Evolutionary Computation

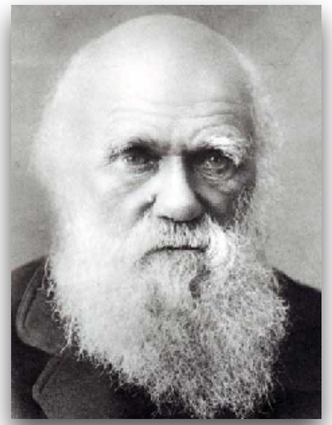
CS454 AI-Based Software Engineering

Shin Yoo

Darwinian Evolution

- An attempt to theorise the emergence of new species.
- It should be noted that Alfred Wallace independently arrived at a very similar conclusion at the same time. Wallace's paper prompted Darwin to publish "On the Origin of Species".
- Note that contemporary views are now very different.

Fig. 24.—Skulls of Pigeons viewed laterally, of natural size. A. Wild Rock-pigeon, *Columba livia*. B. Short-faced Tumbler. C. English Carrier. D. Bagadotten Carrier.

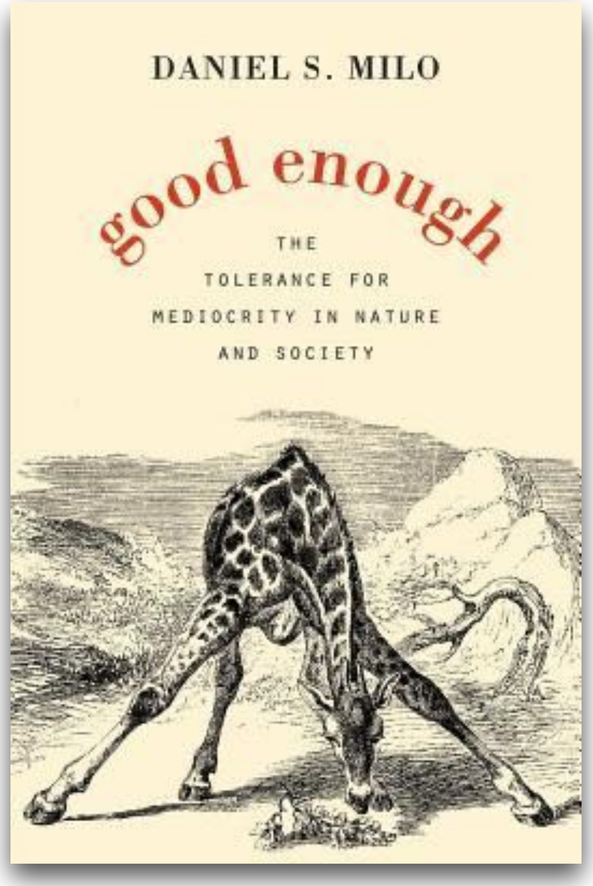


What is it exactly?

- If all offspring survived to reproduce the population would grow (fact).
- Despite periodic fluctuations, populations remain roughly the same size (fact).
- Resources are limited and are relatively stable over time (fact).
- A struggle for survival ensues (inference).
- Individuals in a population vary significantly from one another (fact).
- Much of this variation is heritable (fact).
- Individuals less suited to the environment are less likely to survive and less likely to reproduce; individuals more suited to the environment are more likely to survive and more likely to reproduce and leave their heritable traits to future generations, which produces the process of **natural selection** (inference).
- This slowly effected process results in populations changing to adapt to their environments, and ultimately, these variations accumulate over time to form new species (inference).

Is it really the “survival of the fittest”?

- Nature neither optimises nor has any intention.
- If you are “good enough”, you survive. Or, sometimes, a series of random events can result in genetic drift.
- A highly recommended reading: “Good Enough: The Tolerance for Mediocrity in Nature and Society” by Daniel S. Milo
- Intentional “optimisation” via evolution is a purely artificial concept, and is separate from what takes place in nature.



