] = c1[i] if i % 2 = 0, c[i] = c2[i] otherwise. Return c.
    - Alternatively, any other scheme such that c gets roughly equal information from c1 and c2.

Fitness function

- The fitness function is analogous to a heuristic that estimates how close a candidate is to being a solution.
- In general, the fitness function should be consistent for better performance.
- However, even if it is, there are no guarantees. This is a probabilistic algorithm!

Solution test

- Given a candidate, return whether the candidate is a solution.
- Often just answers the question “does the candidate satisfy some set of constraints?”
- Optional! Sometimes you just want to do the best you can in a given number of generations.

New population generation

- How do we come up with a new population?
  - Given a population P, generate P’ by performing crossover |P| times, each time selecting candidates with probability proportional to their fitness.
  - Get P’’ by mutating each candidate in P’.
  - Return P’’.

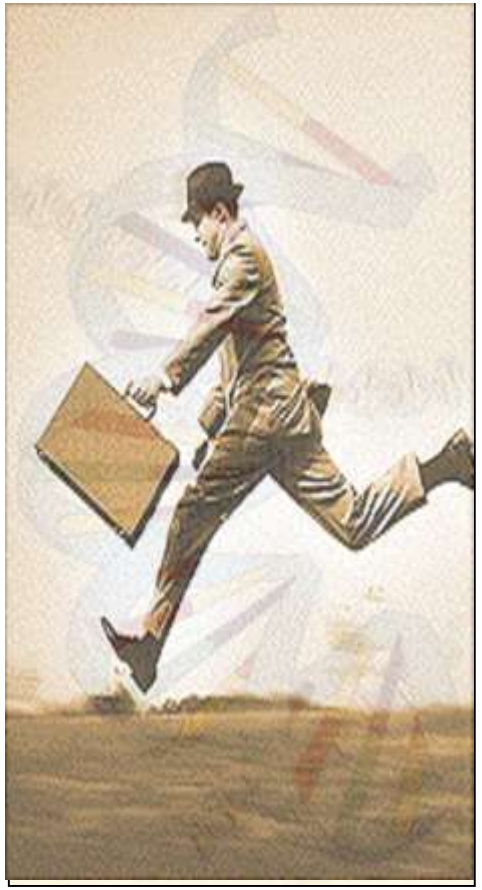
Basic algorithm (recap)

- Create an initial population, either random or “blank”.
- While the best candidate so far is not a solution:
  - Create new population using successor functions.
  - Evaluate the fitness of each candidate in the population.
- Return the best candidate found.

Knapsack Problem

Problem Description:

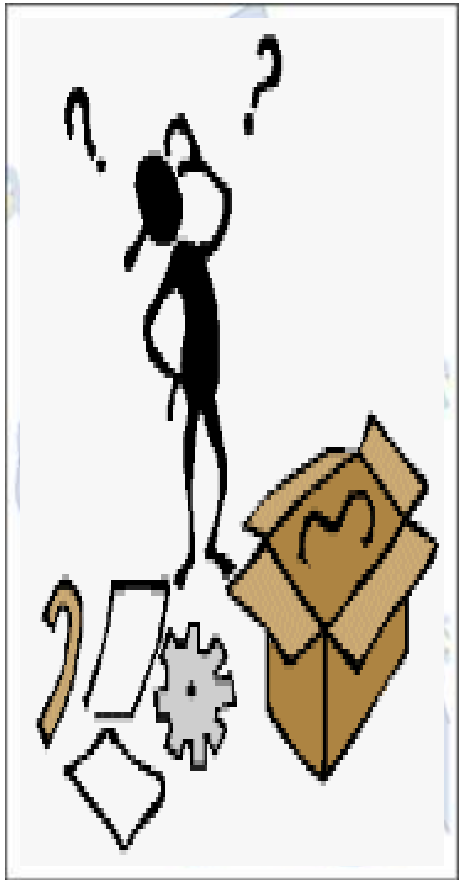
- You are going on a picnic.
- And have a number of items that you could take along.
- Each item has a weight and a benefit or value.
- You can take one of each item at most.
- There is a capacity limit on the weight you can carry.
- You should carry items with max. values.



