PG4200: Algorithms And Data Structures

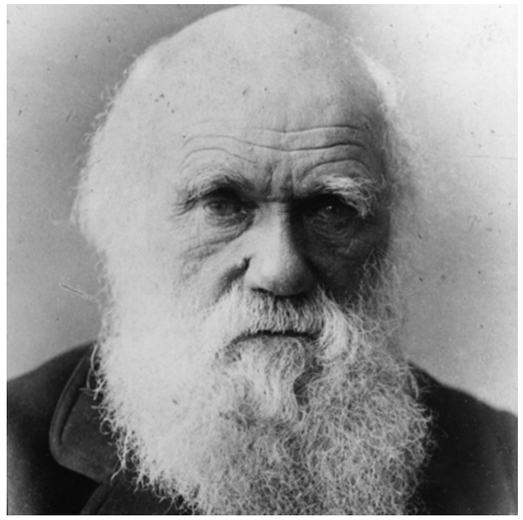
Lesson 11:
Genetic Algorithms and Randomness

Dr. Andrea Arcuri

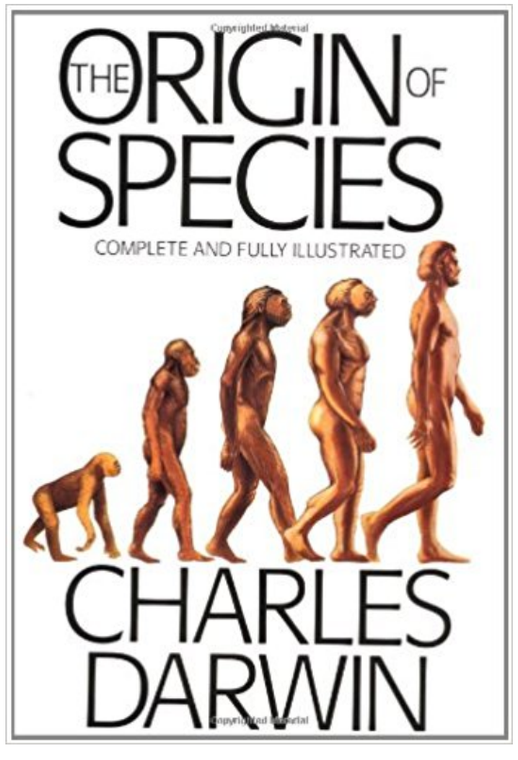
Nature Inspired Algorithms

- Nature is good at solving many different problems
- Idea: get inspiration from natural phenomena to create effective optimization algorithms
- E.g., carbon-to-diamond process: high temperature in Earth's mantle, cooled slowly while raising up to surface
  - Simulated Annealing Algorithm
- E.g., behavior of ants seeking a path between their colony and a source of food, based on pheromone trails
  - Ant Colony Optimization Algorithm

Theory Evolution



- Charles Darwin, *"The Origin of Species"*, 1859



- Theory describing how different species *evolved* from unicellular organisms

Evolutionary Algorithms (EAs)

- EAs are optimization algorithms based on the theory of evolution
- **(1+1) EA**: the simplest EA
- **Genetic Algorithms**: the most popular EA
- But there are more…

(1+1) EA

- Mimic the evolution of a single individual that reproduces asexually, giving birth to only one offspring
  - Ie, 1 individual that produces 1 offspring
- Evolution is driven by mutations in the chromosome of the offspring
- Kill the offspring if worse than parent, otherwise kill parent after birth
  - Yep, evolution can be cruel…

Representation

- The solutions in your problem space are the *individuals* you want to evolve
  - Eg queen positions on board
- Chromosome: the actual representation of the individual
  - Eg, binary array 0/1 for knapsack problem
- Mutation: similar to neighborhood in Hill Climbing, it samples new *similar* individuals

