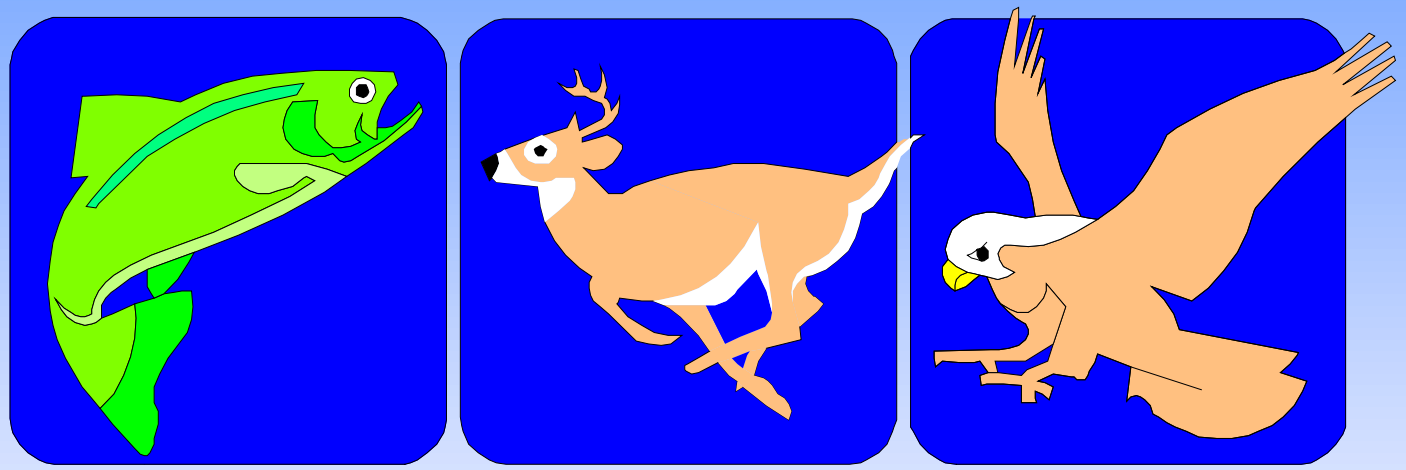
Learn how to morph faces with a Generative Adversarial Network!

https://www.youtube.com/watch?v=dCKbRCUyop8

Creating and Training a Generative Adversarial Networks (GAN) in Keras (7.2)

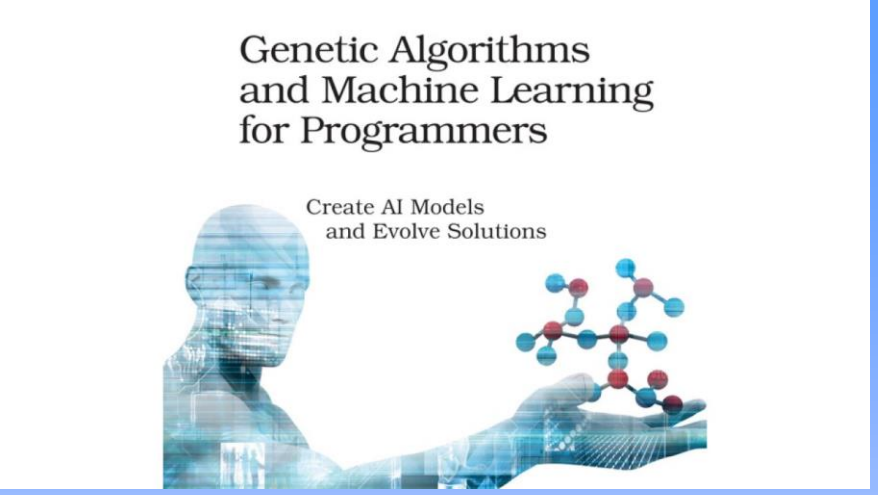
https://www.youtube.com/watch?v=T-MCludVNn4

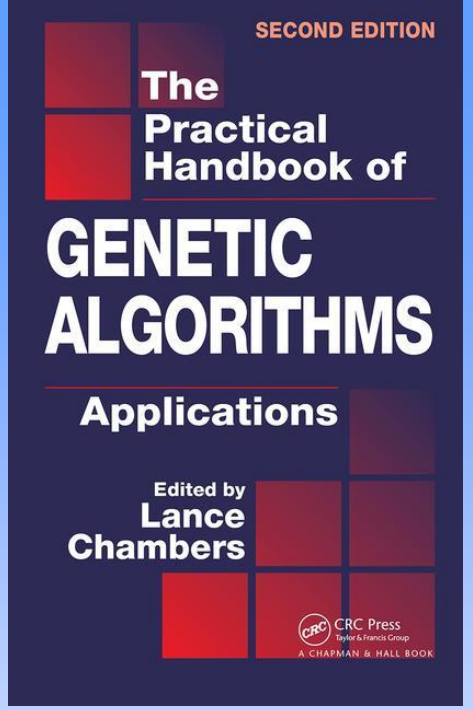
A Simple Example

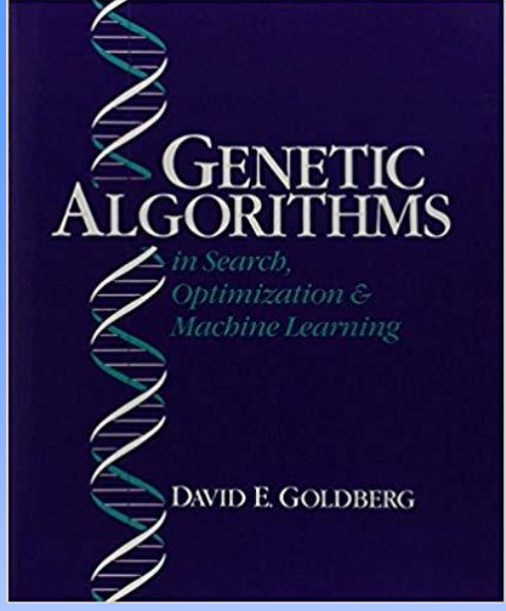


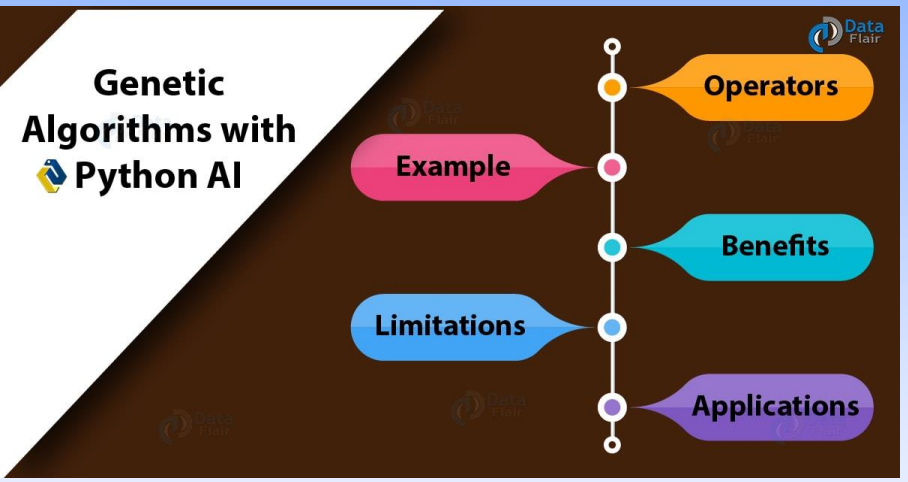
"The Gene is by far the most sophisticated program around."

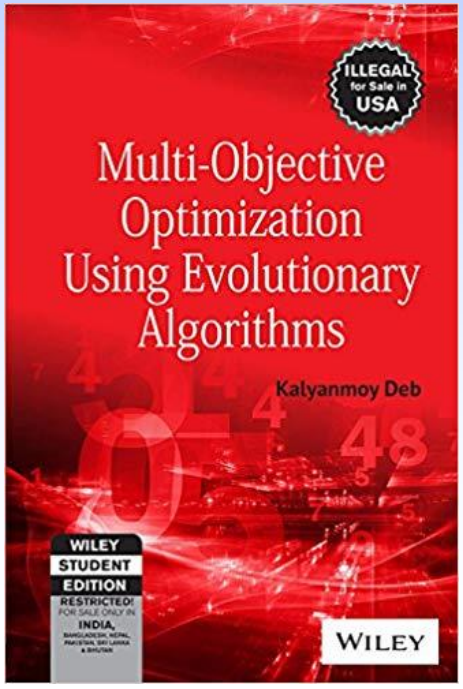
- Bill Gates, *Business Week*, June 27, 1994

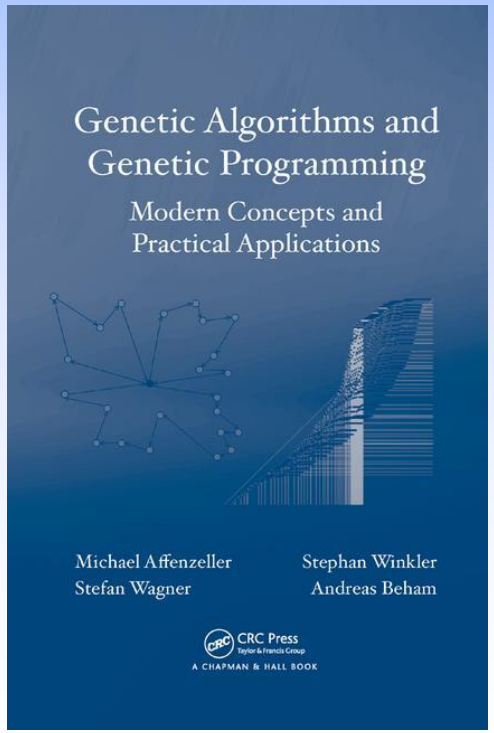

Genetic Algorithms
and Machine Learning
for Programmers
Create AI Models
and Evolve Solutions


SECOND EDITION
The
Practical
Handbook of
GENETIC
ALGORITHMS
Applications
Edited by
Lance
Chambers
CRC Press
A CHAPMAN & HALL BOOK

