INTELLIGENT COMPUTING

(COMPUTATIONAL INTELLIGENCE)
Evolutionary Computing - Genetic Algorithms in Detail

S. Sheridan

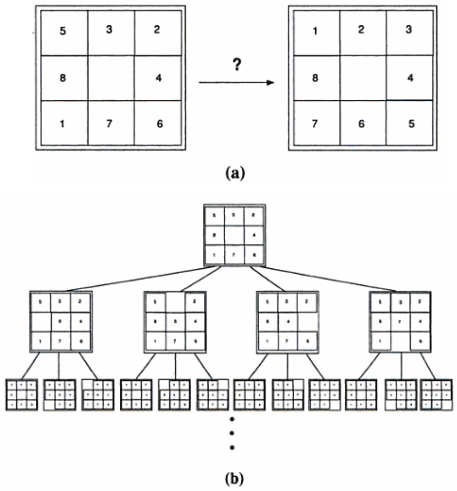
GENETIC ALGORITHMS AND TRADITIONAL SEARCH

- The category of problems that genetic algorithms are typically used to solve could be described roughly as SEARCH type problems. It is important to point out that the word "SEARCH" has at least three overlapping meanings in the context of computer science.
- **Search for stored data:** Here the problem is to efficiently retrieve information stored in computer memory.
- **Search for paths to goals:** Here the problem is to efficiently find a set of actions that will move from a given initial state to a given goal. This form of search is central to many approaches to Artificial Intelligence. For example, tile based puzzles.
- **Search for solutions:** This is a more general class of search than "search for paths to goals." The idea is to efficiently find a solution to a problem in a large space of candidate solutions.

GENETIC ALGORITHMS AND TRADITIONAL SEARCH

- There is a clear difference between the first kind of search and the second two. The first concerns problems in which one needs to find a piece of information (e.g. a telephone number) in a collection of stored information.
- In the second two, the information to be searched is not explicitly stored; rather candidates solutions are created as the search proceeds.
- For example, the AI search methods for solving the 8-Puzzle do not begin with a complete search tree in which all the nodes are already stored in memory.
- For most problems of *interest* there are too many possible nodes in the tree to store them all. Rather, the search tree is expanded step by step to find optimal or high-quality solutions by examine only small portions of the tree.

GENETIC ALGORITHMS AND TRADITIONAL SEARCH



(a) The problem is to find a sequence of moves that will go from the initial state to the state with the tiles in the correct order (the goal state). (b) A partial search tree for the 8−puzzle.

GENETIC ALGORITHMS AND OPTIMISATION PROBLEMS

- GA's have been used in a wide variety of optimisation tasks, including numerical optimisation and combinatorial problems as circuit layout and job-shop scheduling. One very well known optimisation problem is the travelling salesman problem.

- The **travelling salesman problem** (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? It is an **NP-hard** problem in combinatorial optimisation, important in operations research and theoretical computer science.

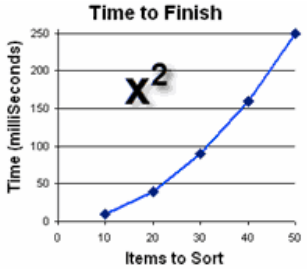
- The problem was first formulated in 1930 and is one of the most intensively studied problems in optimisation. It is used as a benchmark for many optimisation methods.

GENETIC ALGORITHMS AND OPTIMISATION PROBLEMS

http://www.youtube.com/watch?v=SC5CX8drAtU

WHAT ARE NP-HARD PROBLEMS?

- If you measure how long a program takes to run when given more and more difficult problems, such as sorting a list of 10 items, 20 items, 30 items etc, you can then plot the times and come up with a function.