Evolutionary Computation

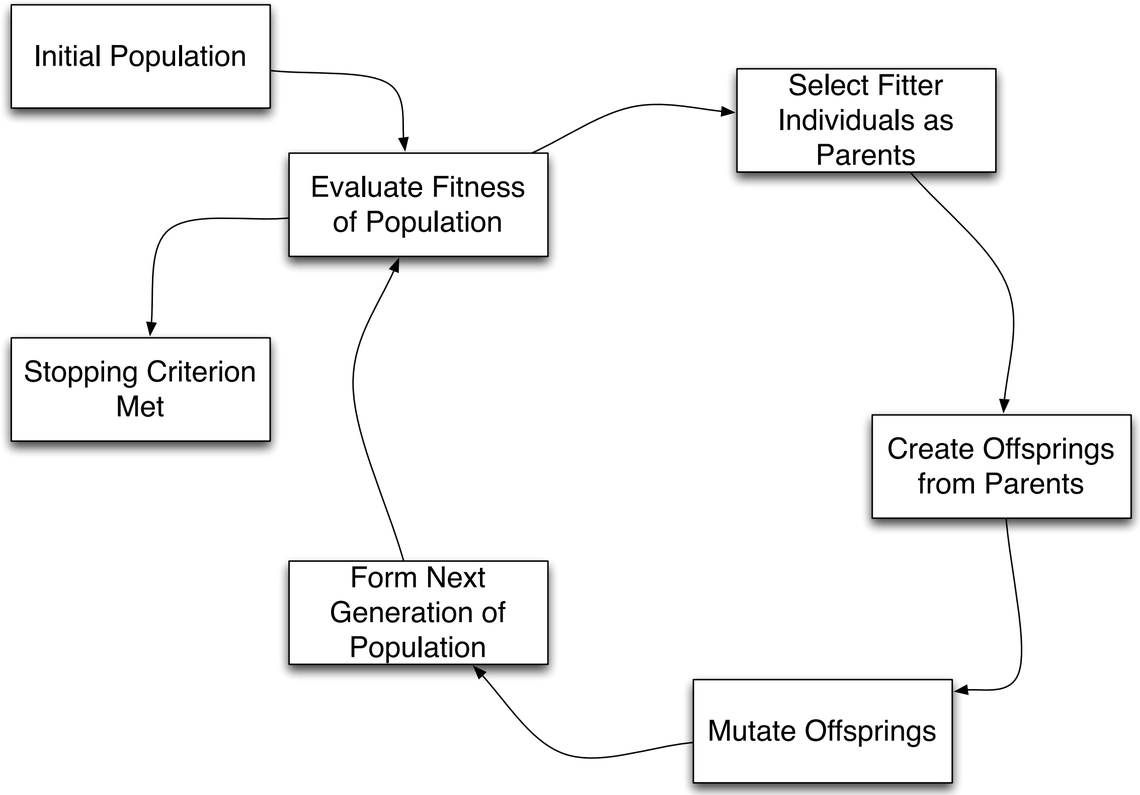
- A broad idea that 1) repeated random-ish sampling 2) guided by the degree of “how fit” candidate solutions can 3) gradually improve the solution quality, eventually producing a good enough solution for your problem.
- We simulate the Darwinian evolution with a randomised algorithm:
  - **Genotype**: parts of genetic material that determines a specific characteristic of an individual —> representations of candidate solutions
  - **Phenotype**: the actual characteristic that is the manifestation of a specific genotype —> the measure of how good
- We try to manipulate genotype, hoping for better phenotype.

Evolutionary Computation

- Compared to local search algorithms we have previously seen, Evolutionary Computation search (e.g., genetic algorithms) is considered as “Global Search”.
  - You do not travel through connected neighbourhood in the solution space: rather, your population are scattered throughout the space.
  - There is no gradual movement: rather, you create (=sample) solutions by following the Darwinian evolution analogy (we will see the details later).

The Evolutionary Computation Loop

- There are many EC based algorithms.
- They are typically population-based, i.e., they maintain a pool of candidate solutions.
- At each iteration, we form the **next** generation, by manipulating the current generation in various ways.



Initial Population

- Usually initialised randomly: this introduces the **variance** among individuals.
- We mean phenotype variance. Depending on problems, genotype variance may not always result in phenotype variance.

Selection Operators

- We apply selection operators to the population, to choose two parent individuals.
- This is one of two places where we apply **evolutionary pressure**: the fitter you are, the more successful you are in terms of reproduction.
- As long as you keep the evolutionary pressure, any selection is possible.
  - Fitness Proportional Selection, Rank-Based Selection
  - Roulette Wheel Sampling, Stochastic Universal Sampling
  - Tournament Selection

Likelihood of Being Selected

- **Fitness Proportional Selection**: probability of selecting an individual is directly proportional to its absolute fitness over the entire population, i.e., $P_{FPS}(i) = \frac{f_i}{\sum_{j=1}^{\mu} f_j}$