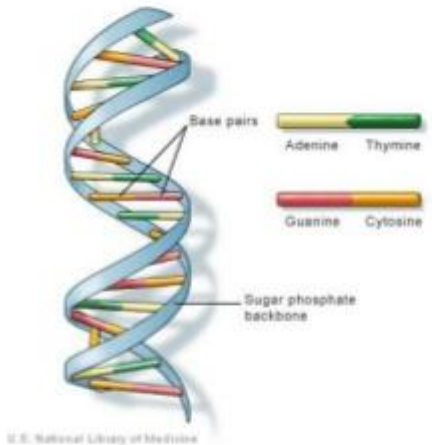

GENETIC
ALGORITHMS
in Search,
Optimization &
Machine Learning
DAVID E. GOLDBERG


Data
Flair
Genetic
Algorithms with
Python AI
Operators
Example
Benefits
Limitations
Applications


ILLEGAL
for Sale in
USA
Multi-Objective
Optimization
Using Evolutionary
Algorithms
Kalyanmoy Deb
WILEY
STUDENT
EDITION
RESTRICTED!
FOR SALE ONLY IN
INDIA,
WILEY


Genetic Algorithms and
Genetic Programming
Modern Concepts and
Practical Applications
Michael Affenzeller
Stefan Wagner
Stephan Winkler
Andreas Beham
CRC Press
A CHAPMAN & HALL BOOK

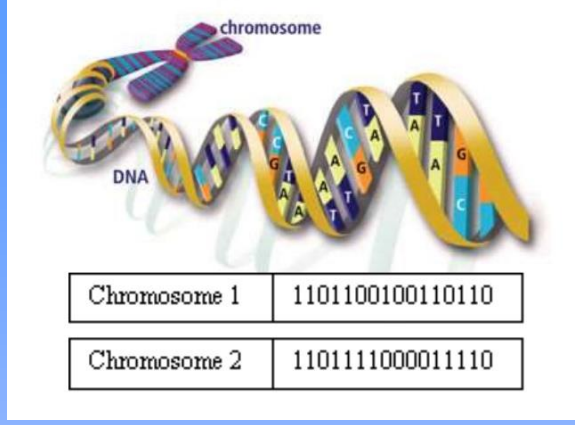
What is a Genetic Algorithm?



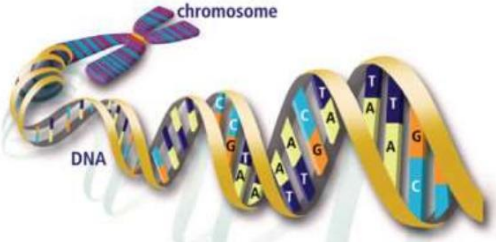
- A genetic algorithm is an adaptation procedure based on the mechanics of natural genetics and natural selection.
- Genetic Algorithm's have 2 essential components:
  - "Survival of the fittest"
  - Genetic Diversity
- Originally developed by John Holland (1975).
- The genetic algorithm (GA) is a search heuristic that mimics the process of natural evolution.
- Uses concepts of "Natural Selection" and "Genetic Inheritance" (Darwin 1859).

slide text from Douglas Reynolds

Genetic Algorithms - History



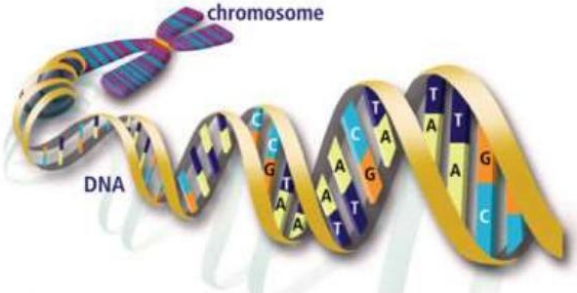
- Pioneered by John Holland in the 1970's
- Got popular in the late 1980's
- Based on ideas from Darwinian Evolution
- Can be used to solve a variety of problems that are not easy to solve using other techniques



Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Genetic Algorithms (GA) OVERVIEW

- A class of probabilistic optimization algorithms
- Inspired by the biological evolution process
- Uses concepts of “Natural Selection” and “Genetic Inheritance” (Darwin 1859)
- Originally developed by John Holland (1975)

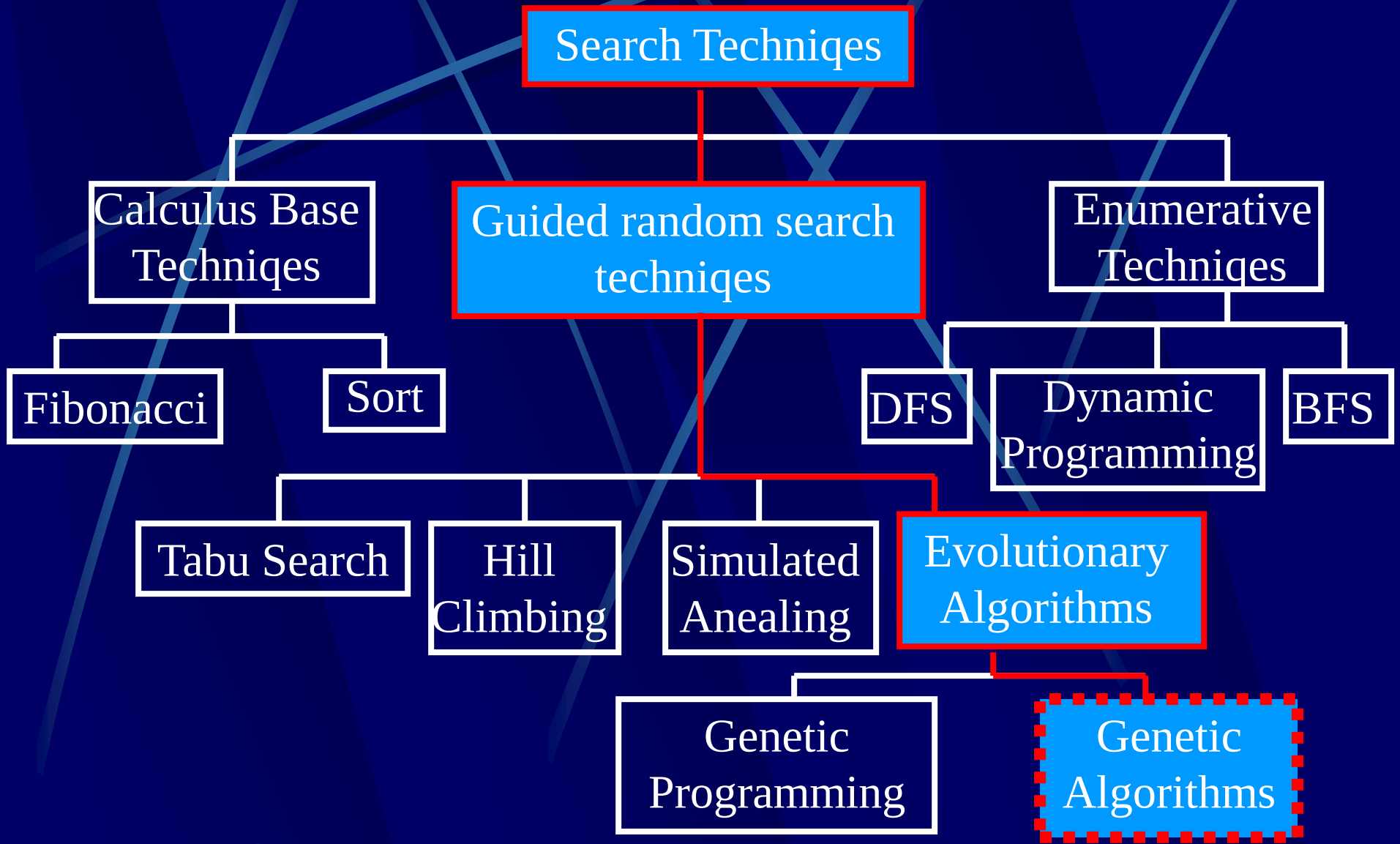


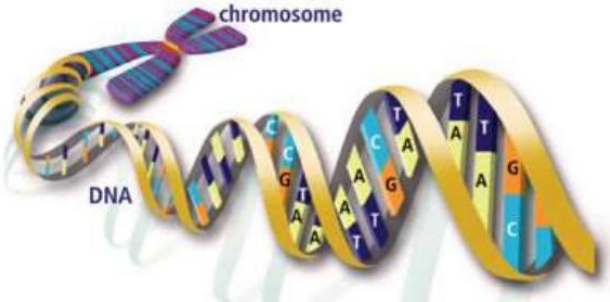
Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

GA overview (cont)

- Particularly well suited for hard problems where little is known about the underlying search space
- Widely-used in business, science and engineering

Classes of Search Techniques

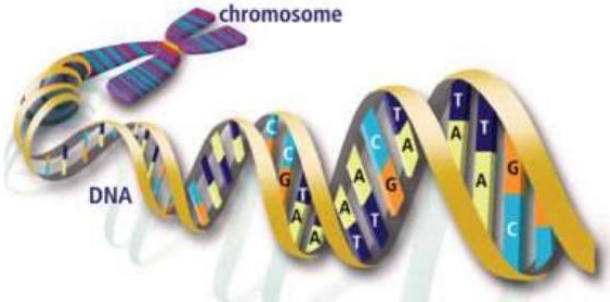




Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Evolution in the real world

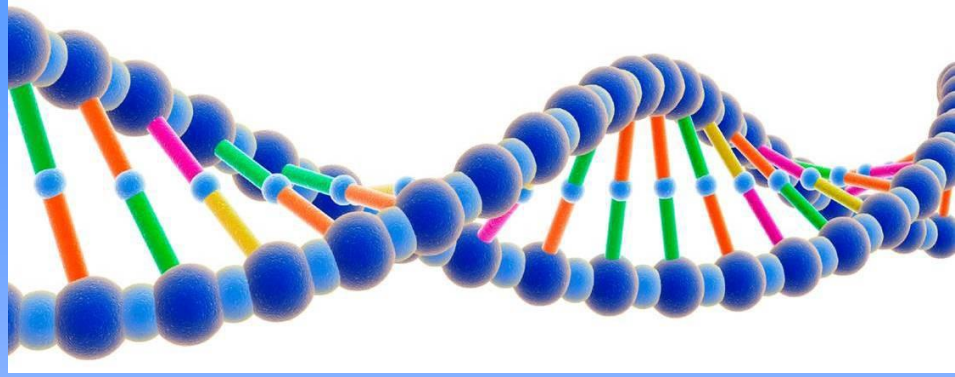
- Each cell of a living thing contains ***chromosomes*** - strings of *DNA*
- Each chromosome contains a set of ***genes*** - blocks of DNA
- Each gene determines some aspect of the organism (like eye colour)
- A collection of genes is sometimes called a ***genotype***



Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

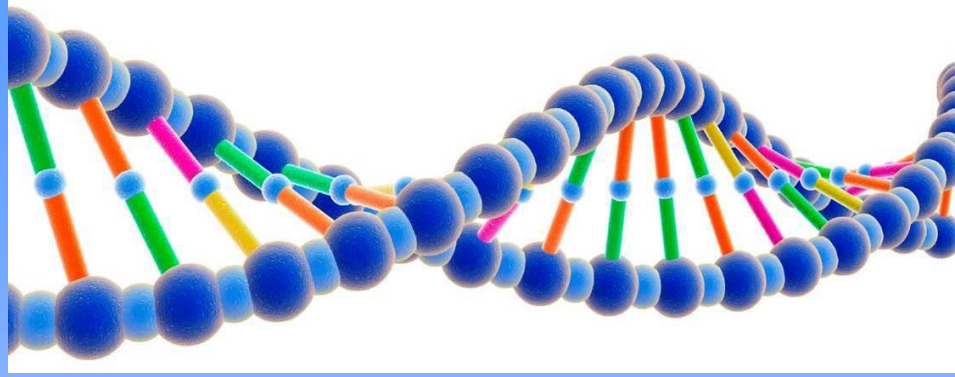
Evolution in the real world

- A collection of aspects (like eye colour) is sometimes called a ***phenotype***
- Reproduction involves recombination of genes from parents and then small amounts of ***mutation*** (errors) in copying
- The ***fitness*** of an organism is how much it can reproduce before it dies
- Evolution based on "survival of the fittest"



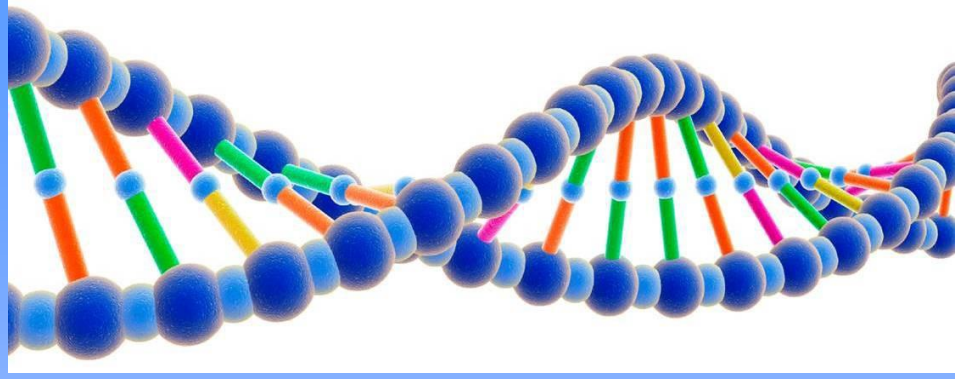
Start with a Dream…

- Suppose you have a problem
- You don’t know how to solve it
- What can you do?



Start with a Dream…

- Suppose you have a problem
- You don’t know how to solve it
- What can you do?
- Can you use a computer to somehow find a solution for you?
- This would be nice! Can it be done?



A dumb solution

A “blind generate and test” algorithm:

Repeat

- Generate a random possible solution
- Test the solution and see how good it is

Until solution is good enough

Can we use this dumb idea?

- Sometimes - yes:
  - if there are only a few possible solutions
  - and you have enough time
  - then such a method *could* be used

Can we use this dumb idea?

- For most problems - no:
  - many possible solutions
  - with no time to try them all
  - so this method *can not* be used

A “less-dumb” idea (GA)

Generate a *set* of random solutions

Repeat

- Test each solution in the set (rank them)
- Remove some bad solutions from set
- Duplicate some good solutions
  - make small changes to some of them

Until best solution is good enough

How do you encode a solution?

- Obviously this depends on the problem!
- GA's *often* encode solutions as fixed length "bitstrings" (e.g. 101110, 111111, 000101)
- Each bit represents some aspect of the proposed solution to the problem
- For GA's to work, we need to be able to "test" any string and get a "score" indicating how "good" that solution is

Silly Example - Drilling for Oil

- Imagine you had to drill for oil somewhere along a single 1km desert road
- Problem: choose the best place on the road that produces the most oil per day
- We could represent each solution as a position on the road
- Say, a whole number between [0..1000]

Where to drill for oil?

Solution1 = 300

Solution2 = 900

Road

0

500

1000

Digging for Oil

- The set of all possible solutions [0..1000] is called the *search space* or *state space*
- In this case it's just one number but it could be many numbers or symbols
- Often GA's code numbers in binary producing a bitstring representing a solution
- In our example we choose 10 bits which is enough to represent 0..1000

Convert to binary string

	512	256	128	64	32	16	8	4	2	1
900	1	1	1	0	0	0	0	1	0	0
300	0	1	0	0	1	0	1	1	0	0
1023	1	1	1	1	1	1	1	1	1	1

In GA's these encoded strings are sometimes called *"genotypes"* or *"chromosomes" and the individual bits are sometimes called "genes"*

Drilling for Oil

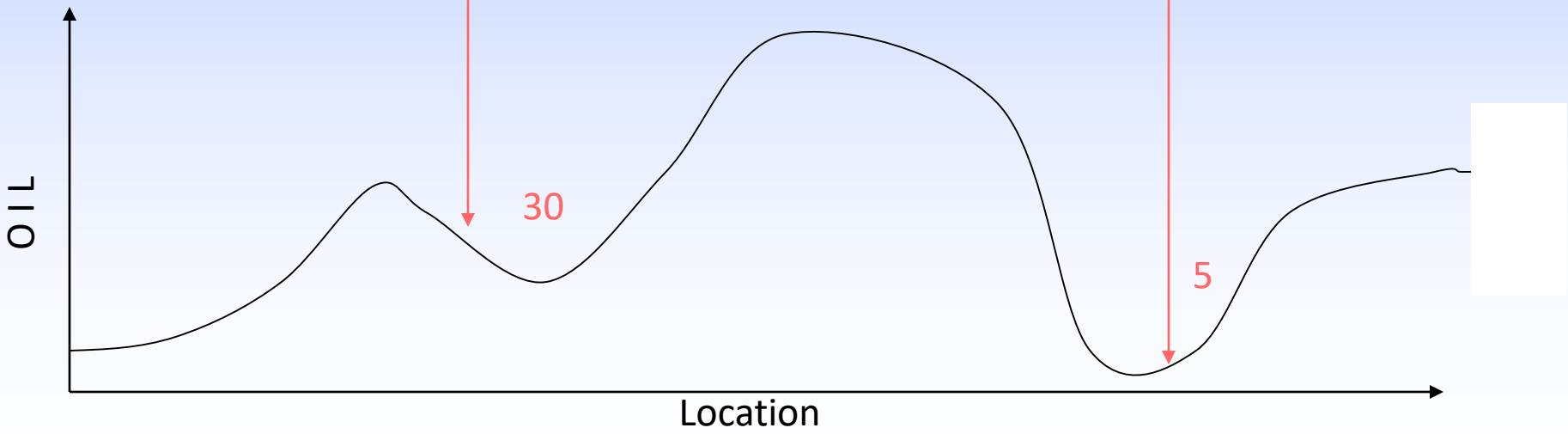
Solution1 = 300
(0100101100)

Solution2 = 900 (1110000100)

Road

0

1000



Summary

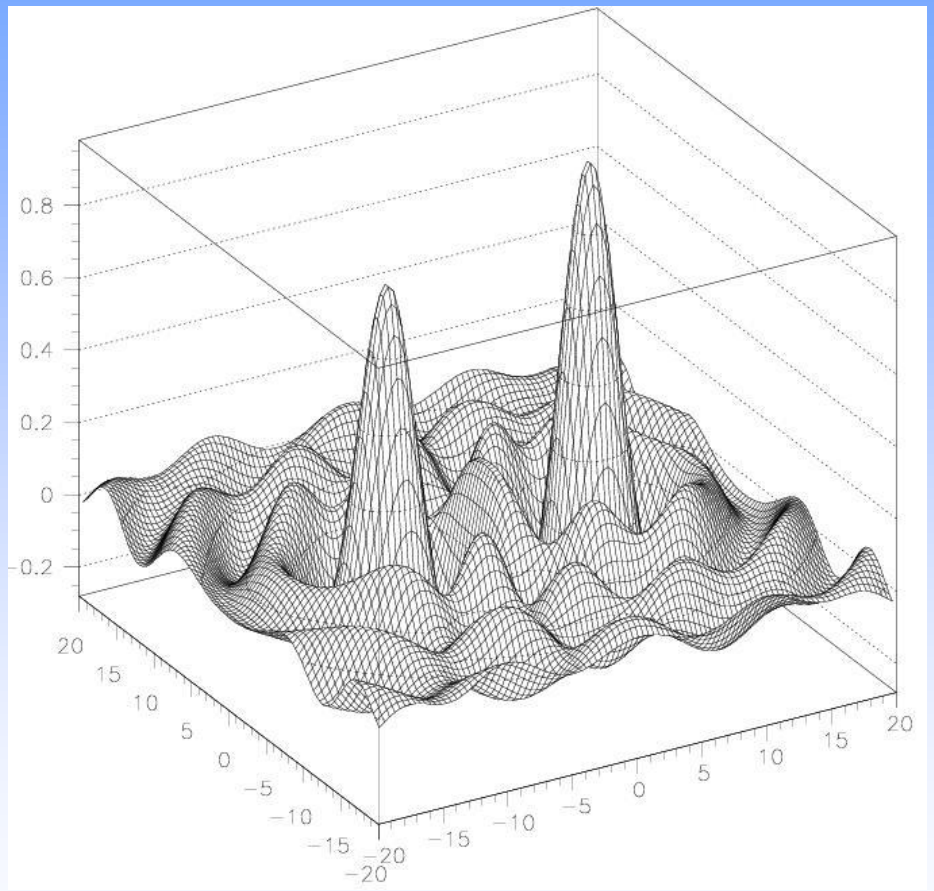
We have seen how to:

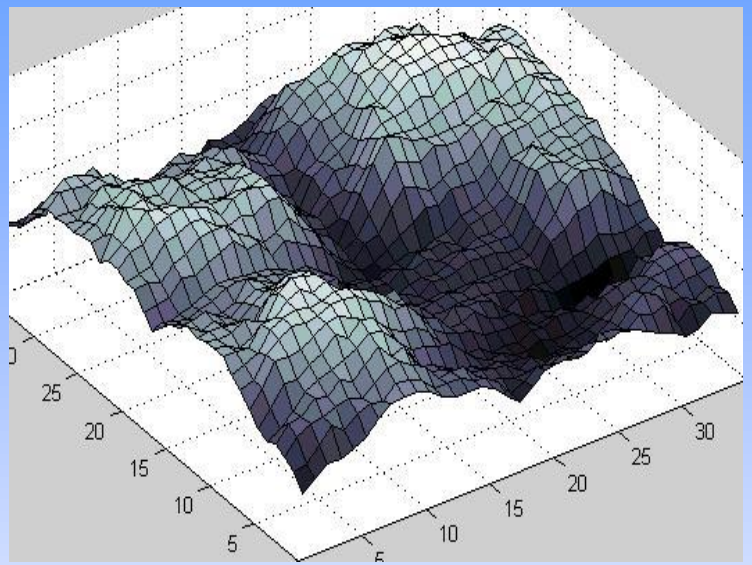
- represent possible solutions as a number
- encoded a number into a binary string
- generate a score for each number given a *function* of "how good" each solution is - this is often called a *fitness function*
- Our silly oil example is really optimisation over a function f(x) where we adapt the parameter x

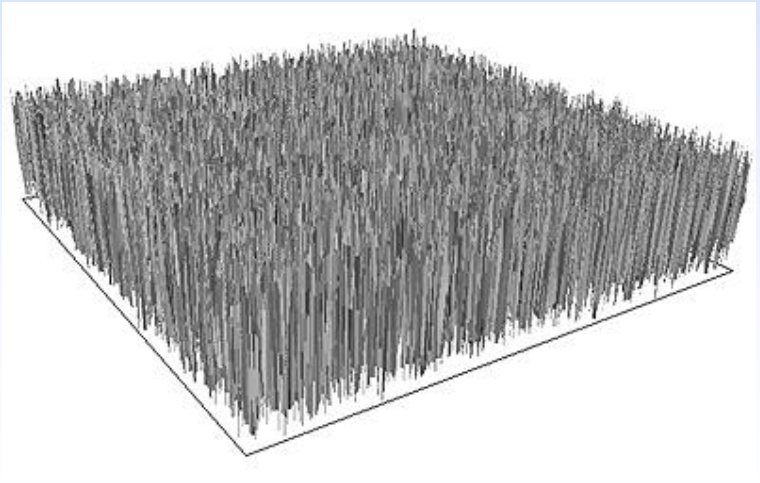
Search Space

- For a simple function f(x) the search space is one dimensional.
- But by encoding several values into the chromosome many dimensions can be searched e.g. two dimensions f(x,y)
- Search space an be visualised as a surface or *fitness landscape* in which fitness dictates height
- Each possible genotype is a point in the space
- A GA tries to move the points to better places (higher fitness) in the the space

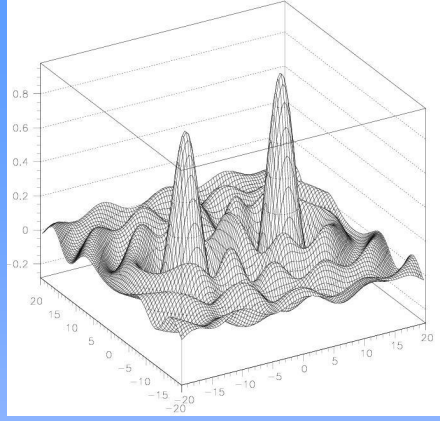
Fitness landscapes



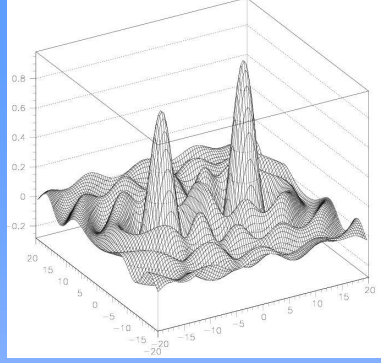




Search Space



- Obviously, the nature of the search space dictates how a GA will perform
- A completely random space would be bad for a GA
- Also GA's can get stuck in local maxima if search spaces contain lots of these
- Generally, spaces in which small improvements get closer to the global optimum are good



Back to the (GA) Algorithm

Generate a *set* of random solutions

Repeat

Test each solution in the set (rank them)

Remove some bad solutions from set

Duplicate some good solutions

make small changes to some of them

Until best solution is good enough

Adding Sex - Crossover

- Although it may work for simple search spaces our algorithm is still very simple
- It relies on random mutation to find a good solution
- It has been found that by introducing "sex" into the algorithm better results are obtained
- This is done by selecting two parents during reproduction and combining their genes to produce offspring

Adding Sex - Crossover

- Two high scoring “parent” bit strings (*chromosomes)* are selected and with some probability (crossover rate) combined
- Producing two new *offspring* (bit strings)
- Each offspring may then be changed randomly (*mutation*)

Selecting Parents

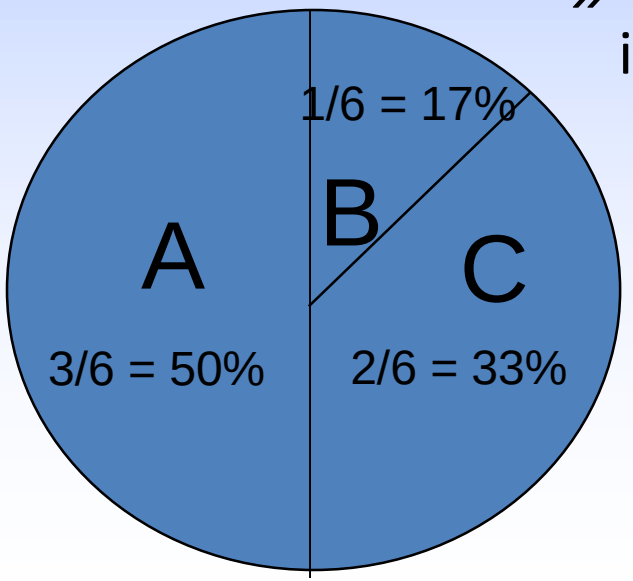
- Many schemes are possible so long as better scoring chromosomes more likely selected
- Score is often termed the *fitness*
- "Roulette Wheel" selection can be used:
  - Add up the fitness's of all chromosomes
  - Generate a random number R in that range
  - Select the first chromosome in the population that - when all previous fitness's are added - gives you at least the value R

Example population

No.	Chromosome	Fitness
1	1010011010	1
2	1111100001	2
3	1011001100	3
4	1010000000	1
5	0000010000	3
6	1001011111	5
7	0101010101	1
8	1011100111	2

SGA operators: Selection

- Main idea: better individuals get higher chance
  - Chances proportional to fitness
  - Implementation: roulette wheel technique
    - » Assign to each individual a part of the roulette wheel
    - » Spin the wheel n times to select n individuals