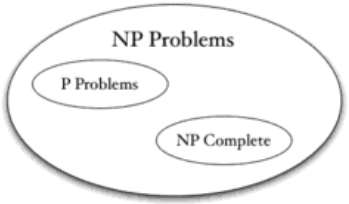


- For example the time might increase by x^2, so a problem that is twice as hard takes 4 times as long. That program would be in **"P"**, as it is solvable in "Polynomial" time. In this case the polynomial is: $\mathbf{t = x^2}$

WHAT ARE NP-HARD PROBLEMS?

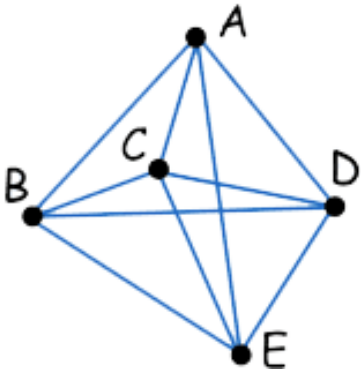
- If you had an "amazing" computer **N** (non-deterministic) that was not bound by the normal way a computer works which is step-by-step, you could solve lots of problems in **P** time.

- So, programs that take dramatically longer as the problem gets harder (i.e. not in "P") could be solved quickly on this amazing "N" computer and so are in "NP". Thus "NP" means "we can solve it in polynomial time if we can break the normal rules of step-by-step computing".



WHAT ARE NP-HARD PROBLEMS?

- Imagine you need to visit 5 cities on your sales tour. You know all the distances. Which is the shortest round-trip to follow?
- One obvious solution to the TSP is to check all possibilities. But this only works for small problems.



ABCEDA?
ADECBA?
AEDCBA?
…?

WHAT ARE NP-HARD PROBLEMS?

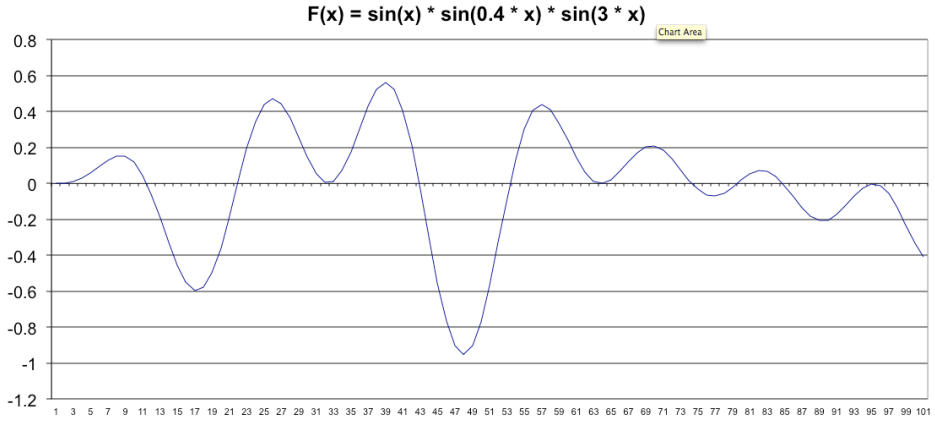
- If you add a new city it needs to be tried out in every previous combination. So this method takes "factorial time": t = (n-1)!
- Lets say your program could solve a 20-city problem in 1 second, then a 21-city problem would take about 21 seconds to solve. A 22-city problem would take about 462 seconds (over 7 minutes), and a 30-city problem would take 3 Million Years. Ouch!
- It is believed that if anyone could ever solve an "NP-Complete" problem in "P" time, then all "NP-complete" problems could also be solved that way by using the same method, and the whole class of "NP-Complete" would cease to exist.

GENETIC ALGORITHMS: EXAMPLE PROBLEM

- For the remainder of this lecture we will look at an example problem that can be solved using a genetic algorithm.
- We will introduce a basic framework for Genetic Algorithms, written in Java, that can be extended to cover different encoding techniques and operators.
- We will cover the encoding techniques and genetic operator implementations that are required for our example problem.
- In our lab session we will examine how the Java framework can be extended to deal with problems that require different encoding approaches.