Knapsack Problem

Example:

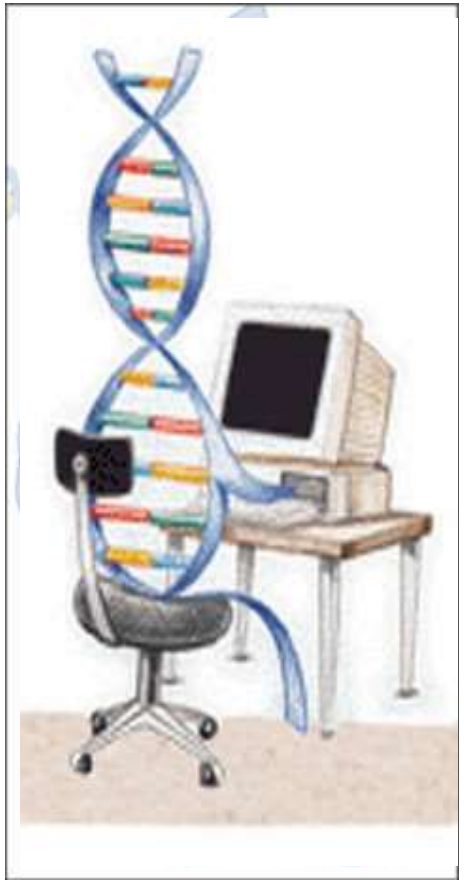
- **Item:** 1 2 3 4 5 6 7
- **Benefit:** 5 8 3 2 7 9 4
- **Weight:** 7 8 4 10 4 6 4
- **Knapsack holds a maximum of 22 pounds**
- **Fill it to get the maximum benefit**



Genetic Algorithm

Outline of the Basic Genetic Algoritm

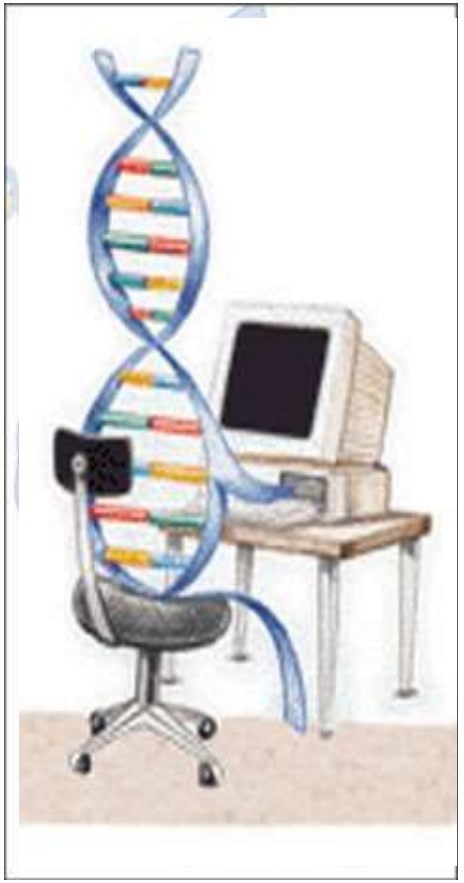
1. **[Start]**
   - ✓ Encoding: represent the individual.
   - ✓ Generate random population of *n* chromosomes (suitable solutions for the problem).
2. **[Fitness]** Evaluate the fitness of each chromosome.
3. **[New population]** repeating following steps until the new population is complete.
4. **[Selection]** Select the best two parents.
5. **[Crossover]** cross over the parents to form a new offspring (children).



Genetic Algorithm

Outline of the Basic Genetic Algoritm Cont.

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

Start

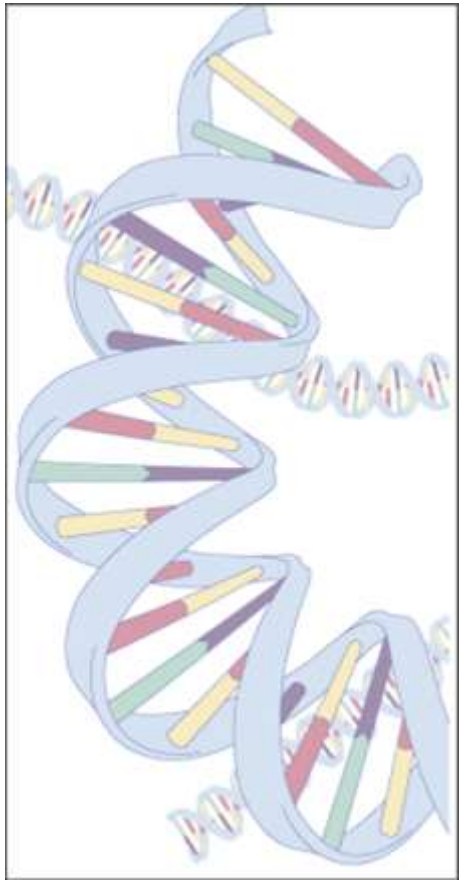
- Encoding: 0 = not exist, 1 = exist in the Knapsack

  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



Basic Steps Cont.