Mutation in 0/1 Sequence

1	1	0	0	0	1	0	1	0	0

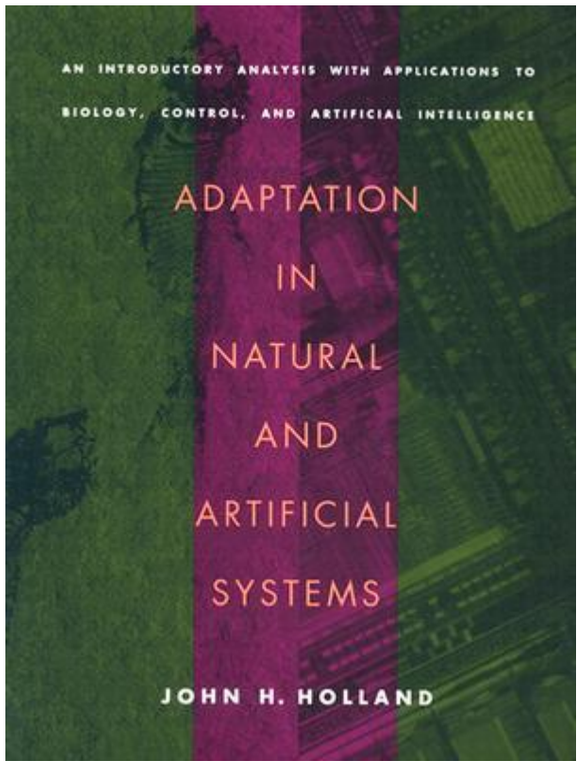
- Mutation: given N bits, each bit can be flipped with probability 1/N
  - Eg 10% probability in the above example
- *On average,* only 1 bit is flipped per mutation operation
- *Non-zero* probability to jump to any solution in the search space with a *single* mutation operator
  - ie, all bits can be flipped, although with low probability (1/N)^N
- Reason: small modifications on average, but allow larger jumps to escape from local optima

Genetic Algorithms (GAs)

- 1950s: Turing proposed use of evolution in computer programs

- 1970s: John Holland created GAs to address optimization problems

- Simulate evolution of an entire *population* of individuals, which procreate sexually



Population

- Keep track of K individuals
  - Eg, stored in an array
- At the end of search, return the best solution in the population

Generation

- The search will be composed of 1 or more *generations*
- At each generation, select individuals for reproduction
- Create a new generation of same size K
- Kill the previous generation
  - Yes, evolution is really cruel…
- Note: given same amount of fitness evaluations Z or time budget, a larger population K means less generations G, ie Z = K * G

Selection

- The fittest individuals will have higher chances of reproduction

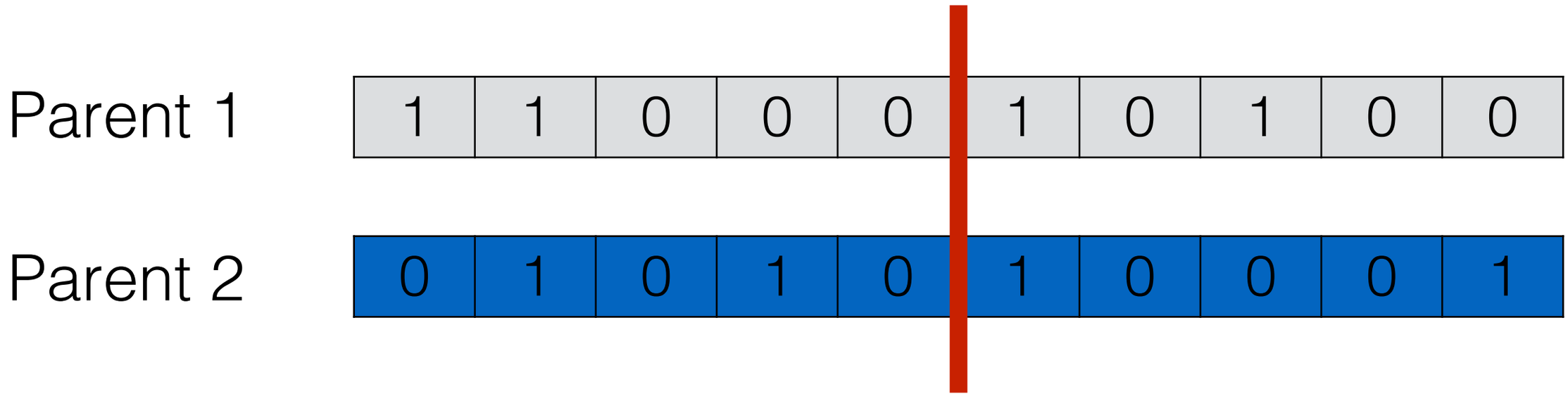
- Different strategies for parent selection