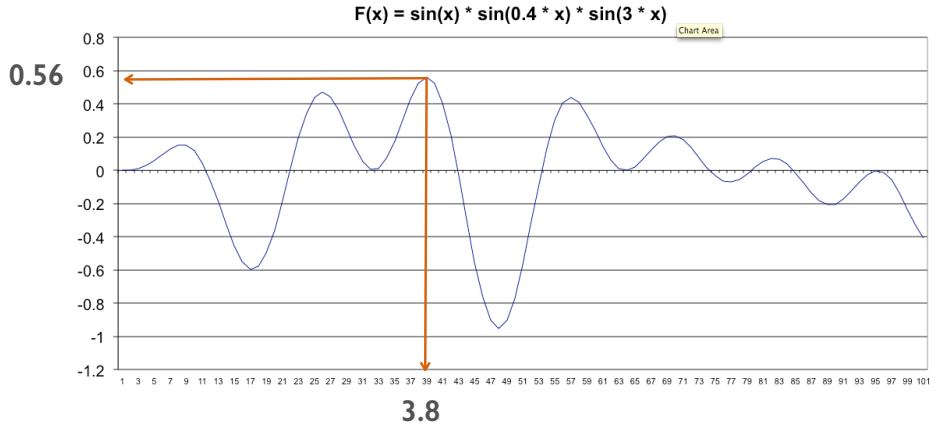
GENETIC ALGORITHMS - EXAMPLE

- Suppose we had a function F(x) with one independent variable X that was plotted over the interval [0.0,10.0].



GENETIC ALGORITHMS - EXAMPLE

- The problem that we want to solve is to find a value for X that will yield the largest possible value of F(X). The maximum value of **0.56** occurs at x = **3.8**



GENETIC ALGORITHMS - EXAMPLE

- While this problem can be solved trivially by a brute force search over the range of the independent variable X, the GA method scales very well to similar problems of higher dimensionality; for example we may have products of sine waves using 20 independent variables x1,x2,…x20.

- In this case brute force search would be prohibitively expensive. Even though this example is quite simple, it's a good place to start.

- Next we will need to choose an encoding scheme for the problem which will in turn determine how the genetic operators for crossover and mutation will be implemented.

CHOOSING AN ENCODING SCHEME

- In this problem we have only one parameter, the independent variable X. For this example, we will encode the parameter X using bit string encoding.

- So each possible solution (chromosome) in our population will consist of ten bits (so we have ten bit genes per chromosome). This means we can use the Java **BitSet** class with a length of 10 to represent a chromosome.

```
private BitSet genes;
genes = new BitSet(10);
```

- Therefore, each chromosome in our population will represent a floating point number in the range [0.0,10.0]. For example:

0	1	1	1	1	0	1	0	0	0

= **4.77**

CONVERTING BINARY STRINGS

- A little binary refresher …

512	256	128	64	32	16	8	4	2	1
0	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**

0 + 256 +128 + 64 + 32 + 0 + 8 + 0 + 0 + 0 = 488

488 / 102.3 = 4.77

512	256	128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1	1	1

512 +256 + 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 1023

1023 / 102.3 = 10

CONVERTING BINARY STRINGS

- So a function to convert our ten bit binary string to a floating point number would look as follows:

```
// Convert BitSet into float value
float geneToFloat(){
   int base = 1;
   float x = 0;
   for (int j=0; j<length; j++){
      if (genes.get(j)){
         x += base;
      }
      base *= 2;
   }
   x /= 102.3f;
   return x;
}
```


CODING A CHROMOSOME

- Once we have decided on how to encode to the problem we can begin to write the Java code to implement our genetic algorithm. The primary data that we will need to work with will be an array of chromosomes, each with an associated fitness value.

```
public class BinaryEncodedChromosome{

   private BitSet genes;       // Chromosome genes
   private int length;         // Chromosome length
   private float fitness;      // Fitness value for chromosome

   public BinaryEncodedChromosome(int length){//todo}
   public boolean getGeneAt(int pos) {// todo}
   public void setGeneAt(int pos, boolean val) {// todo}
   public void mutateGeneAt(int pos) {//todo}
   public float geneToFloat(){// todo}
   :
   :
}
```


