Introduction to Genetic Algorithms

Guest speaker:

David Hales

www.davidhales.com

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

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***
- 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?
- 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
- 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

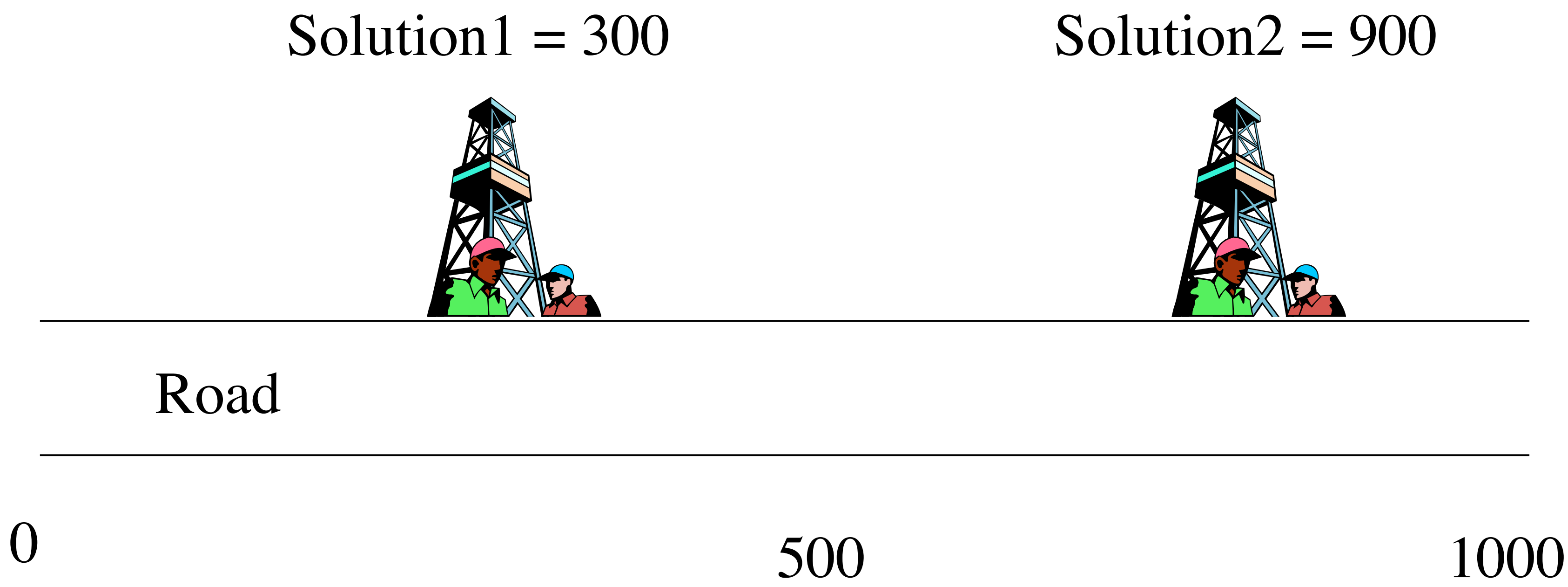
How do you encode a solution?

- 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?