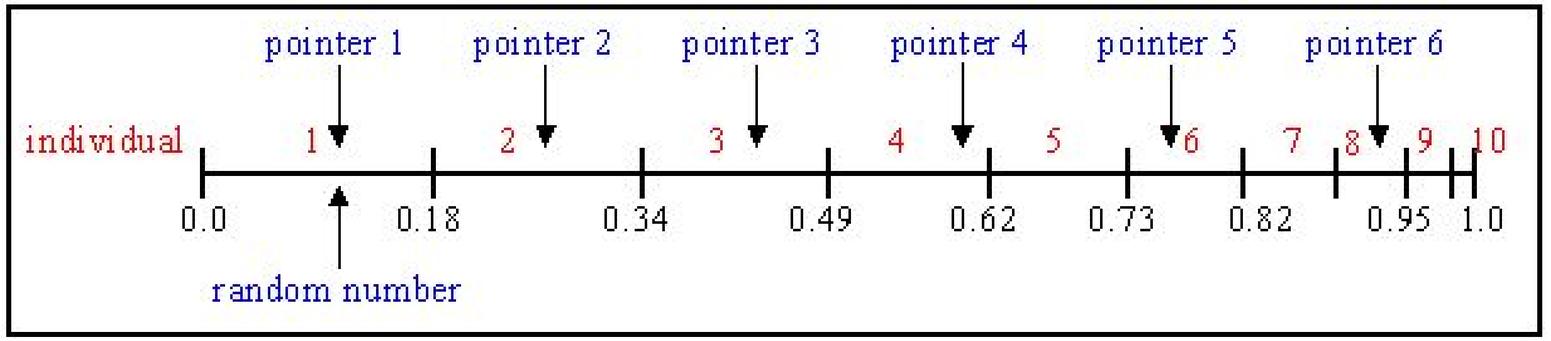
- **Rank-Based Selection:** given μ individuals, the best gets the rank of $\mu - 1$, and the worst 0. Then:
  - Linear: $P_{linear}(i) = \frac{2 - s}{\mu} + \frac{r_i(s-1)}{\sum_{j=1}^{\mu} r_j}, 1 \leq s \leq 2)$
  - Exponential: $P_{exp}(i) = \frac{1 - e^{-r_i}}{\sum_{j=1}^{\mu} 1 - e^{-r_j}}$

Smoothing Hyperparameter

(You can also shift all fitness values, or scale using standard deviation, to smooth probability distribution)

How to actually sample using probabilities

- **Roulette Wheel Sampling**: a series of independent sampling, as in a roulette wheel
- **Stochastic Universal Sampling**: similar to roulette wheel sampling, but with N arms when you want to sample N individuals



Tournament selection

- What if fitnesses cannot be measured on an absolute scale?
  - e.g. On evolving game strategies, fitnesses of two strategies can be evaluated only by playing against each other.
- Or if computing selection probabilities is impossible?
  - e.g. On a distributed setting, acquiring global knowledge of the fitnesses may not be possible.
- Select k random individuals, and only pick the best of them. The higher the k is, the higher the selection pressure is. Simples!

Crossover Operators

- Offsprings inherit genes from their parents, but not in identical forms.
- Think Mendelian recombination of alleles; since we don't have alleles, we actually recombine the whole genotype.

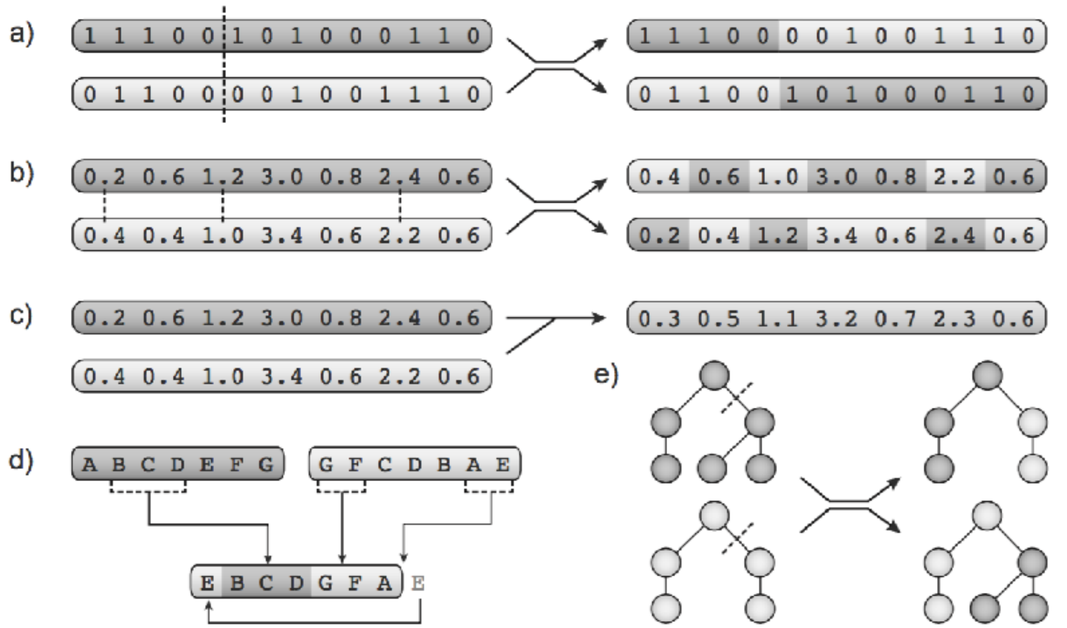


Figure 1.11 Examples of crossover operators. *a*) one-point; *b*) uniform; *c*) arithmetic; *d*) for sequences; *e*) for trees.

Mutation Operators

- This is, usually, the **only** way **new genetic material** is introduced into the population; without mutation, all we do is recombining the initial population (which was randomly generated).
- Small, local modifications to genotypes, such as:
  - single bit-flip / adding/subtracting small amount to integers / swapping two elements in permutations / replacing one node in a tree with a different, compatible type

Generational Selection

AKA who gets to remain in the population?