- Tournament Selection: sample T individuals *randomly* from the population, and choose the best among them (according to the fitness function)

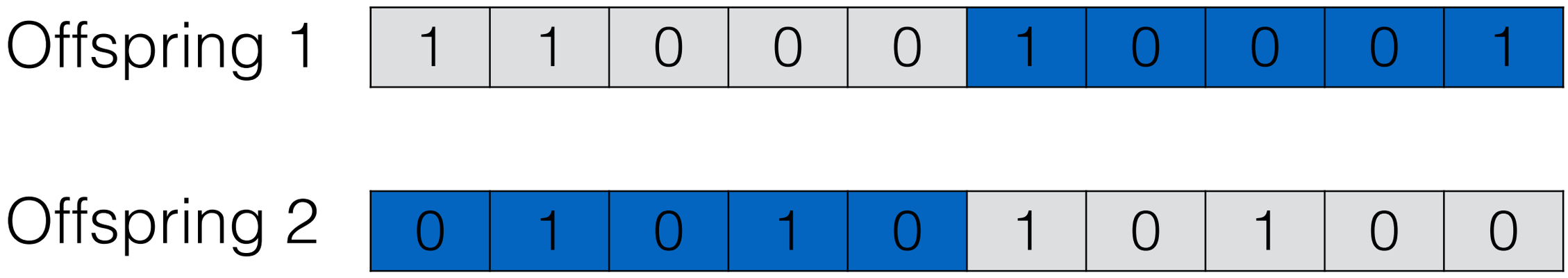
Sexual Reproduction

- Select 2 parents, which give birth to 2 offspring
  - Note, ignoring gender here…
- Offspring will share genetic material with their parents, via the *crossover* operation (aka xover)
- After xover, offspring still have chances of getting mutated

Crossover



Choose a splitting point, eg the middle of the chromosomes



Xover Issues

- In some domains, there are constraints in the representation
  - Eg, recall example of queen positions
- A xover could lead to invalid offspring
- Two options:
  1. Kill them
  2. Try to repair them

Elitism

- You do not want to lose your best individuals
  - Unlucky in (tournament) selection
  - Negative mutations

- At each generation, copy the best B individuals over the next generation, without xover or mutation

- Even in evolution, some elite individuals still cheat the system and get preferential treatment…

Does this “crazy” stuff actually work???

Usage

- Genetic Algorithms are among the most used kind of optimization algorithms
- Successfully used in a lot, a lot of different domains
- Goldberg's 1989 book on GAs is among the most cited books in the *whole* field of Computer Science

Example

- Evolution of antenna designs
  - Optimize beamwidth and impedance bandwidth
- Best found with an Evolutionary Algorithm
- Used for example in NASA spacecrafts

More…

- There is of course much more… we just scratched the surface
- Tradeoff *exploration* vs *exploitation* of search landscape
- Landscape Analysis
- Parameter tuning
- Multi-Objective Optimization of conflicting goals
- *Genotype* vs *Phenotype*
- Population diversity
- Etc. etc.

Which Algorithm to Use?

- *Due to NFL, it depends on the problem*

- **(1+1)EA** is an easy to write algorithm, which gives good performance on many different kinds of real-world problems

- If you want to use some existing library for Java, look at **ECJ** and **jMetal**

Randomness

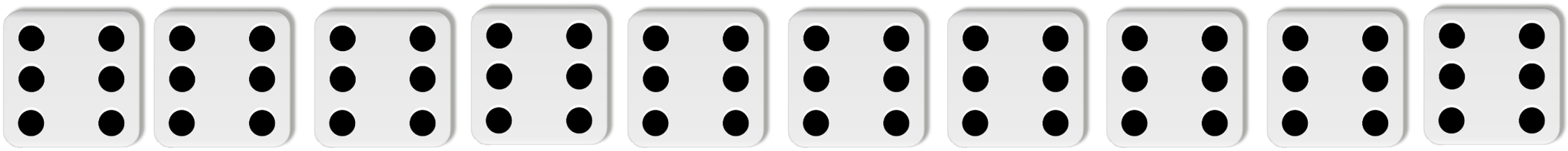
Definition

- Randomness is the lack of pattern or predictability in a series of events (eg, rolling a dice)
- Eg, if you rolling a dice three times, and get 4, 2, and 5, then that should give you no info about what will be value of the 4^{th} roll
- Given info of past events, should have no info to help predicting future events
  - Ie, events in a sequence are fully *independent*

Why You Need Randomness?

