

## Lecture 2: Representations and Operators

# Representations and Operators

- ▶ Representations
  - ▶ Binary encoding is not the only representation
- ▶ Operators
  - ▶ Single-point crossover and mutation are not the only genetic operators
- ▶ In this lecture we will examine other representations and operators

# Representations

# Representations

- ▶ Recall that the SGA uses binary chromosomes
- ▶ As we know, anything that can be represented, can be represented in binary
  - ▶ To some arbitrary precision
- ▶ Is binary always the best way to encode solutions?

# Representations - Sparseness

- ▶ Consider the following example
  - ▶ Problem - find the largest integer in  $\{0, 1, \dots, 8\}$
  - ▶ Encoded as standard binary the fitness function is

<i>Chromosome</i>	<i>Fitness</i>
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	undefined
1010	undefined
1011	undefined
1100	undefined
1101	undefined
1110	undefined
1111	undefined

# Representations - Sparseness

- ▶ The problem with the encoding in the previous example is fairly obvious
- ▶ The encoding is very *sparse*
  - ▶ Mathematically, the function mapping the genotype set to the phenotype set is not *surjective* (*onto*)
  - ▶ Nearly 50% of chromosomes encode invalid solutions to the problem
- ▶ This will seriously hamper the efficiency of the GA's search
- ▶ A less sparse encoding would be in base 3
  - ▶ 2 genes, each with 3 alleles
  - ▶  $3^2 = 9 \implies$  all chromosomes are valid solutions

# Representations

- ▶ Consider the following example
  - ▶ Problem - find the largest integer in  $\{0, 1, \dots, 15\}$
  - ▶ Encoded as standard binary the fitness function is

<i>Chromosome</i>	<i>Fitness</i>
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

# Representations - Discontinuity

- ▶ A further problem with the standard binary encoding used in the previous example is less obvious
- ▶ The 'Hamming distance' between chromosomes encoding adjacent integers is not constant
  - ▶ Hamming distance is the number of genes at which two chromosomes have different alleles
- ▶ Chromosomes that differ in only one or two bits may encode for substantially different solutions
  - ▶  $0000 \rightarrow 0$  vs.  $1000 \rightarrow 9$
- ▶ Chromosomes that differ in all bits may encode for very similar solutions
  - ▶  $1000 \rightarrow 9$  vs.  $0111 \rightarrow 8$
- ▶ N.B. The relationship between chromosome and solution is known as the *genotype-phenotype mapping*
  - ▶ Again, the terminology is directly taken from biology

# Representations - Gray Coding

- ▶ One proposed solution is to use a 'Gray code'
  - ▶ Under binary reflected Gray coding the fitness function is

<i>Chromosome</i>	<i>Fitness</i>
0000	0
0001	1
0011	2
0010	3
0110	4
0111	5
0101	6
0100	7
1100	8
1101	9
1111	10
1110	11
1010	12
1011	13
1001	14
1000	15

# Representations - Gray Coding

- ▶ Now adjacent integers are encoded by chromosomes that only differ in one gene
- ▶ This seems like an improvement over traditional binary encoding
- ▶ But how much of an improvement is it really?
- ▶ For example, what are the average effects of a single point mutation under both encoding schemes?
  - ▶ *Exercise:* calculate the expected deviation in encoded integer resulting from a single uniformly selected point mutation in a 4-gene binary chromosome, under standard and Gray binary encodings. Also calculate the variance of the expectation. Write a computer program to help you if necessary.
- ▶ Crossover is likely to be much more disruptive than mutation under either encoding
  - ▶ Exact calculation of expected disruption would be much more involved though...

# Representations - Permutation and Grouping Problems

- ▶ The comparison between standard and Gray binary encoding is illustrative of a general point
  - ▶ Careful selection of a representation appropriate to the optimisation problem is required
- ▶ ‘Function-based’ problems are not the only optimisation problems
- ▶ Other very important classes of problems are ‘permutation’ and ‘grouping’ problems
  - ▶ These problems have very different characteristics to function-based problems
  - ▶ They include problems such as
    - ▶ Route optimisation (e.g. Travelling Salesman Problem)
    - ▶ Sequence optimisation (e.g. Job Shop Scheduling Problem)
    - ▶ Payload optimisation (e.g. Bin Packing Problem)
    - ▶ ...