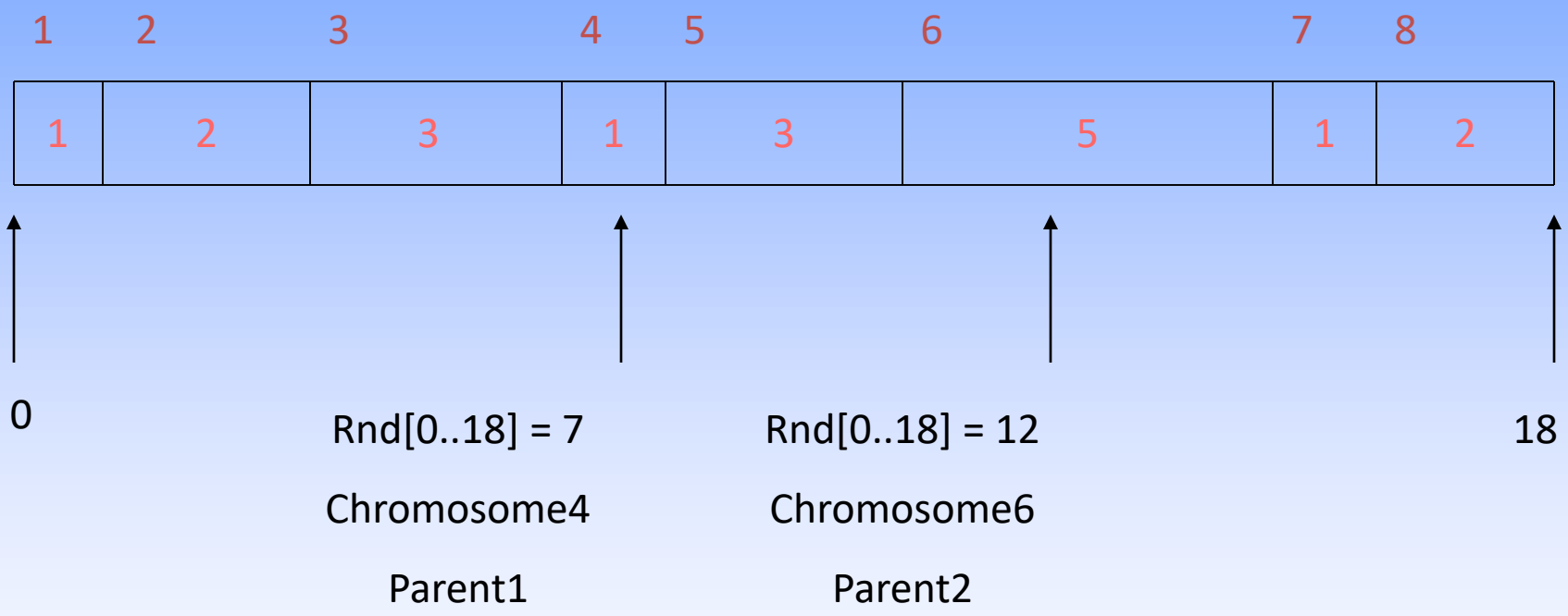


fitness(A) = 3

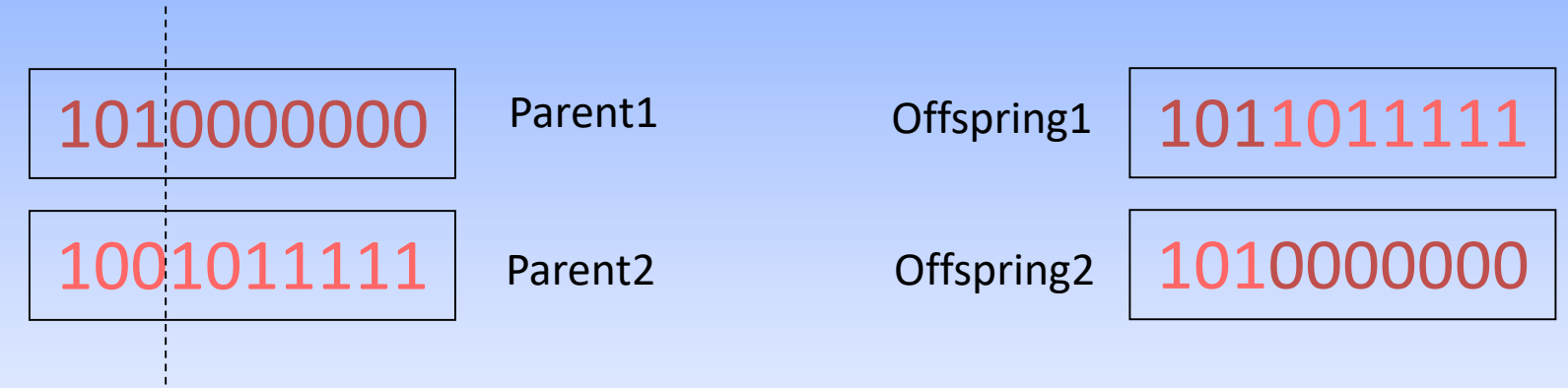
fitness(B) = 1

fitness(C) = 2

Roulette Wheel Selection

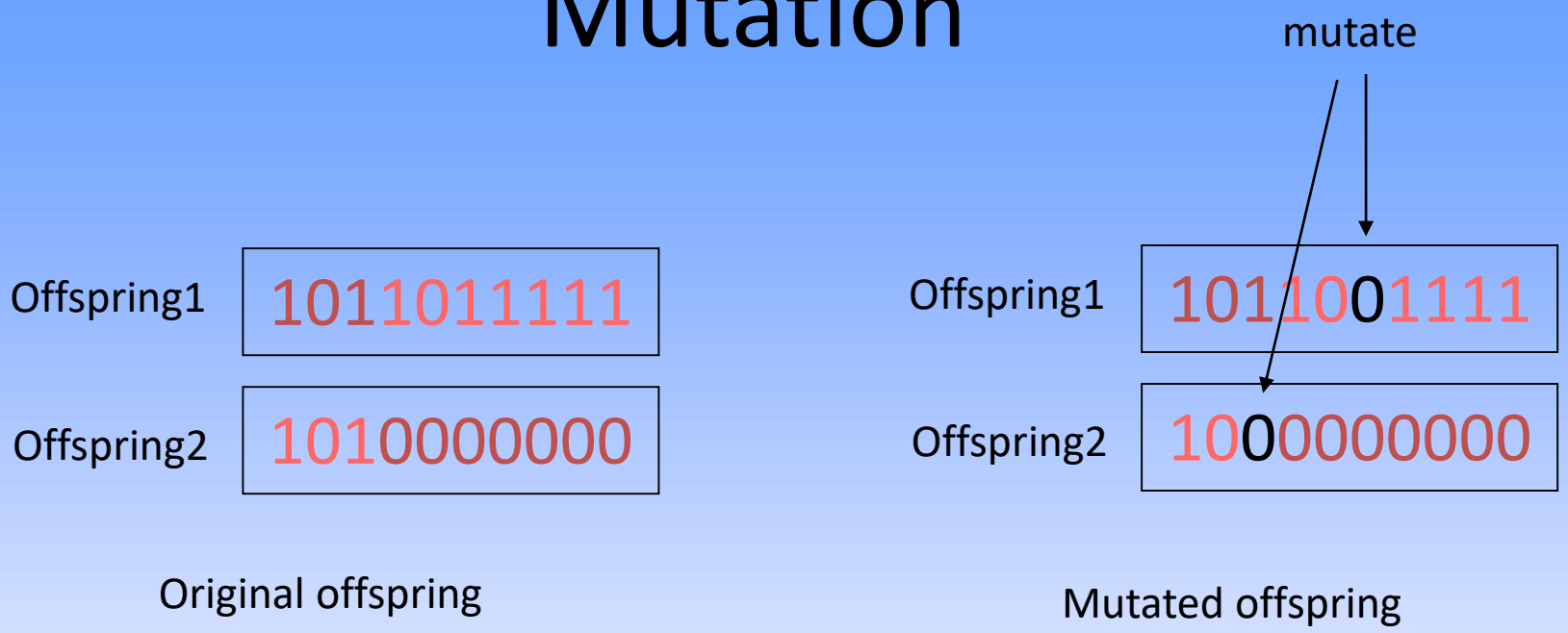


Crossover - Recombination



With some high probability (*crossover rate*) apply crossover to the parents. (*typical values are 0.8 to 0.95*)

Mutation



With some small probability (the *mutation rate*) flip each bit in the offspring (*typical values between 0.1 and 0.001*)

Back to the (GA) Algorithm

Generate a *population* of random chromosomes

Repeat (each generation)

- Calculate fitness of each chromosome
- **Repeat**
  - Use roulette selection to select pairs of parents
  - Generate offspring with crossover and mutation
- **Until** a new population has been produced

Until best solution is good enough

An example after Goldberg ‘89 (1)

- Simple problem: max x^2 over {0,1,…,31}
- GA approach:
  - Representation: binary code, e.g. 01101 $\leftrightarrow$ 13
  - Population size: 4
  - 1-point xover, bitwise mutation
  - Roulette wheel selection
  - Random initialisation
- We show one generational cycle done by hand

x^2 example: selection

String no.	Initial population	x Value	Fitness $f(x) = x^2$	$Prob_i$	Expected count	Actual count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4.00	4
Average			293	0.25	1.00	1
Max			576	0.49	1.97	2

X^2 example: crossover

String no.	Mating pool	Crossover point	Offspring after xover	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 | 1	4	0 1 1 0 0	12	144
2	1 1 0 0 | 0	4	1 1 0 0 1	25	625
2	1 1 | 0 0 0	2	1 1 0 1 1	27	729
4	1 0 | 0 1 1	2	1 0 0 0 0	16	256
Sum					1754
Average					439
Max					729

X^2 example: mutation

String no.	Offspring after xover	Offspring after mutation	x Value	Fitness $f(x) = x^2$
1	0 1 1 0 0	1 1 1 0 0	26	676
2	1 1 0 0 1	1 1 0 0 1	25	625
2	1 1 0 1 1	1 1 0 1 1	27	729
4	1 0 0 0 0	1 0 1 0 0	18	324
Sum				2354
Average				588.5
Max				729

The simple GA

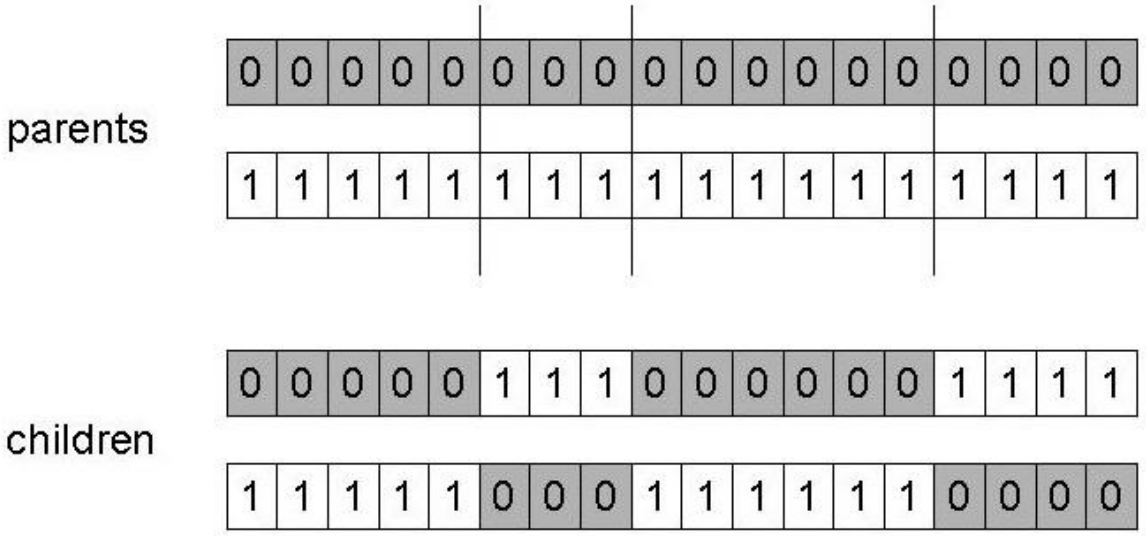
- Has been subject of many (early) studies
  - still often used as benchmark for novel GAs
- Shows many shortcomings, e.g.
  - Representation is too restrictive
  - Mutation & crossovers only applicable for bit-string & integer representations
  - Selection mechanism sensitive for converging populations with close fitness values
  - Generational population model (step 5 in SGA repr. cycle) can be improved with explicit survivor selection

Alternative Crossover Operators

- Performance with 1 Point Crossover depends on the order that variables occur in the representation
  - more likely to keep together genes that are near each other
  - Can never keep together genes from opposite ends of string
  - This is known as *Positional Bias*
  - Can be exploited if we know about the structure of our problem, but this is not usually the case

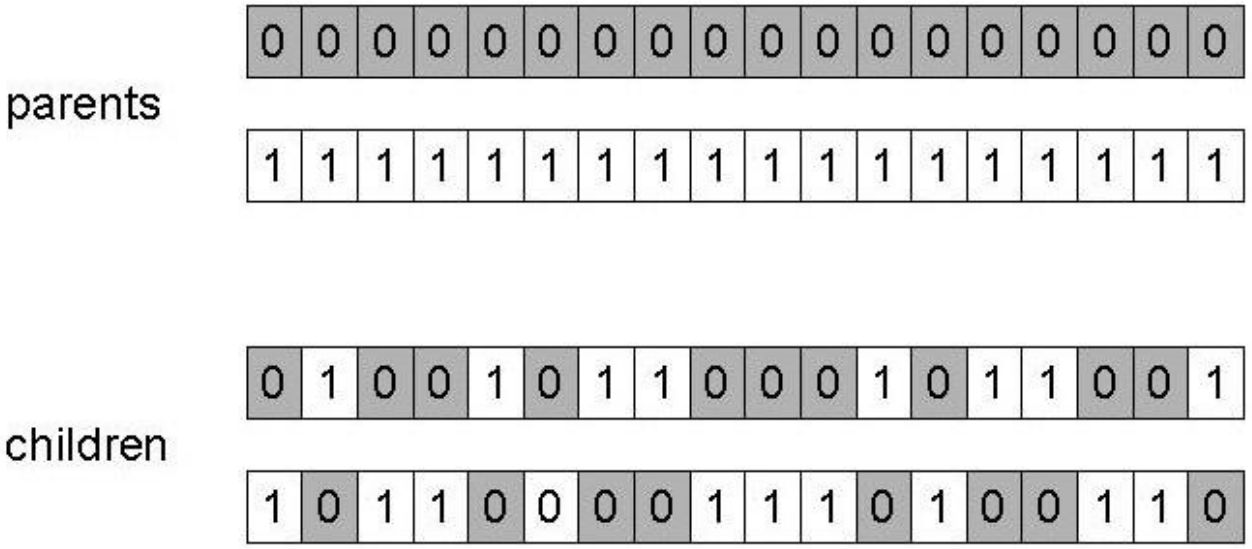
n-point crossover

- Choose n random crossover points
- Split along those points
- Glue parts, alternating between parents
- Generalisation of 1 point (still some positional bias)



Uniform crossover

- Assign 'heads' to one parent, 'tails' to the other
- Flip a coin for each gene of the first child
- Make an inverse copy of the gene for the second child
- Inheritance is independent of position



Crossover OR mutation?