Digging for Oil

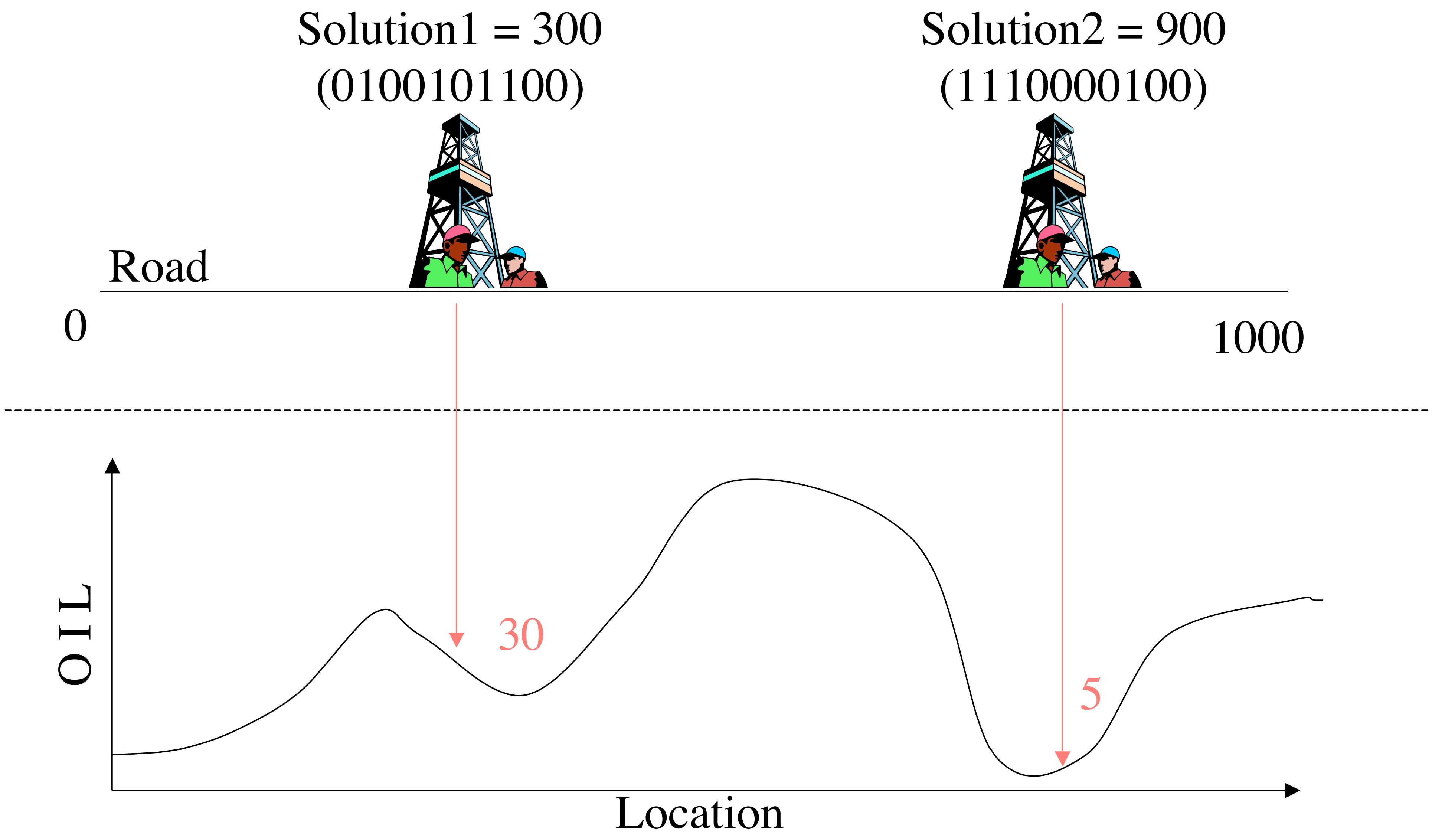
- The set of all possible solutions [0..1000] is called the *search space* or *state space*
- In this case it's just one number but it could be many numbers or symbols
- Often GA's code numbers in binary producing a bitstring representing a solution
- In our example we choose 10 bits which is enough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