- Generational Replacement: the offsprings become the new current population (no parent survives)
- Elitism: maintain M best individuals from the parents' generation (reasons: noisy fitness, too strong mutations, too complicated search space…)
- Gradual Replacement: replace M worst individuals from the parents' generation with M best individuals from the offsprings.

Stopping Criterion

- Deciding one can be hard: these are stochastic algorithms, and you don’t know what the global optimum is.
- In reality, one of the following two:
  - Fixed number of fitness evaluations, or
  - When a good enough solution has been found
- At the end, report either:
  - The best solution in the last generation population, or
  - The best solution observed so far (you need to maintain “archive”)

Suppose we break a 6 digit numeric password with GA

- Let’s assume that we have a tool that tells us how many digits are correct

Password: 893714

193562

243690

123456

121214

Randomly Generated Initial Population

Suppose we break a 6 digit numeric password with GA

- Let’s assume that we have a tool that tells us how many digits are correct

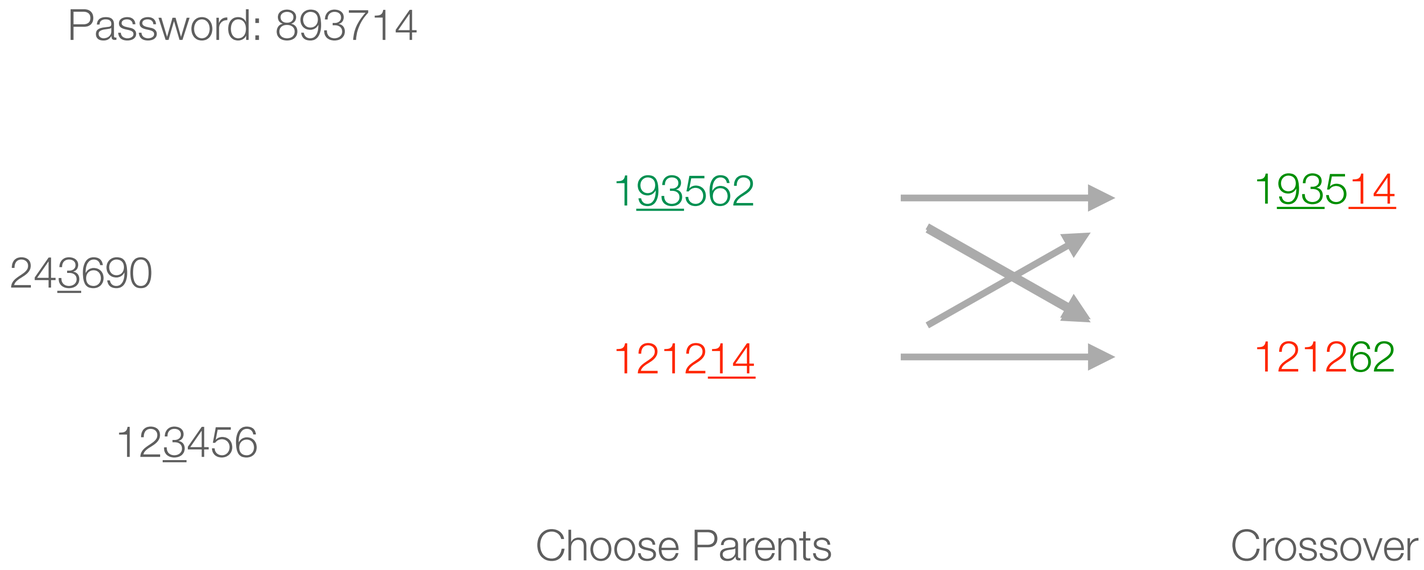
Password: 893714

193562	Score: 2