- Decade long debate: which one is better / necessary / main-background

- Answer (at least, rather wide agreement):
  - it depends on the problem, but
  - in general, it is good to have both
  - both have another role
  - mutation-only-EA is possible, xover-only-EA would not work

Crossover OR mutation? (cont'd)

Exploration: Discovering promising areas in the search space, i.e. gaining information on the problem

Exploitation: Optimising within a promising area, i.e. using information

There is co-operation AND competition between them

- Crossover is explorative, it makes a *big* jump to an area somewhere "in between" two (parent) areas
- Mutation is exploitative, it creates random *small* diversions, thereby staying near (in the area of) the parent

Crossover OR mutation? (cont'd)

- Only crossover can combine information from two parents
- Only mutation can introduce new information (alleles)
- Crossover does not change the allele frequencies of the population (thought experiment: 50% 0's on first bit in the population, ?% after performing *n* crossovers)
- To hit the optimum you often need a 'lucky' mutation

Many Variants of GA

- Different kinds of selection (not roulette)
  - Tournament
  - Elitism, etc.
- Different recombination
  - Multi-point crossover
  - 3 way crossover etc.
- Different kinds of encoding other than bitstring
  - Integer values
  - Ordered set of symbols
- Different kinds of mutation

Many parameters to set

- Any GA implementation needs to decide on a number of parameters:
  - Population size (N),
  - Mutation rate (m),
  - Crossover rate (c)
- Often these have to be “tuned” based on results obtained - no general theory to deduce good values
- Typical values might be: N = 50, m = 0.05, c = 0.9

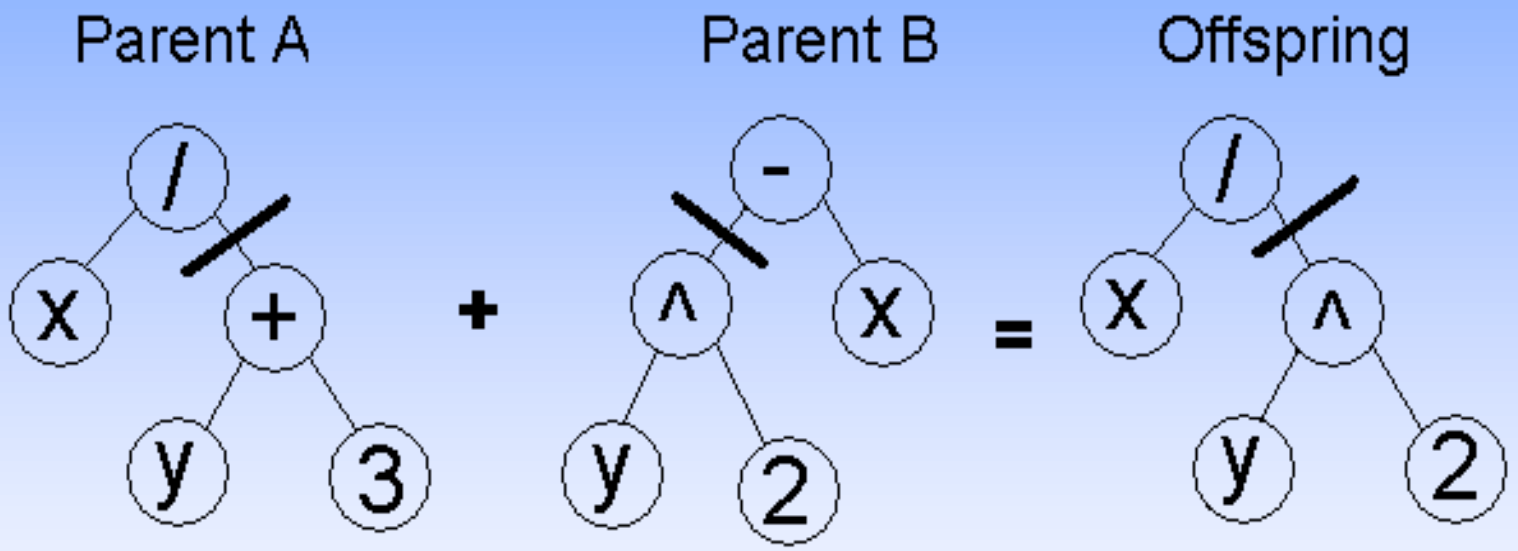
Why does crossover work?

- A lot of theory about this and some controversy
- Holland introduced “Schema” theory
- The idea is that crossover preserves “good bits” from different parents, combining them to produce better solutions
- A good encoding scheme would therefore try to preserve “good bits” during crossover and mutation

Genetic Programming

- When the chromosome encodes an entire program or function itself this is called genetic programming (GP)
- In order to make this work encoding is often done in the form of a tree representation
- Crossover entials swaping subtrees between parents

Genetic Programming



It is possible to evolve whole programs like this but only small ones. Large programs with complex functions present big problems

Implicit fitness functions

- Most GA's use explicit and static fitness function (as in our "oil" example)
- Some GA's (such as in Artificial Life or Evolutionary Robotics) use dynamic and implicit fitness functions - like *"how many obstacles did I avoid"*
- In these latter examples other chromosomes (robots) effect the fitness function

Problem

- In the Travelling Salesman Problem (TSP) a salesman has to find the shortest distance journey that visits a set of cities
- Assume we know the distance between each city
- This is known to be a hard problem to solve because the number of possible routes is N! where N = the number of cities
- There is no simple algorithm that gives the best answer quickly

Problem

- Design a chromosome encoding, a mutation operation and a crossover function for the Travelling Salesman Problem (TSP)
- Assume number of cities N = 10
- After all operations the produced chromosomes should always represent valid possible journeys (visit each city once only)
- There is no single answer to this, many different schemes have been used previously

Cycle crossover

Basic idea:

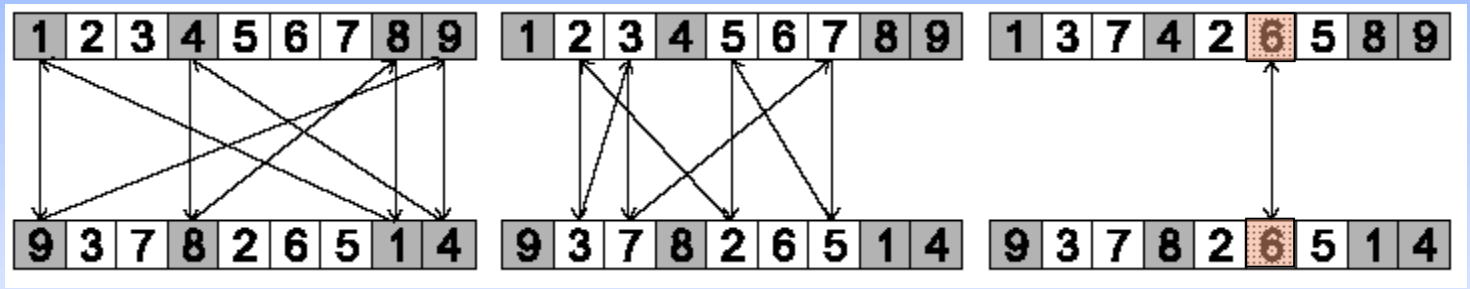
Each allele comes from one parent *together with its position*.

Informal procedure:

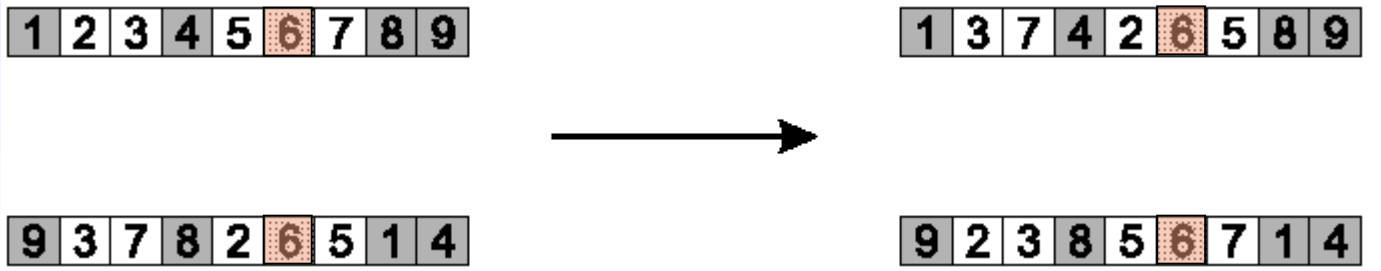
1. Make a cycle of alleles from P1 in the following way.
   (a) Start with the first allele of P1.
   (b) Look at the allele at the *same position* in P2.
   (c) Go to the position with the *same allele* in P1.
   (d) Add this allele to the cycle.
   (e) Repeat step b through d until you arrive at the first allele of P1.
2. Put the alleles of the cycle in the first child on the positions they have in the first parent.
3. Take next cycle from second parent