CODING A POPULATION

- Next we need to define a population class to store an array of Chromosome objects. This population class will contain most of the genetic operators methods such as selection, crossover and mutation routines.

```
public class Population{

    private int population_size;       // No. of chromosomes in pop
    private int chromosome_length;     // No. of genes per chromosome
    private BinaryEncodedChromosome[] population; // Population
    private int [] rouletteWheel;      // Used for selecting parents
    private int rouletteWheelSize;     // Size of roulette wheel
    private float crossoverFraction;   // % of crossover
    private float mutationFraction;    // % of mutation

    public Population(int population_size, int chromosome_length, float
                      crossoverFraction, float mutationFraction){//todo}
    public void initialisePopulation(){//todo}
    public void sort(){//todo}
    public void evaluate(){//todo}
    public void crossover(){//todo}
    public void mutate(){//todo}
    :
    :
}
```


CODING A POPULATION

- One important point to make about the Population class is that it takes the parameters that will control the genetic algorithm as its arguments. This allows us to easily test out different population sizes, crossover fractions and mutation fractions without having to change code.

```
public Population(int population_size, int chromosome_length,
      float crossoverFraction, float mutationFraction){//todo}
```

- The **crossover_fraction** sets the fraction of chromosomes in the population that are considered for the genetic crossover operation.

- The **mutation_fraction** sets the fraction of chromosomes that undergo mutation each generation.

CODING A POPULATION

- One of the first things we must do is create the initial population of chromosomes. To start off with the chromosomes will be given random values.

- For each chromosome in our population we will randomly set each BIT to either 1 or 0. This can be easily done when we create a new Chromosome object by writing code in the Chromosome constructor that will initialise all the BITS in its genes (BITSET).

```
// Initialises the genes to random bits
for (int j=0; j<length; j++)
{
   if (Math.random() < 0.5)
      genes.set(j);
   else
      genes.clear(j);
}
```


CODING A POPULATION

- Once we have initialised our population we need to evaluate each chromosome in order to rank them based on their fitness. We will need to define a fitness function that will allow us to rank chromosomes. In this case the fitness function is the same as the function that generated the graph.

```
private float fitness(float x)
{
   return (float)(Math.sin(x) * Math.sin(0.4f * x)
              * Math.sin(3.0f * x));
}

public void evaluate(){
   for(int j = 0; j < population_size; j++){
      float val = population[j].geneToFloat();
      population[j].setFitness(fitness(val));
   }
}
```


CODING A POPULATION

- We can sort the population once we have calculated and stored the fitness for each chromosome. It doesn't matter which sorting algorithm we use and the order could be ascending or descending depending on the problem. The overall speed of our genetic algorithm will obviously benefit from a quick sorting algorithm.

```
public void sort(){
    int i, j, first, temp;
    for ( i = population_size - 1; i > 0; i -- ){
      first = 0;   //initialize to subscript of first element
      //locate smallest element between positions 1 and i.
      for(j = 1; j <= i; j ++){
         if (population[j].getFitness() < population[first].getFitness())
           first = j;
         }
         swap(first, i);
      }
}
```


CODING A SELECTION ROUTINE

- Now that the population has been sorted based on fitness, we can select some parents in order to carry out the crossover operation. This can be achieved using a simple implementation of the roulette wheel selection technique.

- As can be seen from the diagram above, all we need to do is build an array containing repeating indices of population members. Each index values repeat run will decrease as the index number increases. This means weaker population members will have less index entries in the array. The weakest population members may not have an entry in the roulette wheel at all.