121214	Score: 2
243690	Score: 1
123456	Score: 1

Evaluation

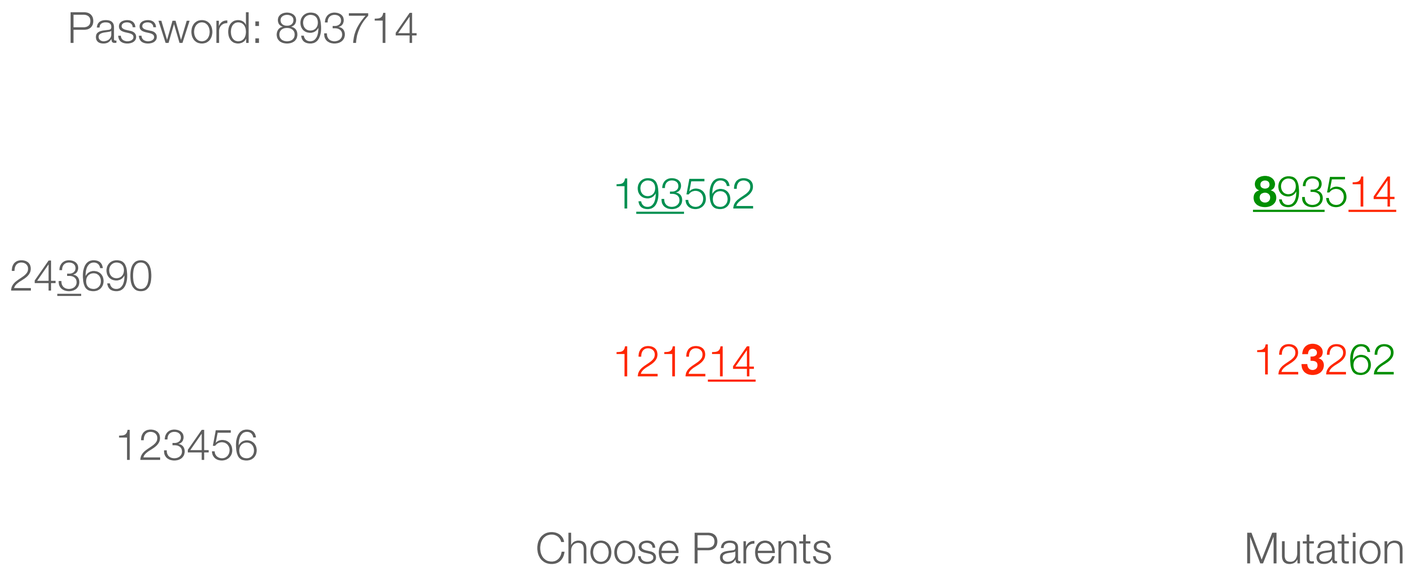
Suppose we break a 6 digit numeric password with GA

- Let’s assume that we have a tool that tells us how many digits are correct



Suppose we break a 6 digit numeric password with GA

- Let’s assume that we have a tool that tells us how many digits are correct



Genetic Programming (GP)

We do GA, but our candidate solutions are programs!

- The essence of evolutionary computation remains the same.
- Representation of candidate solutions need to be structured enough to contain programs (we typically use trees)
- Fitness of candidate solutions depends on how well the candidate programs run against a given set of inputs
  - This makes GP almost machine learning - in some sense, we are training (=evolving) a model (=program) that fits (=processes) the given training data (=given input set) as best as possible.

Programs, not numbers

- Programs are highly structured.
- GP mostly (but not exclusively) uses syntax tree to represent solutions.
  - `max(x + x, x + 3 * y) = (max (+ x x) (+ x ( * 3 y)))`
- Obviously, it is easier to implement GP with some languages: high-level ADT, garbage collection, etc