Cycle crossover example

- Step 1: identify cycles



- Step 2: copy alternate cycles into offspring



Edge Recombination

- Works by constructing a table listing which edges are present in the two parents, if an edge is common to both, mark with a +
- e.g. [1 2 3 4 5 6 7 8 9] and [9 3 7 8 2 6 5 1 4]

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Edge Recombination 2

Informal procedure once edge table is constructed

1. Pick an initial element at random and put it in the offspring
2. Set the variable current element = entry
3. Remove all references to current element from the table
4. Examine list for current element:
   - If there is a common edge, pick that to be next element
   - Otherwise pick the entry in the list which itself has the shortest list
   - Ties are split at random
5. In the case of reaching an empty list:
   - Examine the other end of the offspring is for extension
   - Otherwise a new element is chosen at random

Edge Recombination example

Element	Edges	Element	Edges
1	2,5,4,9	6	2,5+,7
2	1,3,6,8	7	3,6,8+
3	2,4,7,9	8	2,7+, 9
4	1,3,5,9	9	1,3,4,8
5	1,4,6+		

Choices	Element selected	Reason	Partial result
All	1	Random	[*1*]
2,5,4,9	5	Shortest list	[*1 5*]
4,6	6	Common edge	[*1 5 6*]
2,7	2	Random choice (both have two items in list)	[*1 5 6 2*]
3,8	8	Shortest list	[*1 5 6 2 8*]
7,9	7	Common edge	[*1 5 6 2 8 7*]
3	3	Only item in list	[*1 5 6 2 8 7 3*]
4,9	9	Random choice	[*1 5 6 2 8 7 3 9*]
4	4	Last element	[*1 5 6 2 8 7 3 9 4*]

Multiparent recombination

- Recall that we are not constricted by the practicalities of nature
- Noting that mutation uses 1 parent, and "traditional" crossover 2, the extension to $a>2$ is natural to examine
- Been around since 1960s, still rare but studies indicate useful
- Three main types:
  - Based on allele frequencies, e.g., p-sexual voting generalising uniform crossover
  - Based on segmentation and recombination of the parents, e.g., diagonal crossover generalising n-point crossover
  - Based on numerical operations on real-valued alleles, e.g., center of mass crossover, generalising arithmetic recombination operators

Population Models

- SGA uses a Generational model:
  - each individual survives for exactly one generation
  - the entire set of parents is replaced by the offspring
- At the other end of the scale are Steady-State models:
  - one offspring is generated per generation,
  - one member of population replaced,
- Generation Gap
  - the proportion of the population replaced
  - 1.0 for GGA, 1/pop_size for SSGA

Fitness Based Competition

- Selection can occur in two places:
  - Selection from current generation to take part in mating (parent selection)
  - Selection from parents + offspring to go into next generation (survivor selection)
- Selection operators work on whole individual
  - i.e. they are representation-independent
- Distinction between selection
  - operators: define selection probabilities
  - algorithms: define how probabilities are implemented

Implementation example: SGA

- Expected number of copies of an individual *i*

$$E(n_i) = \mu \cdot f(i) / \langle f \rangle$$

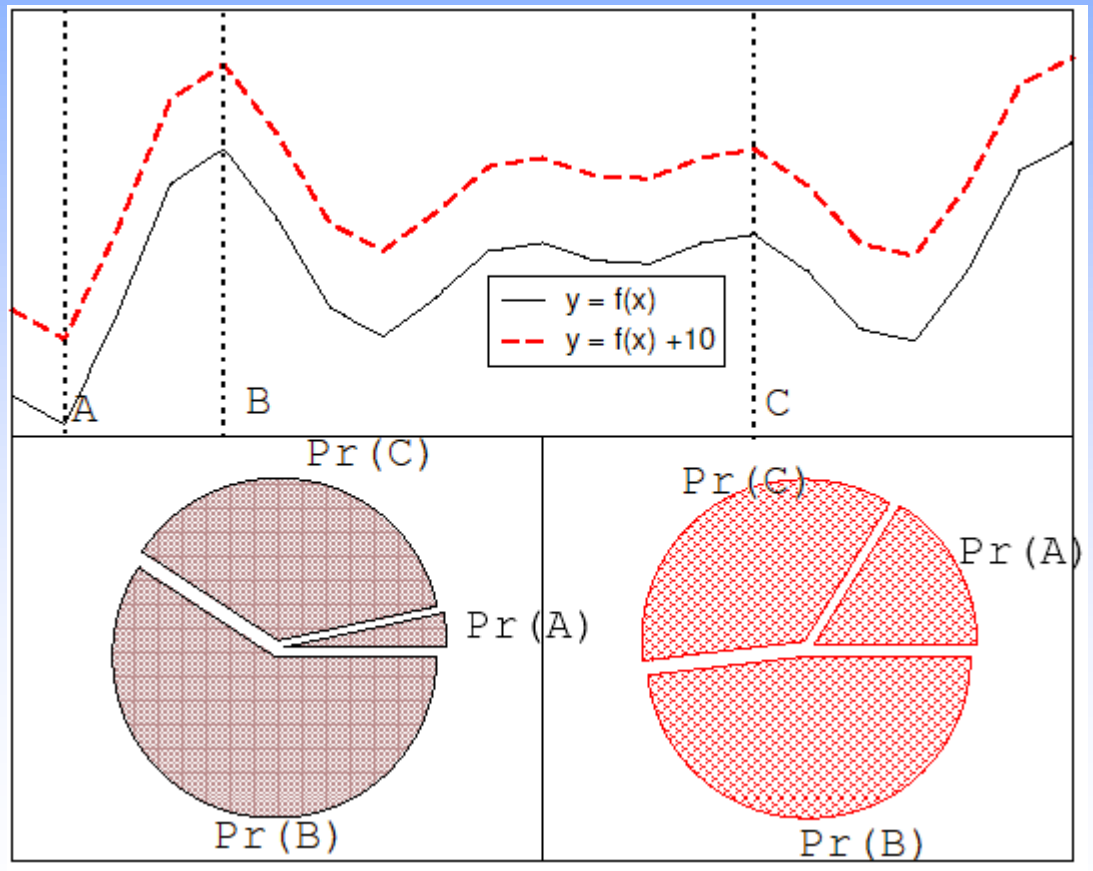
(μ = pop.size, f(i) = fitness of i, ⟨f⟩ avg. fitness in pop.)

- Roulette wheel algorithm:
  - Given a probability distribution, spin a 1-armed wheel *n* times to make *n* selections
  - No guarantees on actual value of n_i
- Baker's SUS algorithm:
  - *n* evenly spaced arms on wheel and spin once
  - Guarantees $floor(E(n_i)) \leq n_i \leq ceil(E(n_i))$

Fitness-Proportionate Selection

- Problems include
  - One highly fit member can rapidly take over if rest of population is much less fit: Premature Convergence
  - At end of runs when fitnesses are similar, lose selection pressure
  - Highly susceptible to function transposition
- Scaling can fix last two problems
  - Windowing: $f'(i) = f(i) - \beta^t$
    - where β is worst fitness in this (last n) generations
  - Sigma Scaling: $f'(i) = max(f(i) - (\langle f \rangle - c \cdot \sigma_f), 0.0)$
    - where c is a constant, usually 2.0

Function transposition for FPS



Rank – Based Selection

- Attempt to remove problems of FPS by basing selection probabilities on *relative* rather than *absolute* fitness
- Rank population according to fitness and then base selection probabilities on rank where fittest has rank μ and worst rank 1
- This imposes a sorting overhead on the algorithm, but this is usually negligible compared to the fitness evaluation time

Linear Ranking

$$P_{lin-rank}(i) = \frac{(2-s)}{\mu} + \frac{2i(s-1)}{\mu(\mu-1)}$$

- Parameterised by factor *s:* $1.0 < s \leq 2.0$
  - measures advantage of best individual
  - in GGA this is the number of children allotted to it
- Simple 3 member example

	Fitness	Rank	P_{selFP}	$P_{selLR} \quad (s=2)$	$P_{selLR} \quad (s=1.5)$
A	1	1	0.1	0	0.167
B	5	2	0.5	0.67	0.5
C	4	2	0.4	0.33	0.33
Sum	10		1.0	1.0	1.0

Exponential Ranking

$$P_{exp-rank}(i) = \frac{1 - e^{-i}}{c}$$

- Linear Ranking is limited to selection pressure
- Exponential Ranking can allocate more than 2 copies to fittest individual
- Normalise constant factor *c* according to population size

Tournament Selection

- All methods above rely on global population statistics
  - Could be a bottleneck esp. on parallel machines
  - Relies on presence of external fitness function which might not exist: e.g. evolving game players
- Informal Procedure:
  - Pick *k* members at random then select the best of these
  - Repeat to select more individuals

Tournament Selection 2

- Probability of selecting *i* will depend on:
  - Rank of *i*
  - Size of sample *k*
    - higher *k* increases selection pressure
  - Whether contestants are picked with replacement
    - Picking without replacement increases selection pressure
  - Whether fittest contestant always wins (deterministic) or this happens with probability *p*
- For *k* = 2, time for fittest individual to take over population is the same as linear ranking with $s = 2 \bullet p$

Survivor Selection

- Most of methods above used for parent selection
- Survivor selection can be divided into two approaches:
  - Age-Based Selection
    - e.g. SGA
    - In SSGA can implement as “delete-random” (not recommended) or as first-in-first-out (a.k.a. delete-oldest)
  - Fitness-Based Selection
    - Using one of the methods above or

Two Special Cases

- Elitism
  - Widely used in both population models (GGA, SSGA)
  - Always keep at least one copy of the fittest solution so far
- GENITOR: a.k.a. “delete-worst”
  - From Whitley’s original Steady-State algorithm (he also used linear ranking for parent selection)
  - Rapid takeover : use with large populations or “no duplicates” policy

Example application of order based GAs: JSSP

Precedence constrained job shop scheduling problem

- J is a set of jobs.
- O is a set of operations
- M is a set of machines
- *Able* $\subseteq$ O $\times$ M defines which machines can perform which operations
- *Pre* $\subseteq$ O $\times$ O defines which operation should precede which
- *Dur* : $\subseteq$ O $\times$ M $\rightarrow$ IR defines the duration of o $\in$ O on m $\in$ M