Summary

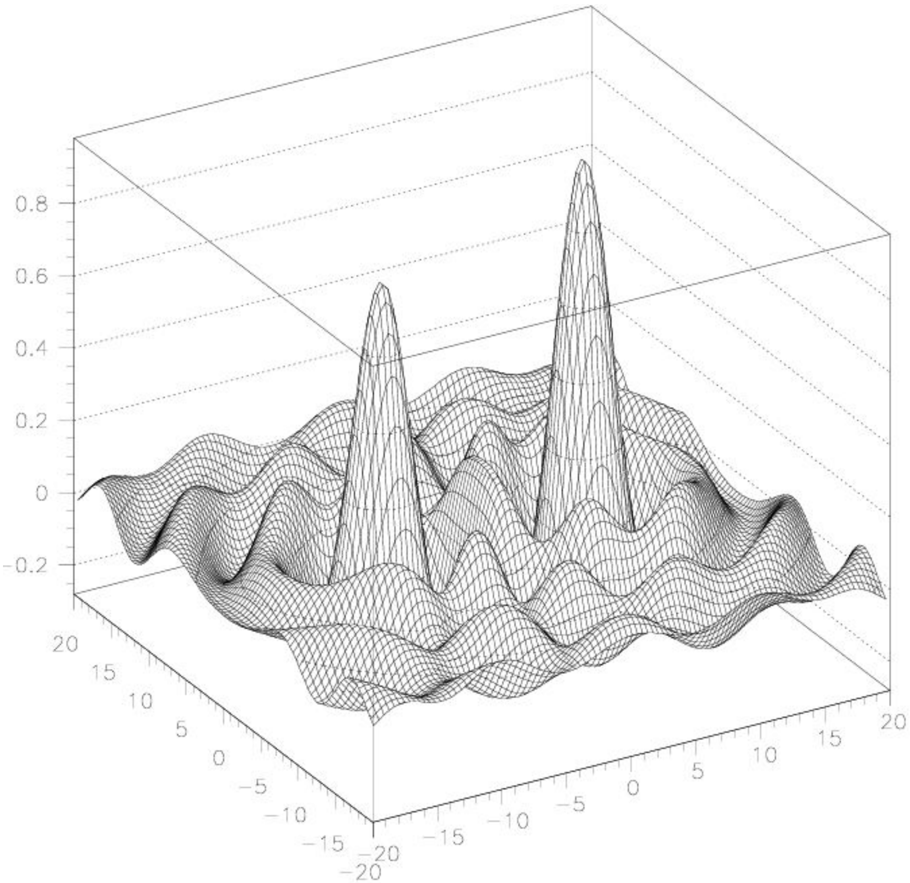
We have seen how to:

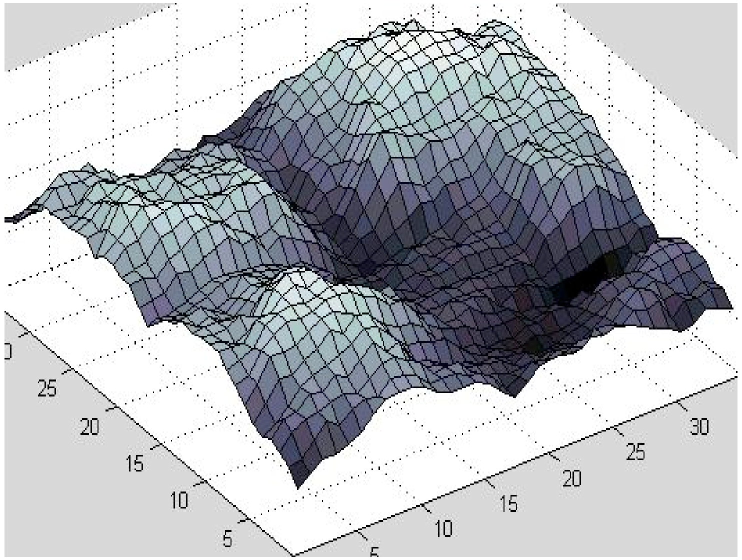
- represent possible solutions as a number
- encoded a number into a binary string
- generate a score for each number given a *function* of “how good” each solution is - this is often called a *fitness function*
- Our silly oil example is really optimisation over a function f(x) where we adapt the parameter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