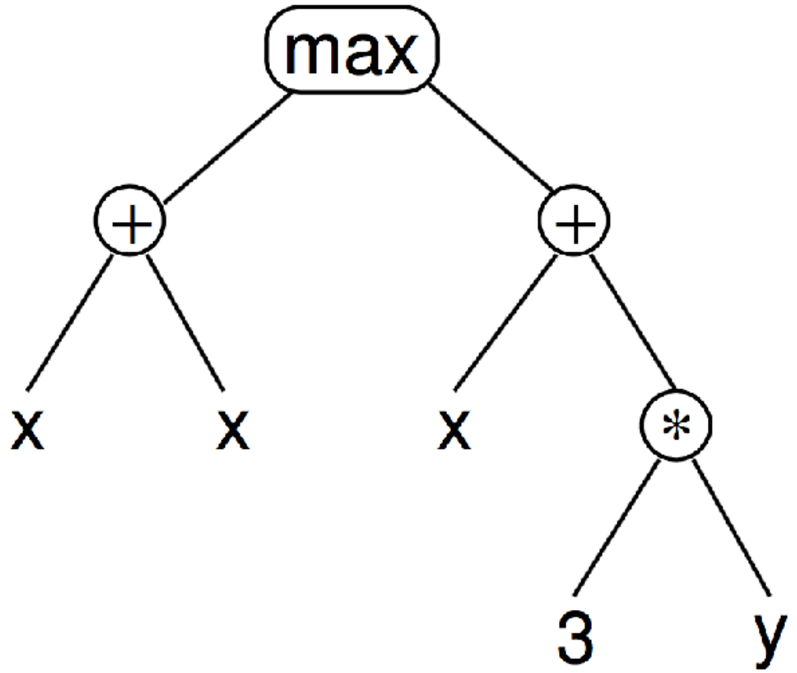
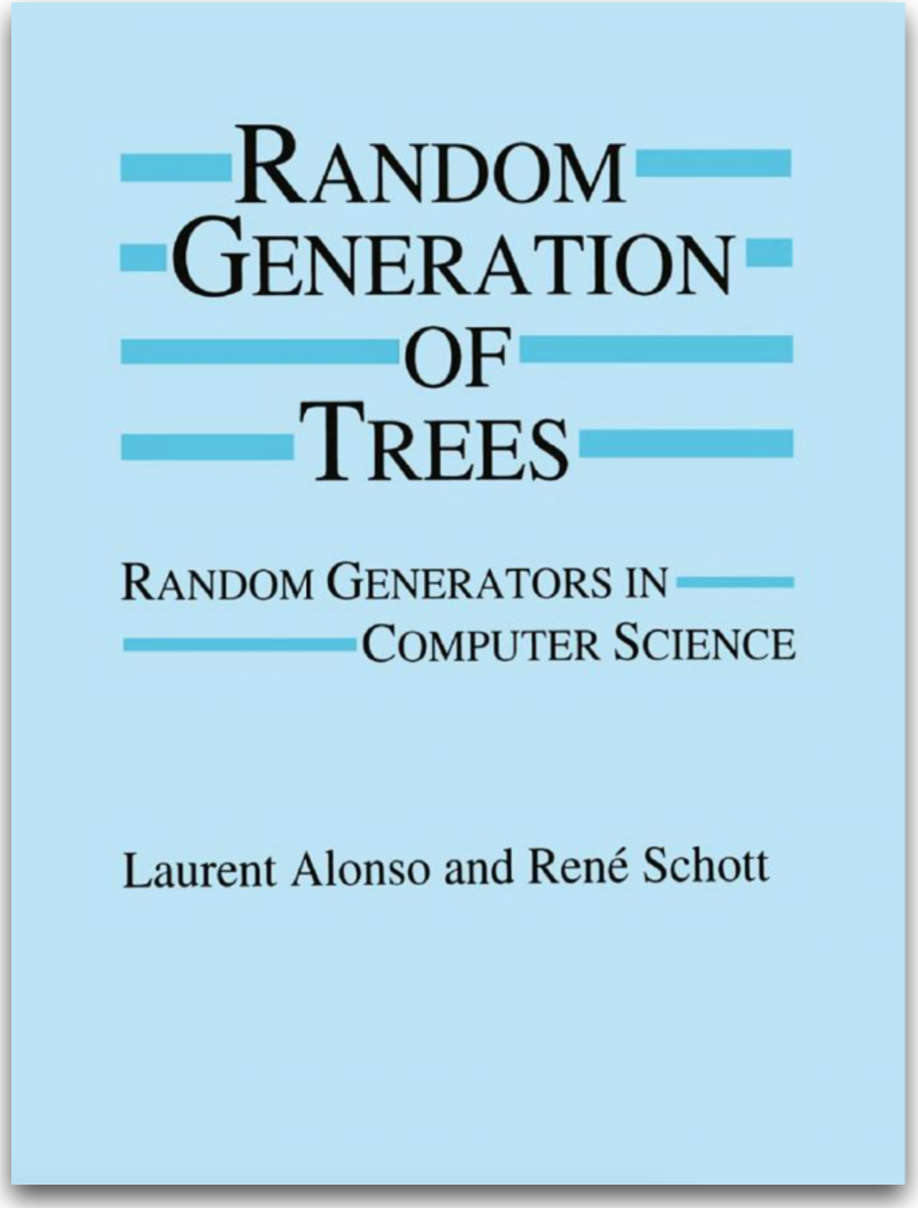


Figure 2.1: GP syntax tree representing `max(x+x,x+3*y)`.

Initialisation

- What is a random tree?
- We need to limit the size of the tree (i.e. depth): we do not want arbitrarily large trees as solutions.
- Many initialisation methods: full, grow, ramped half-and-half, and others.



Full Initialisation

- Grow full trees
  - Add non-terminal nodes only until the depth limit is reached.
  - Then only add terminals as leaves.
  - All trees are fully grown.

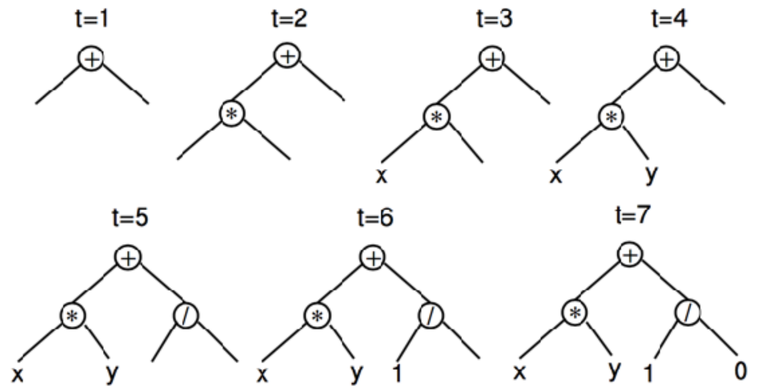


Figure 2.3: Creation of a full tree having maximum depth 2 using the `full` initialisation method (t = time).

Grow Initialisation

- Grow various trees
  - Add any node while there are empty slots and the depth limit is not reached.
  - Results in trees of various sizes, but the ratio between terminals and non-terminals will bias the average size.

