Communications Network Design

lecture 15

Matthew Roughan
<matthew.roughan@adelaide.edu.au>

Discipline of Applied Mathematics
School of Mathematical Sciences
University of Adelaide

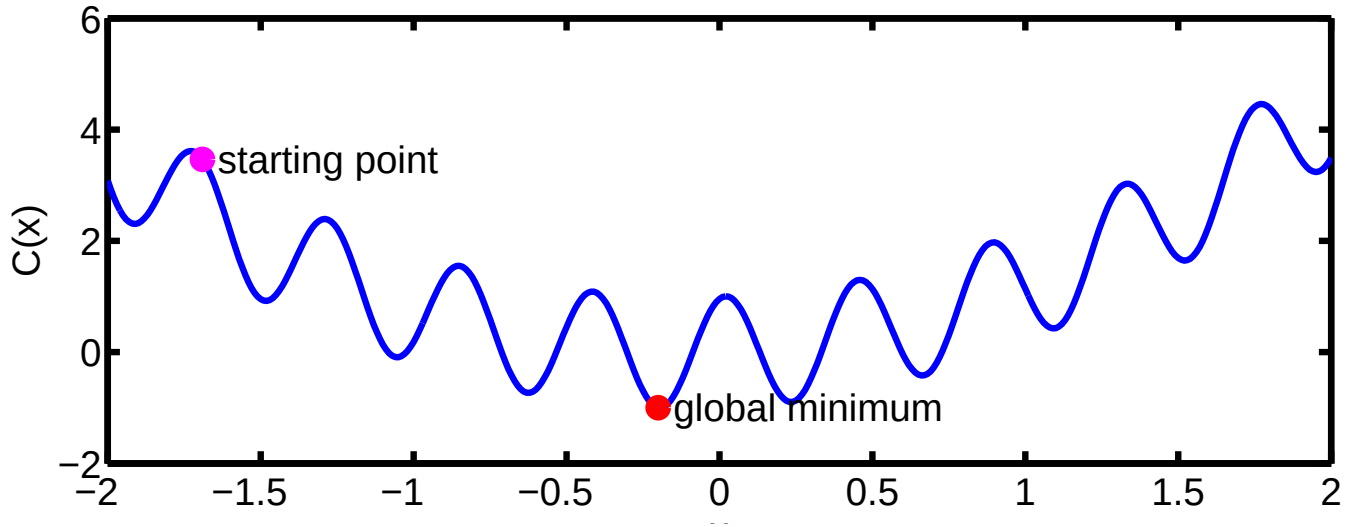
May 20, 2009

Randomized algorithms: genetic algorithms

Genetic algorithms (sometimes called evolutionary computing) work by analogy to Darwin's theory of evolution. We generate populations of solutions, and allow them to "evolve" towards fit solutions (solutions that minimize our objective). GAs have advantages in flexibility: they can even be applied when the objective function isn't known.

Randomized algorithms

- saw simulated annealing
- similar context today
  - non-convex search space, with local minima



- different approach: Genetic algorithms (GAs)

Genetic algorithms

A set of randomized algorithms which derive their behavior from a metaphor of the processes of **evolution** in nature.

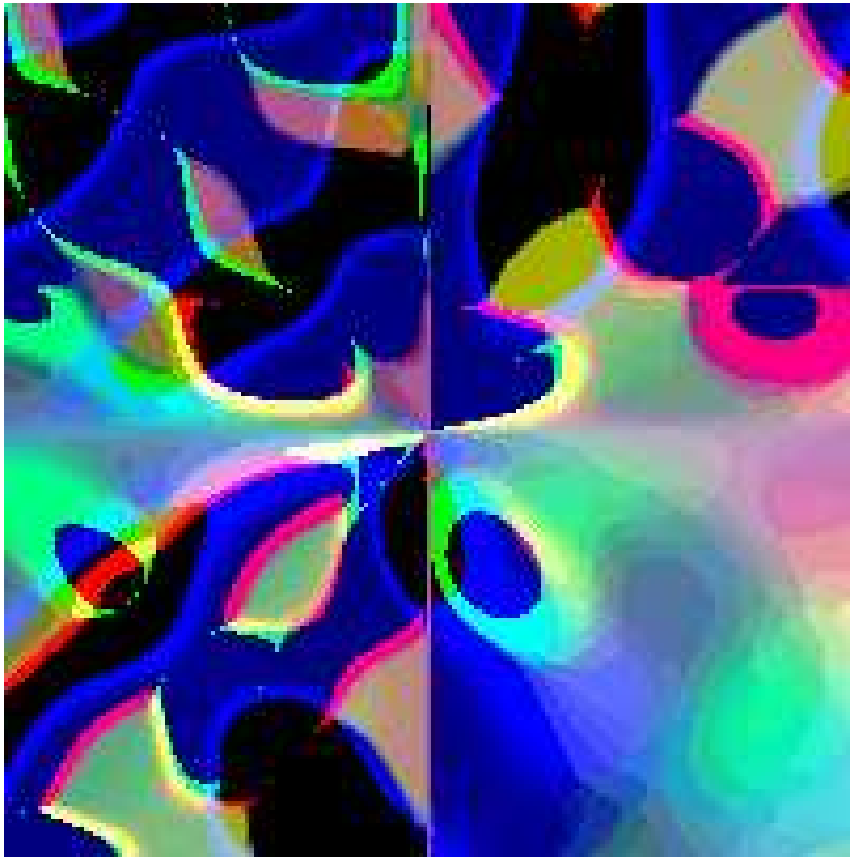
- inspired by Darwin's theory of evolution
  - survival of the fittest
- pioneered by John Holland in the 60s (see [1]).
- lots of applications, e.g. [2, 3]