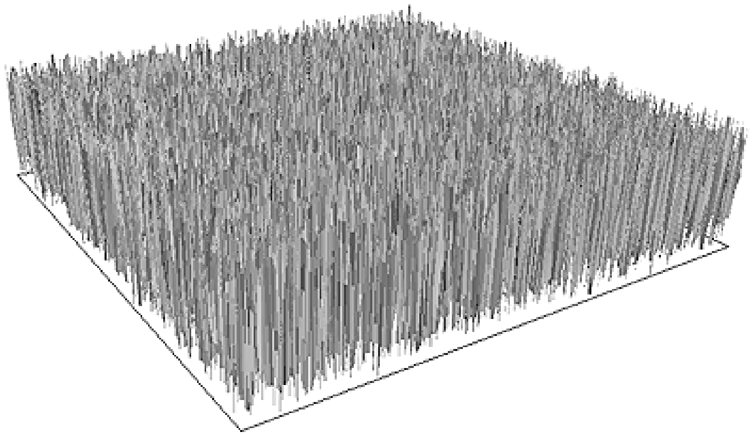
Search Space

- 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*)

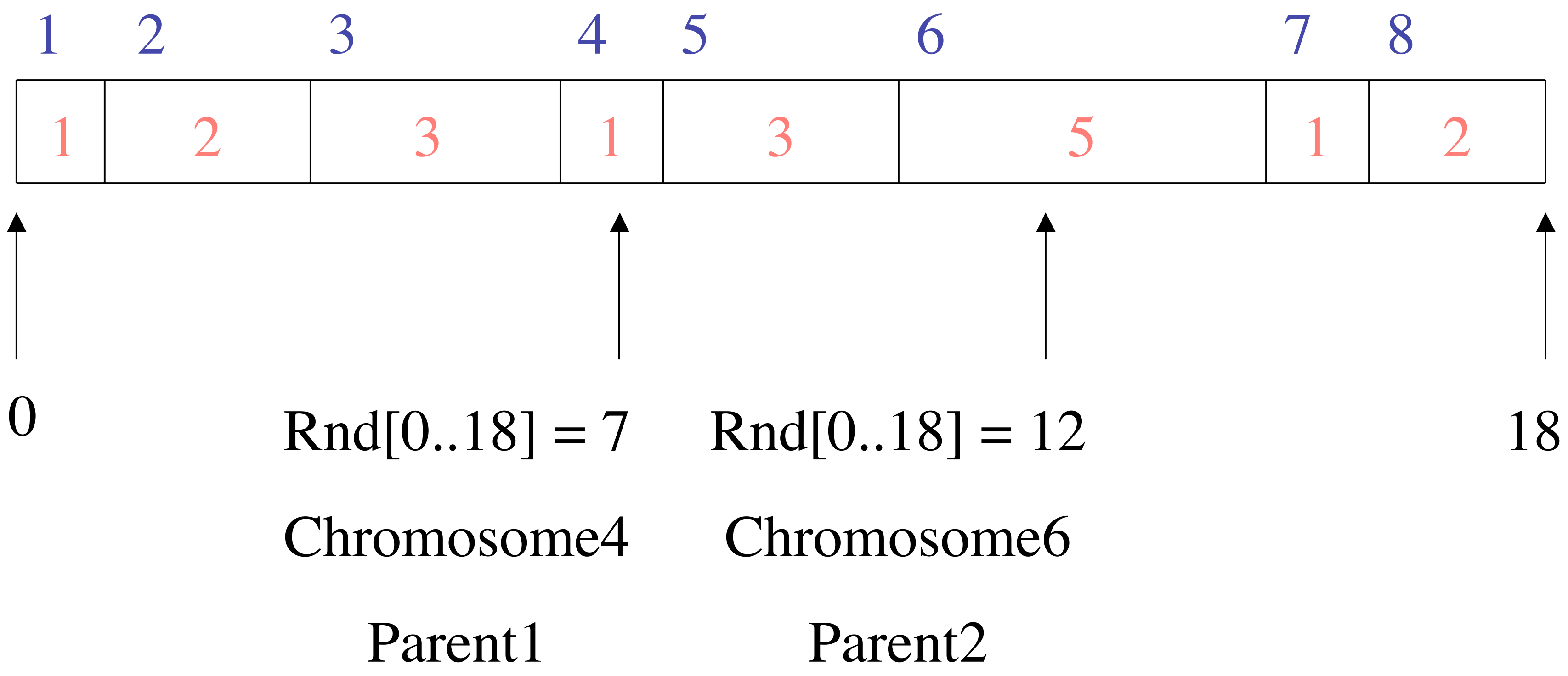
Selecting Parents

- 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

Roulette Wheel Selection



Crossover - Recombination

1010000000	Parent1	Offspring1	1011011111
1001011111	Parent2	Offspring2	1010000000

Crossover single point - random

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