CODING THE CROSSOVER OPERATOR

```
public void crossover(){
    int num = (int)(population_size * crossoverFraction);
    for (int i=num; i < population_size; i++)
    {
        int c1 = (int)(rouletteWheelSize * Math.random() * 0.9999f);
        int c2 = (int)(rouletteWheelSize * Math.random() * 0.9999f);
        c1 = rouletteWheel[c1];
        c2 = rouletteWheel[c2];
        if (c1 != c2)
        {
            // Perform single point crossover
            int locus = 1 + (int)((chromosome_length - 2) * Math.random());
            for (int g=0; g < chromosome_length; g++)
            {
                if (g < locus) // Copy from parent c1 up to locus g
                    population[i].setGeneAt(g, population[c1].getGeneAt(g));
                else // Copy from parent c2 after locus g
                    population[i].setGeneAt(g, population[c2].getGeneAt(g));
            }
        }
   }
}
```


CODING THE MUTATION OPERATOR

```
public void mutate(){
    int num = (int)(population_size * mutationFraction);
    for (int i=0; i<num; i++){
        // Random chromosome
        int c = (int)(population_size * Math.random() * 0.99);
        // Random gene
        int g = (int)(chromosome_length * Math.random() * 0.99);
        population[c].mutateGeneAt(g);
    }
}

public void removeDuplicates(){
    // i>3 dont touch the top of the population
    for (int i=population_size - 1; i>3; i--)
    {
        for (int j=0; j<i; j++)
        {
            if (population[i].equals(population[j])){
                int g = (int)(chromosome_length * Math.random() * 0.99);
                population[i].mutateGeneAt(g);
                break;
            }
        }
    }
}
```


PUTTING IT ALL TOGETHER

- Now that the selection routine and genetic operators have been implemented it would be useful to define a method that will put all of the operations together to carry out a single evolution.

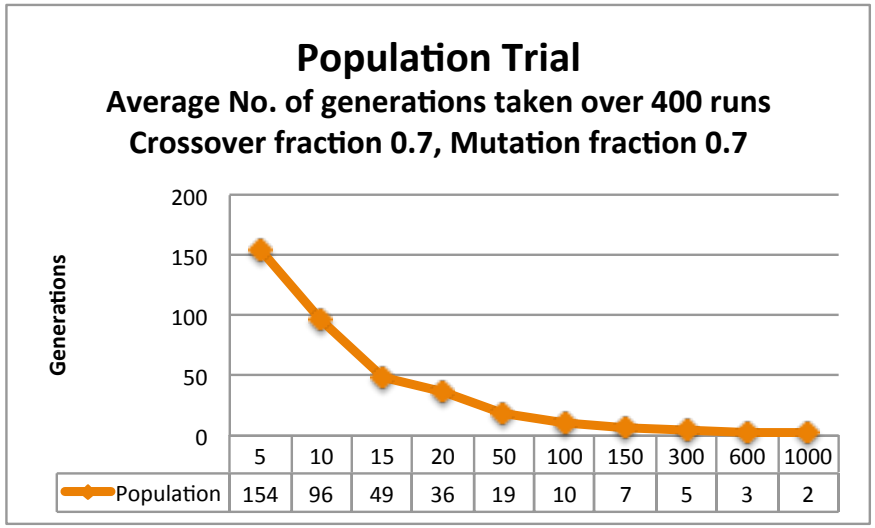
```
public void evolve(){
  evaluate();
  sort();
  crossover();
  mutate();
  removeDuplicates();
}
```