Advantages GAs

- ease with which it can handle arbitrary kinds of constraints and objectives
  - only have to be able to compute them
  - don't even need to be able to express (as math)
- makes them highly applicable where
  - search space is complex or poorly understood
  - expert knowledge is difficult to encode to narrow the search space
  - mathematical analysis is not available

Used in many areas, even in art

- exemplar application: " I don't know much about art, but I know what I like"



http://www.geneticart.org/

Terminology

- living things are built using a plan described in our **chromosomes**
- chromosomes are strings of **DNA** and serve as a model for the whole organism
- a chromosome's DNA is grouped into blocks called **genes**, which have a location called a **locus**
- notionally, a gene codes for a particular trait
  - e.g. blue, or brown eyes
  - possible settings for a trait are called **alleles**
- a complete set of genetic material (all chromosomes) is called a **genotype**
- expression produces a **phenotype** (the organism) from the genotype

Biological evolution

- during reproduction, recombination occurs
  - in this context we call it **crossover**
  - genes from parents combine to give genes for offspring
- **mutation** also happens
  - it means that the elements of DNA are a little changed (randomly)
- fitness of an organism is measured by success of the organism in its reproduction
  - fitter organisms reproduce more, and so propagate their genes further
  - => the theory of evolution
  - all sorts of interesting variations here

Algorithm

1. **initialization:** create (randomly) an initial set of N solutions called the **population**, $\mathcal{P}$
2. **while** not finished
   (a) **evaluate fitness:** $f(x)$ of each $x \in \mathcal{P}$
   (b) **generate a new population:** the offspring
      i. **selection:** select two parents from population according to their fitness (better fitness makes them more likely to be selected)
      ii. **crossover:** With a crossover probability p cross over the parents to form new offspring, otherwise direct copy of the parents.
      iii. **mutation:** With a mutation probability q mutate new offspring at each locus
   (c) **replace old population:**

Comments

- very general
- every bit can be implemented in different ways
- key components
  - chromosome encoding
  - crossover method
  - mutation method
  - selection method
  - fitness criteria
- don't need explicit fitness function
  - could be the result of winners of a game
  - e.g. competition between population

Chromosome Encoding

- **Binary Encoding:**

  `Chromosome A  1101100100110110`

  - each bit represents some characteristic
    - e.g. z_e in budget constraint problem
  - string can represent a number: using Gray code

- **Permutation Encoding:**

  `Chromosome A  1 5 3 2 6 4 7 9 8`
  `Chromosome B  8 5 6 7 2 3 1 4 9`

  - each chromosome is a permutation

- **Value Encoding:** encode values directly

- **Tree Encoding:** used for programs

Gray Code [4, 5]

- represents each number in the sequence of integers $\{0...2^{N-1}\}$ as a binary string of length N
- in an order such that adjacent integers have Gray code representations that differ in only one bit position
- marching through the integer sequence therefore requires flipping just one bit at a time
- Example $N = 3$ (of a binary-reflected Gray code) The binary coding of $\{0...7\}$

numbers	0	1	2	3	4	5	6	7
binary coding	000	001	010	011	100	101	110	111
Gray coding	000	001	011	010	110	111	101	100

Crossover

- operate on selected genes from parents' chromosomes to create offspring's genes
- simplest way is single crossover point
  - randomly choose a **crossover point**
  - copy first chromosome up to the crossover point
  - copy second chromosome after the crossover point

Single crossover point example

```
Parent 1:   1101100100110110
Parent 2:   1111111000011110

Offspring:  1101111000011110
```

crossover

Crossover (continued)

There are other ways how to make crossover

- **multiple crossover points:** more than one random crossover point is chosen
- **random crossover:** randomly select genes from each parent
- **arithmetic crossover:** some arithmetic operation is performed to make a new offspring

Different types of crossovers work better for different problems.