Mutation

mutate

Offspring1 | 1011011111 | Offspring1 | 1011001111
Offspring2 | 1010000000 | Offspring2 | 1000000000

Original offspring

Mutated offspring

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

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: $N = 50$, $m = 0.05$, $c = 0.9$

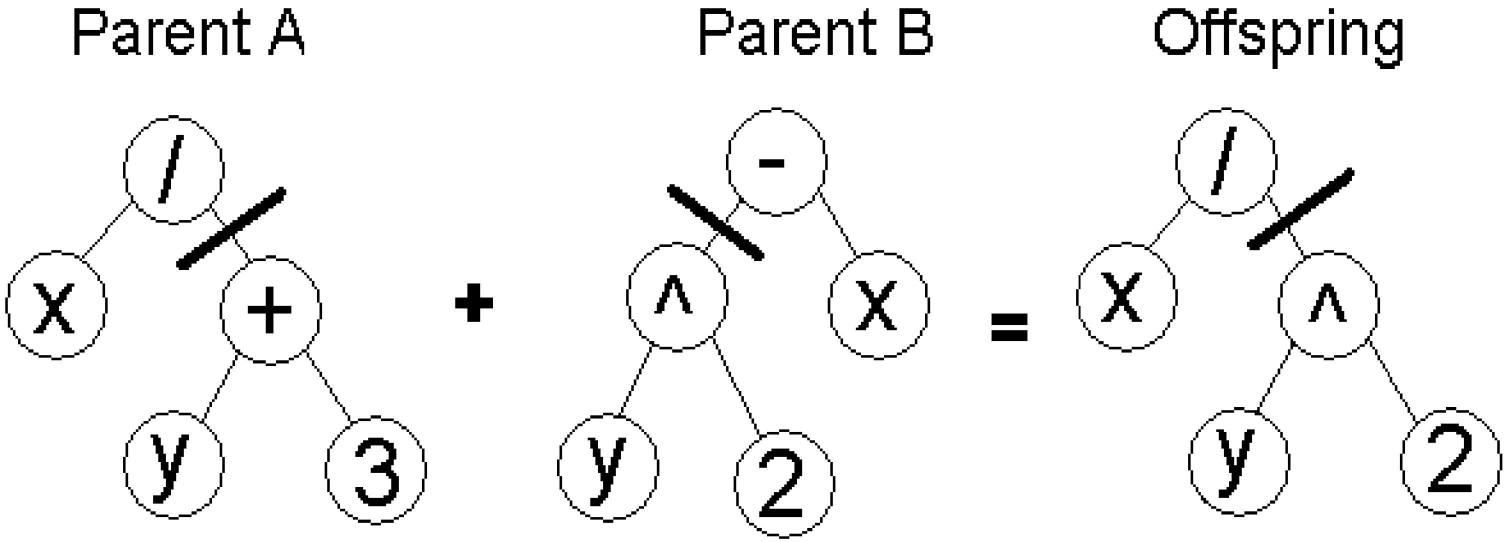
Why does crossover work?

- 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