Fitness & Selection

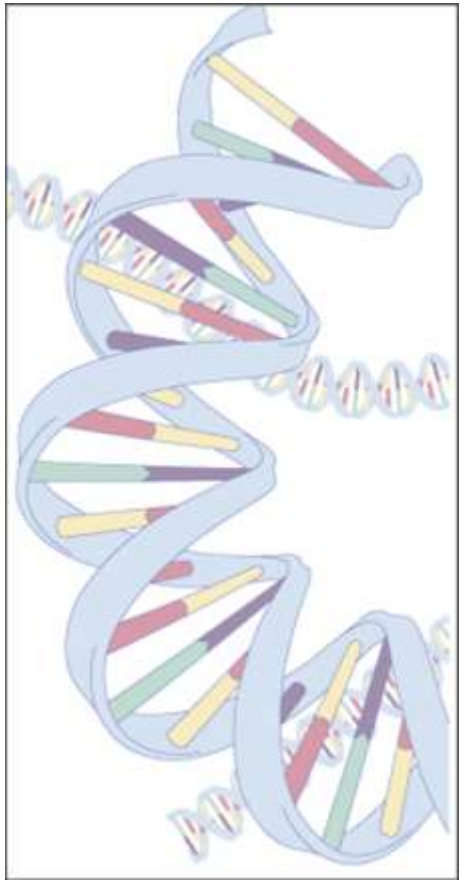
a) 0101010: Benefit= 19, Weight= 24 ✘

Item	1	2	3	4	5	6	7
Chro	0	1	0	1	0	1	0
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