Crossover for permutation encoding

- crossover for permutation coding is a little different
- Single point crossover
  - one crossover point is selected
  - copy from the first parent to the crossover
  - then the other parent is scanned and if the number is not yet in the offspring, it is added

Example:

```
Parent 1:   1 2 3 4 5 6 7 8 9
Parent 2:   4 5 3 6 8 9 7 2 1

Offspring:  1 2 3 4 5 6 8 9 7
```

crossover

Mutation

- intended to prevent all solutions in the population falling into a local optimum (so crossover can't escape)
- randomly changes the offspring
- binary encoding: switch a few randomly chosen bits

Bitwise mutation example

```
Original offspring:     1101100100110110
Mutated offspring:      1101000100111110
```

Mutation for permutation encoding

- as with crossover, lots of possibilities
  - if a group of bits encode a gene, we could mutate whole genes at each step
  - random mutation, but only allow solutions with increased fitness
- for permutation encoding, need different approach
  - e.g., swap a randomly chosen pair

Permutation mutation example

```
Original offspring:     1 2 3 4 5 6 8 9 7
Mutated offspring:      1 8 3 4 5 6 2 9 7
```

Selection algorithms

- **Roulette Wheel Selection:** select randomly based on fitness function. Probability of selection of x_i is

$$p_i = \frac{f(x_i)}{\sum_{i \in P} f(x_i)}$$

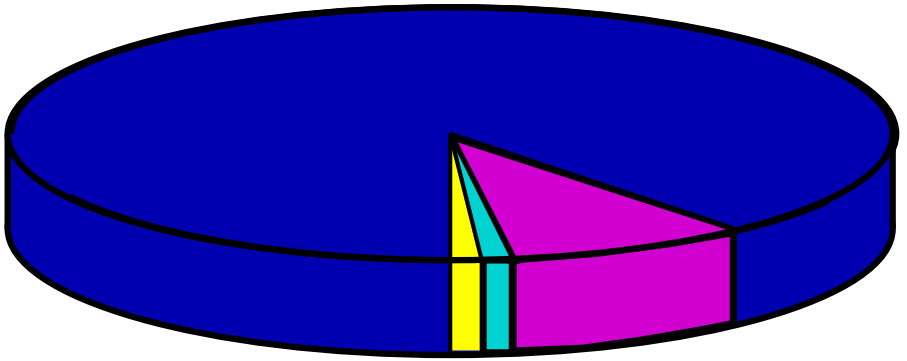
- **Rank Selection:** rank the population in order, so that $f(x_{(1)}) \leq f(x_{(2)}) \leq \cdots \leq f(x_{(N)})$. The probability of selection of $x_{(i)}$ is

$$p_i = \frac{i}{\sum_{i \in P} i} = \frac{2i}{N(N+1)}$$

- **Elitism:** we automatically keep the best one from each generation.

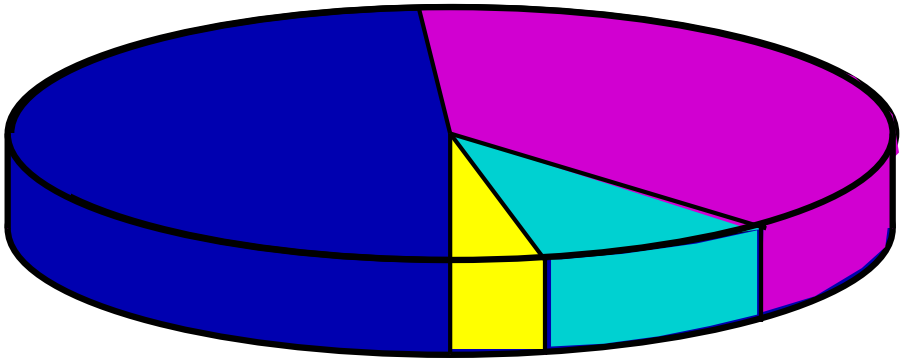
Roulette Wheel Selection

- Parents are selected according to their fitness
- The better the genotype is, the more chances it has to be selected
- Imagine a roulette wheel where all the genotypes in the population are placed.
- The size of the section in the roulette wheel for an individual is proportional to its fitness function
  - the bigger the value is, the larger the section is