# Representations - Permutation and Grouping Problems

- ▶ Permutation and grouping problems typically suffer from very sparse genotype-phenotype mappings
  - ▶ Many genotypes do not encode valid solutions
- ▶ Consider a simple encoding for the 5-city Travelling Salesman Problem (TSP)
  - ▶ Solutions are permutations on the set of cities  $\{A, B, C, D, E\}$
  - ▶ e.g.  $\langle A, C, D, E, B \rangle$
  - ▶ Solutions must be permutations (i.e. every element in the set appears exactly once)
  - ▶ Adjacent genes in the chromosome must have alleles representing adjacent cities in the TSP graph
- ▶ The encoding is also highly *redundant*
  - ▶ Mathematically, the function mapping the genotype set to the phenotype set is not *injective* (*one-to-one*)
  - ▶  $\langle A, C, D, E, B \rangle$  is the same solution as  $\langle C, D, E, B, A \rangle$ , which is the same as  $\langle D, E, B, A, C \rangle$ , etc.
  - ▶ Assuming an undirected graph gives even higher redundancy

# Representations - Permutation and Grouping Problems

- ▶ For grouping problems, solutions represent an allocation of items to sets
- ▶ For example the bin-packing problem
  - ▶ Items of different weights are allocated to bins of fixed capacity
  - ▶ Objective function to be minimised is number of bins used
- ▶ All items must appear in the solution exactly once
  - ▶ Items cannot appear in multiple bins
  - ▶ Items cannot appear in the same bin more than once
- ▶ A solution can be thought of as a permutation, which we can decode using a simple heuristic
  - ▶ Put each item into the first empty bin in which it will fit, or a new bin if no such bin exists
- ▶ A simple permutation encoding will be very sparse and redundant
  - ▶ E.g.  $\langle 1, 3, 2|4, 5, 6 \rangle$  is the same solution as  $\langle 2, 1, 3|6, 4, 5 \rangle$

# Representations - Permutation and Grouping Problems

- ▶ Sparseness and redundancy mean simple crossover is highly *destructive*
  - ▶ E.g. for the TSP example  $\langle A, C, D, E, B \rangle$  crossed with  $\langle D, A, B, E, C \rangle$  using 1X is *guaranteed* to result in an invalid solution
  - ▶ Similarly,  $\langle A, C, D, E, B \rangle$  crossed with  $\langle C, D, E, B, A \rangle$  using 1X results in an invalid solution, *even though both original chromosomes encode the same solution*
- ▶ Similarly, a single point mutation is *guaranteed* to result in an invalid solution
- ▶ Redundancy and the destructive effects of operators will make the GA's search very inefficient
- ▶ *Exercise*: For a problem whose solutions are permutations of 5 objects, what proportion of chromosomes encoded using one gene per position in the permutation are valid solutions? Derive a general expression for encodings with  $\ell$  genes and alleles

# Operators

# Operators - Crossover Bias

- ▶ Crossover is often considered to be the defining feature of a GA and the source of its power
- ▶ Single point crossover (1X) is not the only possible crossover operator
- ▶ It is worth examining the 'bias' of the 1X operator
- ▶ Consider what happens when we apply 1X to the chromosome  $\langle a_1, a_2, \dots, a_\ell \rangle$ 
  - ▶ The probability that alleles at two different genes both end up in the same offspring chromosome together strongly depends on the distance (number of other genes) between them
  - ▶ In particular, note that 1X as described in the previous lecture means that  $a_1$  and  $a_\ell$  will *never* end up in the same offspring chromosome together
  - ▶ 1X exhibits strong *positional bias*
- ▶ On the other hand, 1X has no *distributional bias*
  - ▶ The crossover point is uniformly selected

# Operators - Multipoint Crossover

- ▶ There is no particular reason to only have a single crossover point
- ▶ We can define an  $m$ -point crossover operator MX
  - ▶  $m$  crossover points selected uniformly *without replacement*
    - ▶ Sampling without replacement necessary to ensure exactly  $m$  points are selected
  - ▶ Crossover points works in exactly the same way as 1X
- ▶ *Multipoint crossover* reduces positional bias