- Video games
- Optimization algorithms
  - Eg, recall random restarts in Hill Climbing
- Security algorithms
- Scientific applications
- Etc.

A 6 ten times in a row???



- If dice is *fair*, getting a specific value would have probability 1/6

- Getting a 6 ten times would have probability (1/6)^10
  - Low probability, but not impossible

- Could such sequence of 10 events come from a random generator?
  - In theory it can be, as such sequence is possible

Improbable Events Can Still Happen

How to Get Random Numbers in Java?

- **java.util.Random**
  - Fast, but not particularly good. However, fine for most basic applications

- **java.security.SecureRandom**
  - Slower, but much better. Can be used in security contexts

Random Sources

- How to create a random sequence? (eg, of integers)
- Natural phenomena
  - Atmospheric noise, thermal noise, electromagnetic and quantum phenomena, etc.
  - Expensive to collect such info
- Pseudo-Random Generators
  - Algorithms that create sequences that look like random
  - Not truly random, but can be *enough* random for your needs

Array Implementation

7	2	4	1	1	3	7	1	4	9

- Create manually an array with random-like values
- Keep and index of current value
- When client asks for random value, return the one in index position, and increase the index by 1
- When reaching end of array, start from beginning

Seed

- In previous random generator, first queried value is always 7

- *Seed:* starting point of the index

- Eg, “index = seed % N”
  - Where N is the size of the array
  - Seed could be chosen based on current time, eg given by the CPU clock

Issues

- Calling 30 times, starting from seed 2:
  4,1,1,3,7,1,4,9,**(7)**,2,4,1,1,3,7,1,4,9,**(7)**,2,4,1,1,3,7,1,4,9,**(7)**,2

- Although the first N elements might look random, such sequence of N elements *repeats itself*

- Given the knowledge of a sampled value, I can predict what would be next
  - From a 2 always follows a 4
  - From a 7, either a 1 or a 2 might follow

- Using an array with a large enough N would take a lot of space…

Linear Congruential Generator (LCG)

- The algorithm used in *java.util.Random*
- $X_{i+1} = (aX_i + c)\ \%\ m$
- This recursive function provides a new “random” value X_{i+1} from a previous X_i, where X_0 is the initial seed
- Care needs to be taken to choose good “*a”* and “*c”* values, as to get longest *period P* as possible
- In such sequence, values from 0 to *m-1* can be sampled
- Think about it as an array with length *P,* but without the memory overhead

LCG in Java

- The default period in Random in Java is $P = 2^{48}$
- It might look long, but it has many shortcomings
- An integer has 32 bits, so there are 2^{64} possible pairs of integers… which cannot all be present in a sequence of $P = 2^{48}$ values
  - Ie given current sampled Y, there are some values that are impossible to get in the next call to the random generator
  - Just by observing 3 values in sequence, can efficiently predict all the ones that follow, even if you do not know "*a*", "*b*" and "*m*"

More Advanced Algorithms

- Many algorithms
  - Usually, the better quality of sequence, the slower the algorithms
- Eg, Mersenne Twister (MT) has $P = 2^{19937} - 1$
- Longer period is good, but not sufficient
  - Eg, MT, in its base form, is not cryptographically secure, as just need to see 624 values to predict the whole sequence

Random Bits

- When getting a random number, if randomness is critical, you need to use **all** of its bits

- A random generator usually does not give you guarantees on the properties of subsets of its bits in output

- So, let's say you want a number in 0-99, and so use "*random() % 100*" (where *random* gives a random positive integer)… what's the problem with this?
  - Only the 7 leftmost bits out of 32 are used

Cont.

321,807
7
11,113,207
999,999,907
2,307
452,223,207
137,807
98,880,907
707
7,773,807

- Those 10 elements are all unique, when consider all their 32 bits
- Doing "*random() % 1000*" would give the sequence:
  807,7,207,907,307,207,807,907,707,807
- But doing "*random() % 100*" would give the sequence: 7,7,7,7,7,7,7,7,7,7

Homework

- No Book Chapter
- Study code in the *org.pg4200.les11* package
- Do exercises in *exercises/ex11*