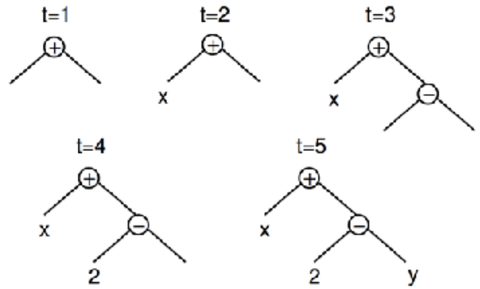


Figure 2.4: Creation of a five node tree using the `grow` initialisation method with a maximum depth of 2 (t = time). A terminal is chosen at $t = 2$, causing the left branch of the root to be closed at that point even though the maximum depth had not been reached.

Ramped Half and Half

- Half of population is initialised with Full method
- Half of population is initialised with Grow method
- A better diversity in terms of shapes and size

Crossover

- Initial idea
  - Randomly choose two crossover points in parent trees;
  - Cut and swap subtrees below the crossover points.

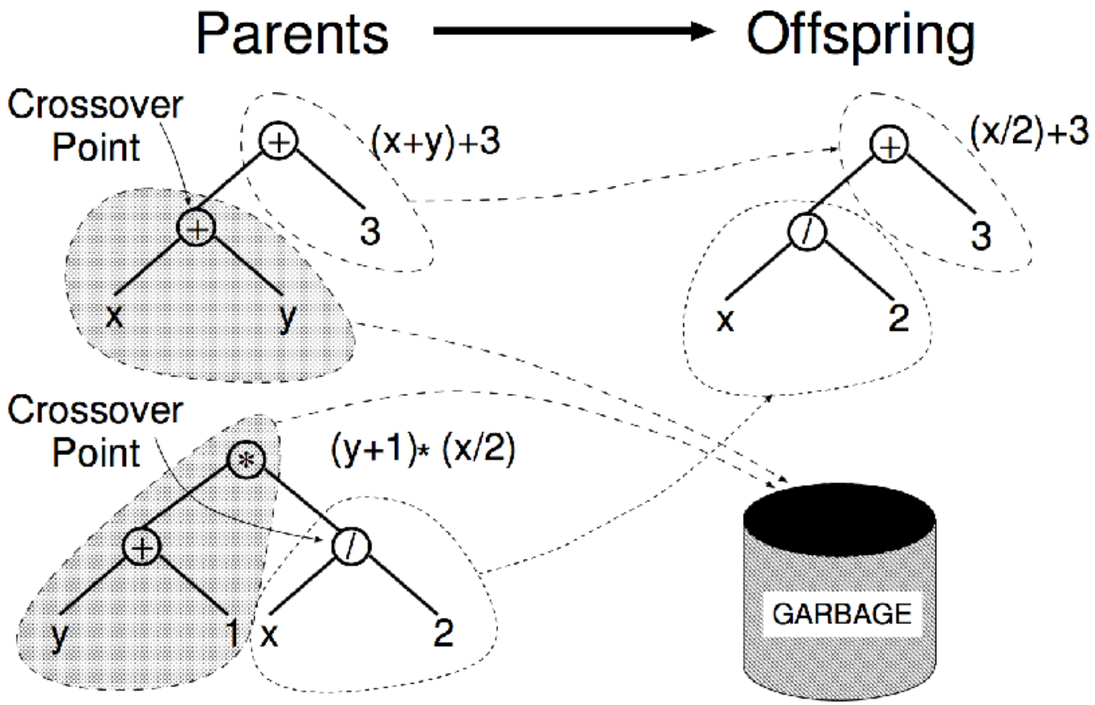


Figure 2.5: Example of subtree crossover. Note that the trees on the left are actually *copies* of the parents. So, their genetic material can freely be used without altering the original individuals.

Crossover

- Often crossover points are NOT sampled with uniform random distribution:
  - Average branching factor is 2 or more, which means the majority of the nodes are leaves, which means the majority of branches will simply cut a single leaf.
  - Type-aware crossover (Koza 1992): 90% chance of choosing a non-terminal node, 10% chance of choosing a terminal node
  - Size-fair crossover: First crossover point in one parent chosen randomly, then the subtree from the other parent should be constrained not to be bigger than this

Mutation

Anything that introduce random changes within a subtree

- Subtree mutation (a.k.a. headless chicken mutation): choose a subtree, and replace it with a randomly generated subtree.
- Point mutation: with a certain probability, replace the node with another node of same arity.
- Hoist mutation: create a new individual, which is a randomly chosen subtree of the parent.
- Shrink mutation: replace a randomly chosen subtree with a randomly chosen terminal node.
- Permutation mutation: change the order of function arguments in trees.
- Systematic constant mutation: use external optimisation to tune the constants in the expression tree.