# Operators - Uniform Crossover

- ▶ To remove any positional bias we can make crossover completely random
- ▶ *Uniform crossover*
  - ▶ During recombination, a (weighted) coin is tossed for each gene, to see which parent the offspring should receive its allele from
  - ▶ I.e. we generate a crossover mask by sampling from a Bernoulli distribution
    - ▶ E.g. 1010001 where 1 indicates first parent contributes the allele, and 0 indicates second parent
  - ▶ We can vary the Bernoulli parameter  $p$  in  $(0, 0.5]$  to make crossover more or less like clonal reproduction
- ▶ *Exercise:* Calculate the probability of two alleles with  $l$  genes between them, on a parental chromosome of length  $\ell$ , ending up together in the same offspring chromosome, for 1X, for 2X, and for UX with arbitrary  $p$  (assume two offspring are created by the crossover operators)

# Operators - Nonlinear Crossover

- ▶ All the crossover operators we've looked at so far can be thought of as *linear*
- ▶ As we've seen for permutation problems, linear crossover is inappropriate
- ▶ One solution is *partially matched crossover* (PMX)
  - ▶ Uniformly select two crossover points between 1 and  $\ell - 1$
  - ▶ Genes between these crossover points specify an interchange mapping
    - ▶ For the genes between the crossover points we specify mappings between the alleles of the parents  $a$  and  $b$  of the form  $a_i \leftrightarrow b_i$
    - ▶ We rewrite the chromosomes of both parents using these mappings to produce two offspring chromosomes
    - ▶ What should happen if the same gene appears in more than one mapping?
- ▶ PMX is one example of *order* crossover

# Operators - Mutation

- ▶ Simple mutation on binary chromosomes is straightforward
  - ▶ Per gene probability of bit-flipping  $\mu$
- ▶ We may also choose to have a constant number of mutations per chromosome
- ▶ What about higher cardinality allelic alphabets?
  - ▶ We may sample uniformly from the other possible alleles
  - ▶ Or it might make more sense to make small mutations frequent and large mutations rare
  - ▶ Should assign zero probability of sampling the current allele, to ensure mutation really does occur
    - ▶ I.e. weight probability of mutation to any allele by the absolute difference from the current allele
- ▶ How can we design a mutation operator for permutation problems?
  - ▶ Recall that single point mutation is guaranteed to result in an invalid solution

# Operator Implementation

- ▶ Design of operators is crucial for GA's search performance
- ▶ Efficient implementation of operators is also important for run-time performance
  - ▶ These operators are going to be applied very many times during GA execution
- ▶ As with most algorithms, there are typically naïve inefficient implementations, and clever efficient ones
- ▶ For example operators can be implemented with bit masks
- ▶ Mutation (binary chromosomes only)
  - ▶  $a \oplus m$
- ▶ Linear crossover (arbitrary vector chromosomes)
  - ▶  $m \otimes a \oplus \bar{m} \otimes b$
- ▶ Where  $m$  is the operator binary mask,  $\bar{m}$  is its complement,  $a$  and  $b$  are chromosomes, and  $\otimes$  and  $\oplus$  are component-wise multiplication and addition respectively

# Operator Implementation

- ▶ For binary chromosomes these operations can be performed exceedingly efficiently using bitwise AND and XOR operations
- ▶ The mask can typically be generated efficiently as well
  - ▶ For mutation and crossover we could start with a binary string of 0s
  - ▶ For each bit we then set it to one by sampling from a Bernoulli distribution with appropriate  $p$
  - ▶ This is inefficient for long strings though
  - ▶ We could do it more efficiently...
    - ▶ The number of expected successes  $n$  in  $N$  Bernoulli trials is given by a binomial distribution  $P_p(n|\ell)$
    - ▶ We start with the 0s string as before
    - ▶ We sample our number of mutations/crosses  $n$  from  $P_p(n|\ell)$
    - ▶ We uniformly sample (without replacement)  $n$  points on the string and set those bits to 1
    - ▶ As the mean number of successful Bernoulli trials is  $\ell p$  our approach is roughly  $\frac{1}{p}$ -times more efficient (assuming efficient uniform and binomial sampling, reasonable given for long strings  $E(n) \ll \ell$ )