Rank Selection

- Roulette Wheel Selection has problems when there are big differences in fitness values
  - one individual may completely dominate
  - other parents have little change to be selected
- Rank selection ranks the population from $1, \ldots, N$
  - selection with probability determined by ranking
  - worst will have probability $2/[N(N+1)]$
  - best will have probability $2N/[N(N+1)]$



Elitism

- create new population by crossover and mutation
  - parents don't appear in new population
  - likely that we will loose the best parent
- **elitism**
  - keep a few of the best current generation
  - rest of new population constructed as above
- elitism can rapidly increase the performance of *GA*
  - prevents a loss of the current best solution
  - algorithm never goes completely backwards

TSP example

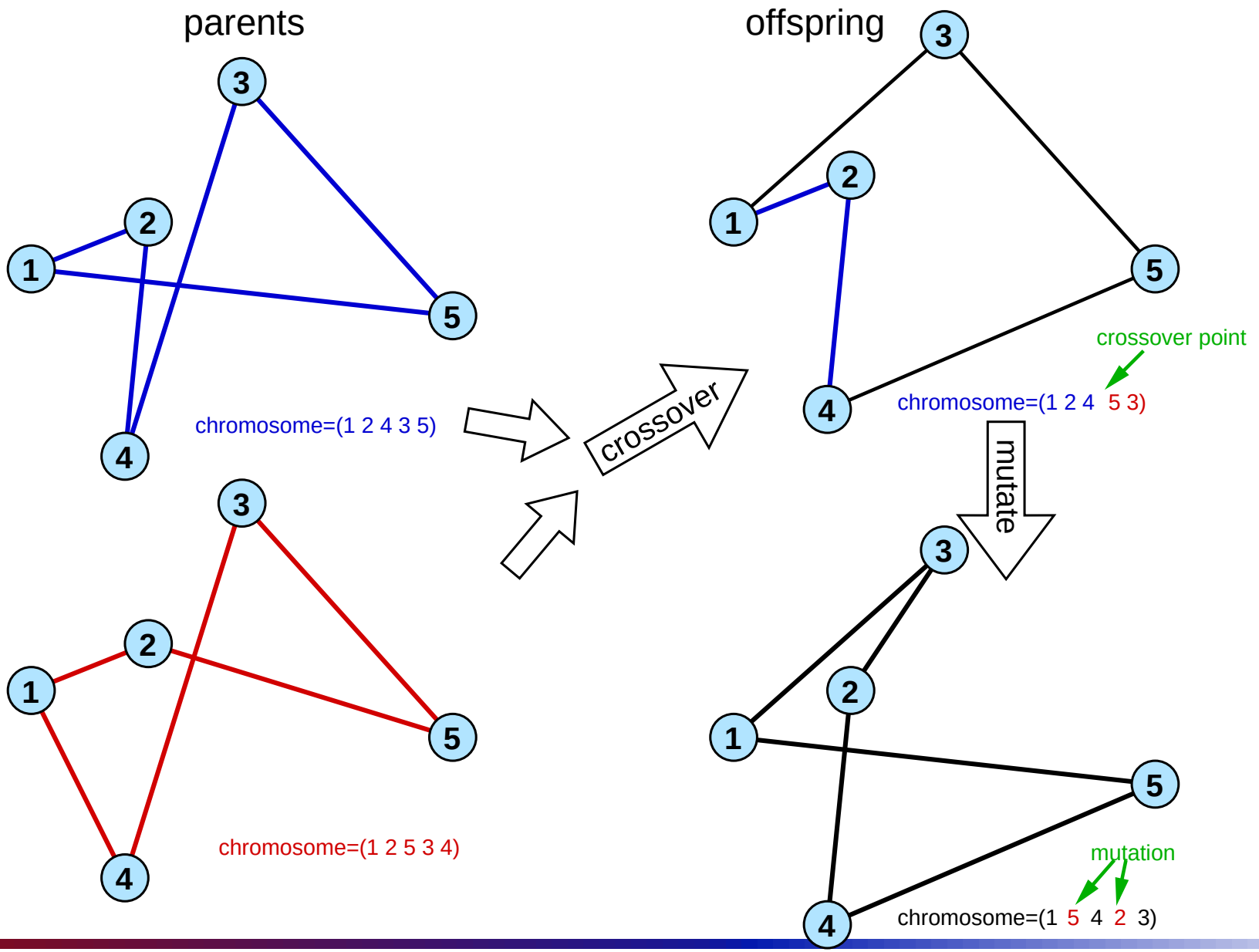
- we could encode by putting z_e into the chromosome
  - this doesn't include constraint that we visit each city once, in a circuit
  - we would have to include this constraint in the fitness function
  - much larger search space
- easier encoding is the permutation encoding
  - gives the order of the cities we visit
  - automatically includes the constraint
- if we have N cities, the chromosome has length N