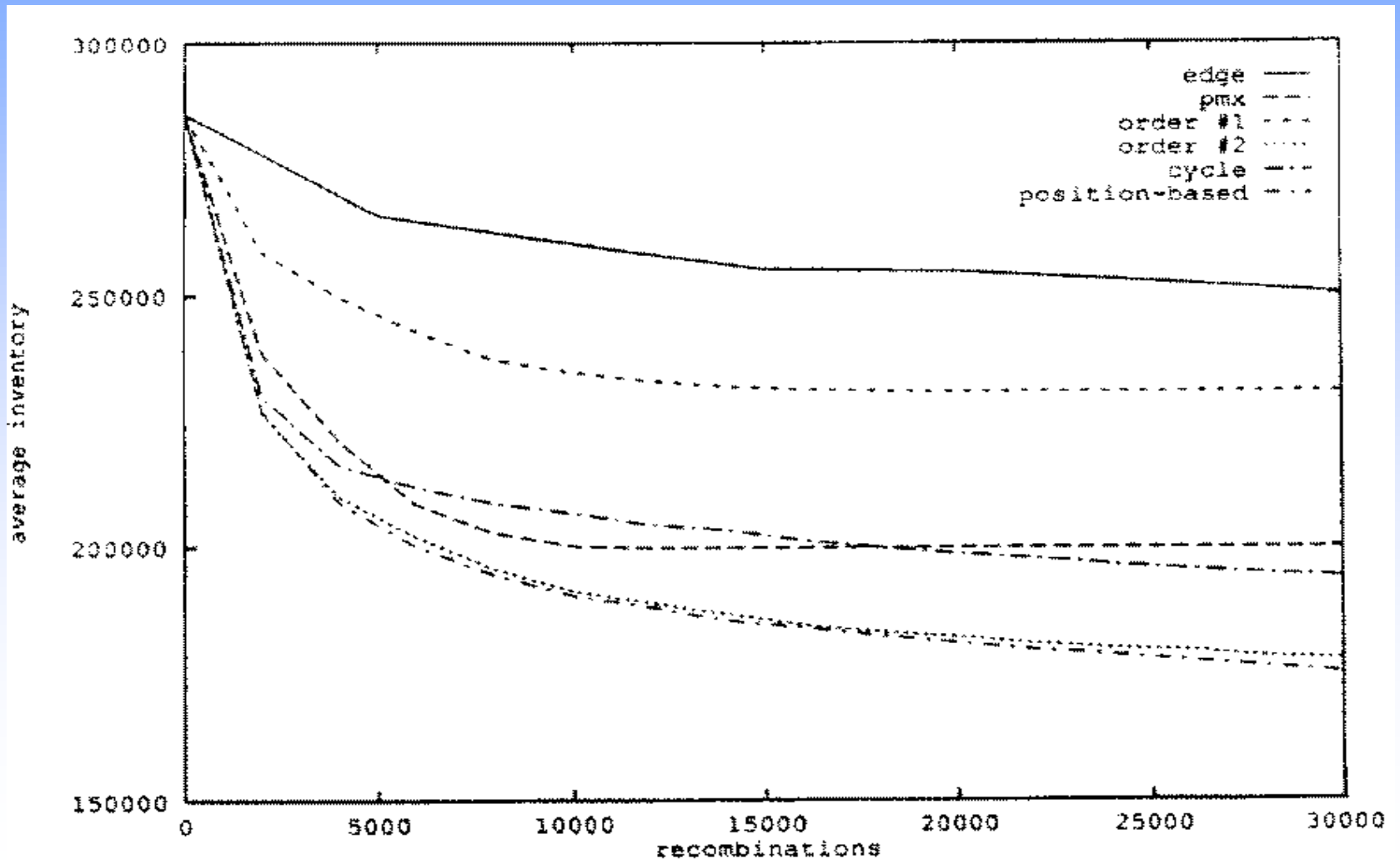
The goal is now to find a schedule that is:

- Complete: all jobs are scheduled
- Correct: all conditions defined by *Able* and *Pre* are satisfied
- Optimal: the total duration of the schedule is minimal

Precedence constrained job shop scheduling GA

- Representation: individuals are permutations of operations
- Permutations are decoded to schedules by a decoding procedure
  - take the first (next) operation from the individual
  - look up its machine (here we assume there is only one)
  - assign the earliest possible starting time on this machine, subject to
    - machine occupation
    - precedence relations holding for this operation in the schedule created so far
- fitness of a permutation is the duration of the corresponding schedule (to be minimized)
- use any suitable mutation and crossover
- use roulette wheel parent selection on inverse fitness
- Generational GA model for survivor selection
- use random initialisation

JSSP example: operator comparison

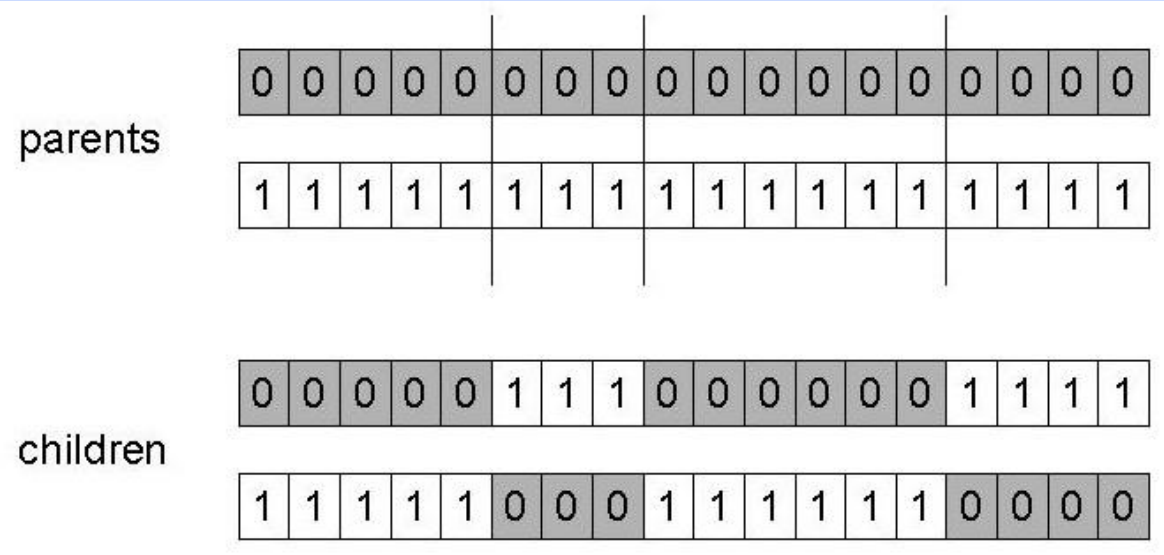


Alternative Crossover Operators

- Performance with 1 Point Crossover depends on the order that variables occur in the representation
  - more likely to keep together genes that are near each other
  - Can never keep together genes from opposite ends of string
  - This is known as *Positional Bias*
  - Can be exploited if we know about the structure of our problem, but this is not usually the case

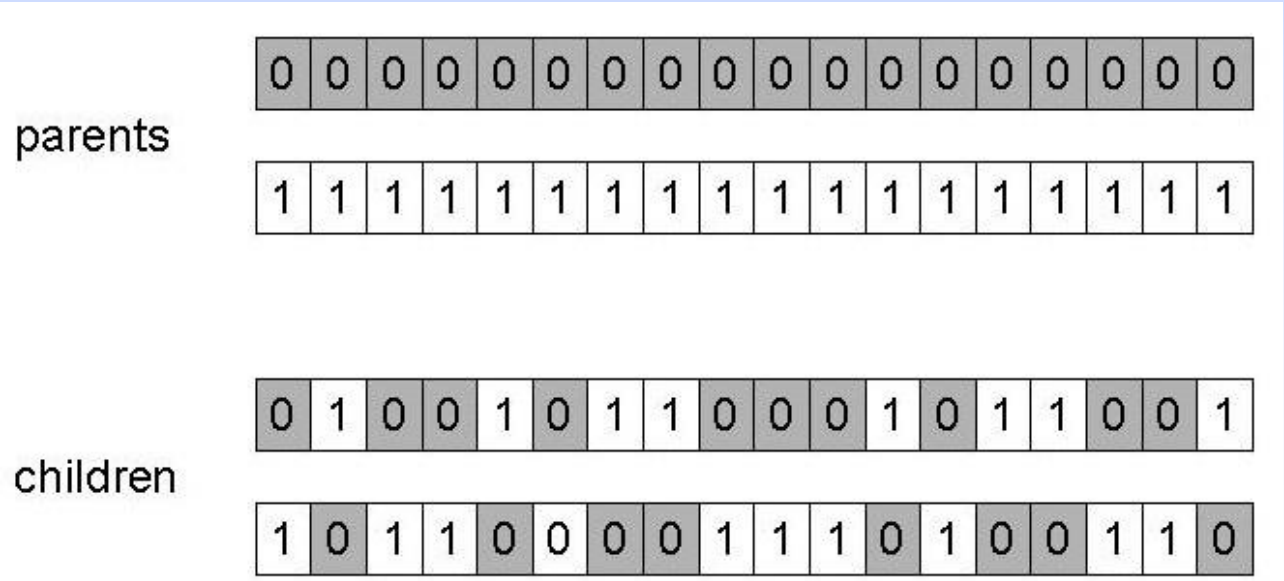
n-point crossover

- Choose n random crossover points
- Split along those points
- Glue parts, alternating between parents
- Generalisation of 1 point (still some positional bias)



Uniform crossover

- Assign 'heads' to one parent, 'tails' to the other
- Flip a coin for each gene of the first child
- Make an inverse copy of the gene for the second child
- Inheritance is independent of position



Crossover OR mutation?

- Decade long debate: which one is better / necessary / main-background

- Answer (at least, rather wide agreement):
  - it depends on the problem, but
  - in general, it is good to have both
  - both have another role
  - mutation-only-EA is possible, xover-only-EA would not work

Crossover OR mutation? (cont'd)

Exploration: Discovering promising areas in the search space, i.e. gaining information on the problem

Exploitation: Optimising within a promising area, i.e. using information

There is co-operation AND competition between them

- Crossover is explorative, it makes a *big* jump to an area somewhere "in between" two (parent) areas
- Mutation is exploitative, it creates random *small* diversions, thereby staying near (in the area of) the parent

Crossover OR mutation? (cont'd)

- Only crossover can combine information from two parents
- Only mutation can introduce new information (alleles)
- Crossover does not change the allele frequencies of the population (thought experiment: 50% 0's on first bit in the population, ?% after performing *n* crossovers)
- To hit the optimum you often need a 'lucky' mutation