Type Closure

- Non-terminals need to be type consistent: it is necessary that any subtree can be used in any argument position of any function in the set.
  - Rewriting may do. For example, `if(boolean, expr_true, expr_false)` can be rewritten as `if(expr1, expr2, expr_true, expr_false)` where the predicate is always fixed as `expr1 < expr2`
  - You will still get all possible comparisons, with the appropriate mutation.
- Strongly-typed GP: all mutation operators should abide by type rules!

Safety & Sufficiency

- Runtime safety: some combinations of nodes will fail at runtime, and need to be re-written to be fail-safe.
  - 1 / x → 1 if x == 0 else 1 / x
  - Overflows are trickier to deal with.
- Sufficiency: do I have all the building blocks to evolve the program I want? This is a much harder problem and there is no generic answer. However, arguably GP's aim is to approximate the target function, and not to find the exact function.

Why (or when) does it work?

- Schema Theory (John Holland, 1975): given genotypes of k symbols with length l, the schemata set is $\{s_1, \ldots, s_k, * \}$ where * means “don’t care”. There are $(k+1)^l$ schemas.
  - Intuitively, schemas can be thought of as non-consecutive building blocks to the solution.
  - Holland mathematically proved that selective reproduction allows exponentially increasing number of samples of schemas with better-than-average fitness, and exponentially decreasing number of schemas with lower-than-average fitness.

Bloats

A widely observed phenomenon that no one can explain yet

- Average size of trees in the population remains relatively static for certain number of generations, then:
- It increases rapidly and significantly. This growth in size is **not accompanied by improvement in fitness**.
- Many attempt to explain why this happens; no unified theory yet.

Three Theoretical Attempts

- **Replication accuracy theory** (McPhee and Miller 1995): success of a GP individual depends on its ability to have offsprings that are functionally similar to itself, hence the tendency to repeat itself.
- **Removal bias theory** (Soule and Foster 1998): inactive (dead) code usually lies lower in the tree, and are smaller than average. When replaced (and removed), larger subtrees take their place, increasing the tree size.
- **Program Search Space theory** (Langdon and Poli 1997): above certain size, there is no correlation between size and fitness, but there are more longer programs, so they are just sampled more often.

Bloat Control

- Size and depth limit: do not accept too large individuals into population
- Bloat-aware genetic operators: do not generate too large individuals
- Bloat aware selection: consider program size as part of selection pressure

Schema Theorem

- Schema: a Hyperplane in the search space
  - For example, “11***”, where * is a “don’t care” symbol
- Instance: a concrete solution; a schema can include multiple instances
  - 11*** contains 2^3 solution instances, e.g., 11111, 11001, …

Schema Theorem

- Fitness of a schema: the mean fitness of all instances of the given schema
- Naturally, global optimisation is to find the schema that has 1) the highest fitness value, and 2) zero “don’t care” symbol
- Holland argues that analysis of GA behaviour becomes easier if we discuss in terms of groups of individuals (i.e., schema), rather than individual solutions

Features of a Schema

- Order (o): number of positions the schema that do not have the don't care symbol
  - Given H = 1**0*1*0, $o(H) = 4$
- Defining length (δ): distance between the outermost defined positions, which equals the number of possible crossover points
  - Given H = 1**0*1*0, $\delta(H) = 7$

Schema Theorem

- Assuming Fitness Proportional Selection, Single-point Crossover, Bitwise Mutation, Generational Selection:

$$\mathbb{E}(m(H, t+1)) \geq m(H, t)\frac{f(H)}{a_t}(1 - \frac{\delta(H)}{l-1}p_c)(1-p_m)^{o(H)}$$

  - $m(H, t)$: number of instances of schema H at generation t
  - a_t: average fitness of population at t
  - l: length of chromosomes
  - $o(H)$: order of H, $\delta(H)$: defining length of H
  - p_c: crossover rate ,p_m: mutation rate

Exploration vs. Exploitation

A cross-cutting concern in all learning/optimisation

- Exploration: we need to try never-seen-before solutions, because you never know.
    - With EC, it is typically cross-over that takes you to unexpected solution space.
- Exploitation: we need to try something similar to known good solutions, because there may be even better ones nearby.
    - With EC, it is typically mutation that lets you explore neighbourhood solutions.

Variations: Memetic Search

- Combine the global search with local search.
  - Remember, an individual solution in Genetic Algorithm arrives at the current location in the solution space NOT by moving through the space, rather by being dropped there as the result of crossover.
  - So, if there is a gradient towards a better solution, why not move along the gradient a little bit?
- **Memetic Search**: every now and then during evolutionary search, perform local search starting from the positions of individual solutions.

Summary

- Evolutionary Computation is a broad branch of optimisation/metaheuristic/ artificial intelligence that aims to mimic the process of Darwinian evolution for problem solving.
- It works best when combining small building blocks can lead to better solutions for your problem.
- Diversity in the population is the key; with sufficient diversity, you then have to balance exploration and exploitation.

Summary

- Evolutionary Computation can be a key component when the given problem does not show any analytical structure.
  - Example: Neural Architecture Search for DNN models