TSP example

Many possible schemes

- Crossover
  - One point
  - Two point
  - None
- Mutation
  - Normal random - a few cities are chosen and exchanged
  - Random, only improving - a few cities are randomly chosen and exchanged only if they improve solution (increase fitness)
  - None - no mutation

TSP example


parents
offspring
chromosome=(1 2 4 3 5)
crossover
chromosome=(1 2 5 3 4)
crossover point
chromosome=(1 2 4 5 3)
mutate
mutation
chromosome=(1 5 4 2 3)

TSP example

Web applet illustration:

http://www-cse.uta.edu/%7Ecook/ai1/lectures/applets/gatsp/TSP.html

Parameters of GAs

- crossover probability
- mutation probability
- population size

Parameters: crossover probability

- If there is no crossover, offspring are exact copies of parents
  - but this doesn't mean the population is the same
- If there is crossover, offspring are made from parts of both parent's genotype (often just one chromosome)
- Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosomes will be better. However, it is good to leave some part of old population survive to next generation.
- Crossover rate should be high generally, about 80%-95% (though it can vary)

Parameters: mutation probability

- if there 0% mutation, offspring are generated immediately after crossover
- if mutation probability is 100%, whole chromosome is changed
- mutation prevents the GA from falling into local extrema
  - similar to simulated annealing
- mutation should not occur very often, because then GA will just be a random search.
  - mutation rate should be very low. Best rates seems to be about 0.5%-1%

Parameters: population size

- too small a population there are
  - few possibilities to perform crossovers
  - too small part of search space covered
- too large a population
  - GA slows down
  - at some point hit diminishing returns
- Good population size is about 20-30, however sometimes sizes 50-100 are reported as the best
  - Some research also shows, that the best population size depends on the size of chromosomes, e.g. for chromosomes with 32 bits, the population should be higher than for chromosomes with 16 bits.

Note

- even though simulated annealing and genetic algorithms are called random algorithms they are not completely random
- it's not just randomly testing solutions
- we use a stochastic process
- however the result is highly non-random

Pretty example movies

GAs have often been used in generating artificial life

- Karl Sims

`http://www.genarts.com/karl/evolved-virtual-creatures.html`

`http://alife.ccp14.ac.uk/ftp-mirror/alife/zooland/pub/research/ci/Alife/karl-sims/`

Example of evolved artificial life

- Torsten Rei: realistic animations of stick figures, by adding "muscles" to them, and using distance walked as fitness.

`http://cognews.com/1060458741/index_html`

Example of evolved artificial life

Example of evolved artificial life

Example of evolved artificial life

- techniques like these used in LoTR animations

Other randomized algorithms

There are other randomized algorithms

- **ants:** metaphor is a colony of ants (simple agents) running simple rules, to achieve highly organized collective behaviour (also called Swarm Intelligence)

  `http://www.merlotti.com/EngHome/Computing/AntsSim/ants.htm`
  `http://www.codeproject.com/cpp/GeneticandAntAlgorithms.asp`

- **tabu search:** iteratively try to find solutions to the problem, but to keep a short list of previously found solutions and to avoid 're-finding' those solutions in subsequent iterations. Basically, if you try a solution, it becomes tabu in future tries [6].

References

[1] J. H. Holland, Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press, 1975.

[2] J. J. Grenfenstette, ed., Proceedings of the First International Conference on Genetic Algorithms and Their Applications. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1985.

[3] J. D. Schaffer, ed., 'Proceedings of the Third international Conference on Genetic Algorithms. Morgan Kaufmann Publishers, Inc., 1989.

[4] F. Gray, "Pulse code communication." U. S. Patent 2 632 058, March 17 1953.

[5] M. Gardner, "Mathematical games," Scientific American, p. 106, August 1972.

[6] F. Glover, "Tabu search: A tutorial," Interfaces, vol. 20, no. 4, pp. 74-94, 1990.