PUTTING IT ALL TOGETHER

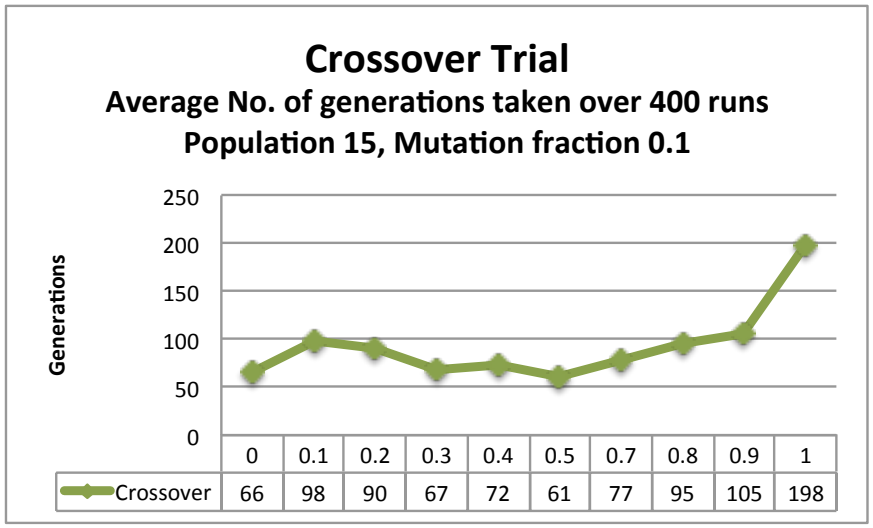
- Next we will need to pull all of the this code together by writing a main driver program to control the number of generations our genetic algorithm will run for.

```
public static void main(String[] args){

    // Create a population using particular trial data
    Population p = new Population(POPULATION_SIZE, CHROMOSOME_LENGTH,
                                  CROSSOVER_FRACTION, MUTATION_FRACTION);

    for(int gen = 0; gen < NUM_GENERATIONS; gen++){

        System.out.println("*** Generation " + gen + " ***");

        p.evolve();
        p.output();
    }
}
```

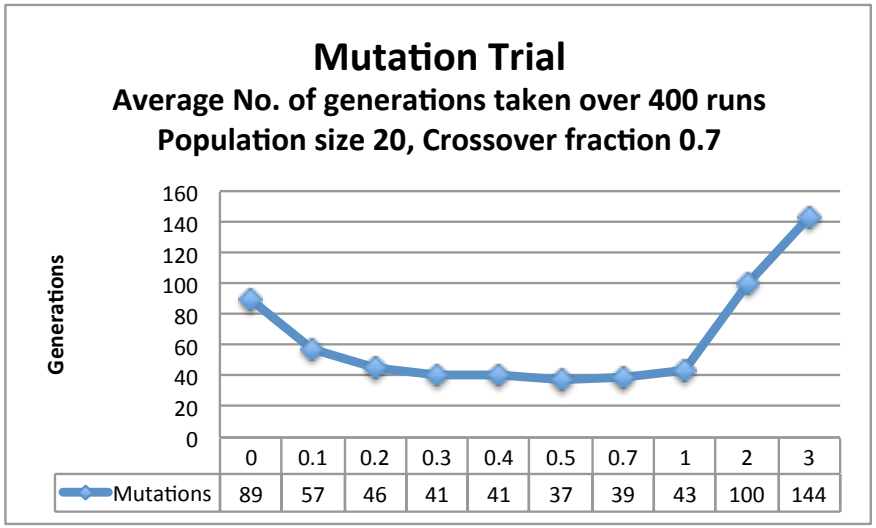
TESTING GENETIC PARAMETERS



TESTING GENETIC PARAMETERS



TESTING GENETIC PARAMETERS



WHY GA'S? ADVANTAGES

- Dilemma of global optimum vs many local optima. GA strike perfect balance (John Holland - Genetic Algorithm 1992)
- GA's can manipulate parameters simultaneously (Forrest - Genetic Algorithms: Principles of natural selection applied to computation. 1993)
- GA's don't have specific knowledge of the problem. All possible search pathways are considered. (John Koza - Genetic Programming III 1999)

WHY GA'S? DISADVANTAGES

- Computationally expensive and time consuming (fitness function)
- Issues in representation of problem
- Proper writing of fitness function
- Proper values of size of population, crossover and mutation rates.
- Deceptive Fitness Function (Mitchell, Melanie 1996)
- Premature convergence
- No one mathematically perfect solution since problems of biological adaptation dont have this issue.

APPLICATIONS