b) 1100100: Benefit= 20, Weight= 19. ✓

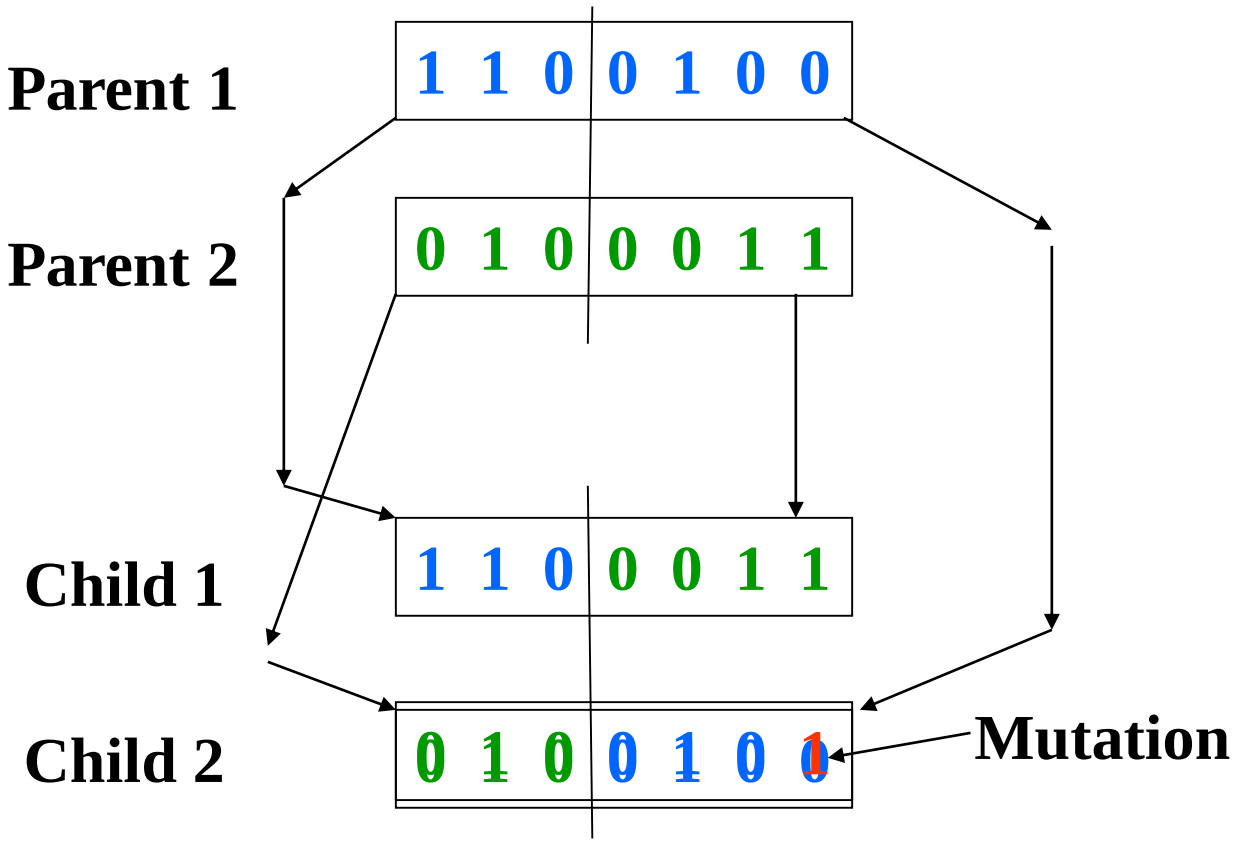
c) 0100011: Benefit= 21, Weight= 18. ✓

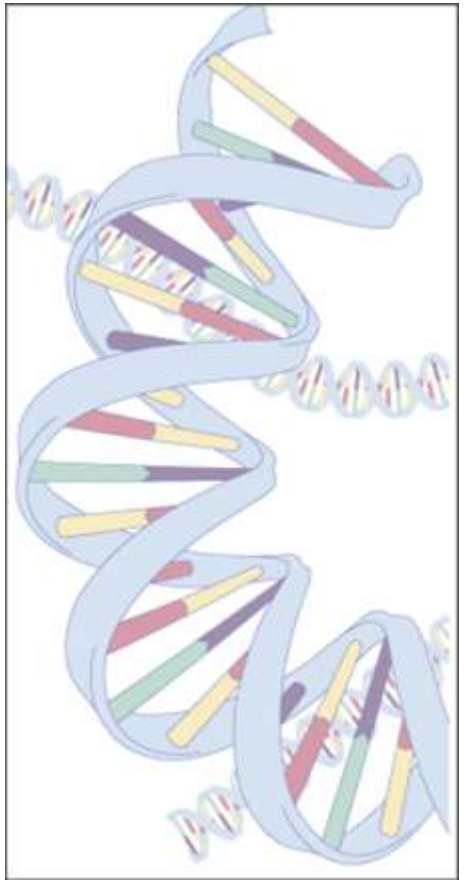
=> We select Chromosomes b & c.



Basic Steps Cont.

Crossover & Mutation

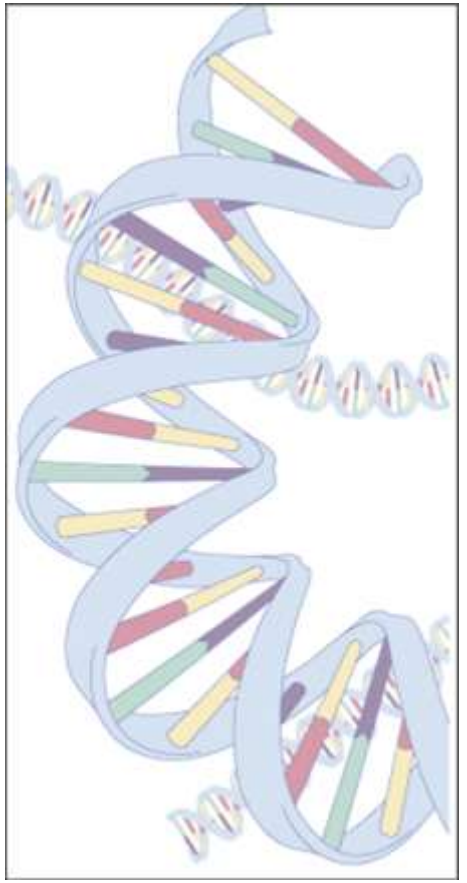




Basic Steps Cont.

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].**

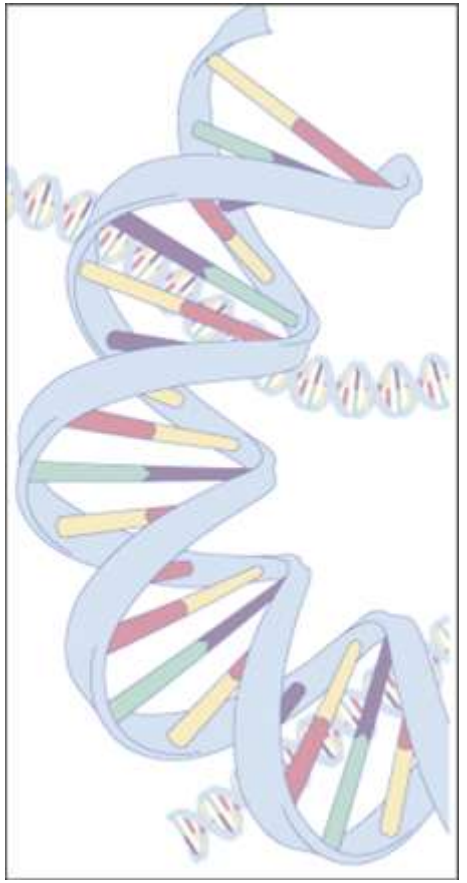


Genetic Algorithm

Conclusion

- GA is nondeterministic – two runs may end with different results

- There's no indication whether best individual is optimal



Pros and Cons

- Pros
  - Faster (and lower memory requirements) than searching a very large search space.
  - Easy, in that if your candidate representation and fitness function are correct, a solution can be found without any explicit analytical work.
- Cons
  - Randomized – not optimal or even complete.
  - Can get stuck on local maxima, though crossover can help mitigate this.
  - It can be hard to work out how best to represent a candidate as a bit string (or otherwise).