**EVOLUTIONARY COMPUTATION**

**Evolutionary computation** is a family of [algorithms](https://en.wikipedia.org/wiki/Algorithm) for [global optimization](https://en.wikipedia.org/wiki/Global_optimization) inspired by [biological evolution](https://en.wikipedia.org/wiki/Biological_evolution), and the subfield of [artificial intelligence](https://en.wikipedia.org/wiki/Artificial_intelligence) and soft computing studying these algorithms. In technical terms, they are a family of population-based trial and error problem solvers with a [metaheuristic](https://en.wikipedia.org/wiki/Metaheuristic) or [stochastic optimization](https://en.wikipedia.org/wiki/Stochastic_optimization) character.

An initial set of candidate solutions is generated and iteratively updated. Each new generation is produced by stochastically removing less desired solutions and introducing small random changes.

This kind of approach in machine learning is useful when there is no hypothesis (or very few) about the nature of the solution. We just have a way to assess a candidate solution.

EC is used in different fields, we will see some aspects of the unified approach:

* general scheme
* terminology
* some significant variants
* general usage guidelines

Evolutionary process: **basic scheme**

* a **population of individuals** (solutions) compete for **limited resources**
* the population is **dynamic**: individuals die and are born
* **fittest** individual survives and reproduce more than the others
* **offspring** inherit some characters from parents: they are similar but not identical (variational inheritance)

We need to define what are **individual**, **population** and **resources**; how individuals **compete**, how **fitness** is measured and how individuals **reproduce**.

The **individual** is a candidate solution for the considered problem, it is called **phenotype**.

e.g. a program in a programming language, a set of parameters, a picture…

The goal of the process is to find the optimal one.

The individual is internally represented by its **genotype**, that is, the individual’s characteristics given by the individual itself and some well-defined data structures. We talk about **indirect representation** if phenotype != genotype.

We talk about phenotype and genotype to resemble nature and to ease manipulation: for recombination, we consider genotype (bit/int string, Rn vector, tree, …) and for fitness phenotype.

A **population** is a set of individuals with a fixed (maximum) size. “limited resources” is a place in the population. The population is dynamic, that is individual born and die.

The **generational model** or replacement strategy determines how and when individuals are replaced.

Underlying (and common) **assumptions**:

* individuals’ life is instantaneous
* given the genotype, the phenotype (if any) and the fitness are immediately known and remain constant the whole life
* time flowing is determined by births (and deaths)

In the general scheme for the generational model we have:

* a population of **m parents**
* a population of **n offspring**
* a Boolean flag: overlapping

**Common values for m** are between 10 and 1000, it depends on the algorithm.

Usually the population size is constant because if the number of individuals keep increasing at a certain point the computer will not be able to handle data, if it keeps decreasing at a certain point there will be no individuals.

At each time tick:

1. build n offspring from the m parents

2. obtain an n + m population by merging parents and offspring

3. select m individuals to survive

**Overlapping generation model**

At each time tick:

1. build n offspring from the m parents

2. obtain an n + m population by merging parents and offspring

3. select m individuals to survive

These scheme is called “n+m with overlapping”

**Non-Overlapping generation model**

At each time tick:

1. build n offspring from the m parents (assume n ≥ m)

2. select m individuals to survive among the n offspring

**All parents die!**

Common cases:

n=m, overlapping

n=m, non-overlapping

n = 0.8\*m, overlapping

n=1, overlapping (steady state)

Possible **problem**: different levels of dynamicity in the single time tick can make different variant comparison difficult. **Solution**: measure time flowing as number of birth, referred to population size m. A generation occurs each m births.

**Selection criteria**

Let’s consider some option to define how to **select individuals to survive** and **parents to reproduce**.

* uniform (neutral) selection
* fitness-proportional selection
* rank-proportional selection
* truncation selection
* tournament selection
* …

**Fitness-proportional**:

1. given the **numerical fitness** of each individual

2. randomly pick one individual with probability proportional to the fitness (the better, the larger probability)

**Rank-proportional:**

1. given the rank of each individual in a fitness-based ranking (not necessarily numerical)

2. randomly pick one individual with probability proportional to the rank (the better, the larger probability)

(Can be applied to a non-numerical fitness, in principle)

**Uniform**:

 1. pick randomly an individual (with **uniform probability**)

**Truncation**:

1. pick the best individual (**elitism**)

(Deterministic, doesn’t involve probabilities)

**Tournament selection (uniform + truncation)**:

Given a parameter **nsize** (size of the tournament):

1. randomly (with uniform probability) pick nsize individuals

2. from them, choose the one with the best fitness

**Comparing criteria**: criteria differ in how strongly they tend to prefer fit vs. unfit individuals:

uniform selection: no preferences

truncation selection: strong preference of fit individuals

tournament: nsize → 1: no preference, nsize → m: strong preference

With **strong preference** (or selective/evolutionary pressure): population tends to converge to fittest individuals and evolution concentrates in improving most promising solutions: this is called **exploitation**. The risk is of “falling” in local optimum.

With weak preference (or selective/evolutionary pressure): population includes also unfit individuals and evolution investigates many different (maybe not promising) solutions: this is called **exploration**. The risk is of not finding a good solution.

Exploration/exploitation trade-off is hard to rule!

**Common choices for selectors**

Reproduction: tournament of nsize, e.g., m=npop =500, nsize =5

Survival: truncation

Reproduction: fitness proportional

Survival: truncation

**Reproduction**

Reproduction consists in building n offspring from the m parents. Reproduction happens in the genotype space, it can be seen as a function defined in this space.

General scheme:

Given one or more parents, an offspring is generated by applying a unary or binary genetic operator on parent genotypes:

unary (**mutation**): f : G → G

binary (**recombination or crossover**): f : G2 → G

Given n and a set of weighted operators, generate offspring with operators according to their weights (deterministically or stochastically)

**Choice of operators**

Usually, we have this combination:

Operators:

* crossover for generating 80% of offspring
* mutation for generating 20% of offspring

Deterministically:

1. for 0.8n times

1.1 select 2 parents (with reproduction selection criterion)

1.2 apply crossover to genotypes

2. for 0.2n times

2.1 select 1 parent (with reproduction selection criterion)

2.2 apply mutation to genotype

Stochastically:

1. for n times

1.1 randomly choose between mutation/crossover with 20/80 probability

1.2 select 1 or 2 parents (with reproduction selection criterion) accordingly

1.3 apply operator to genotype(s)

Actually, there is another method: apply crossover and then mutation.

**Mutation for bit string genotypes**

Most classical option: **probabilistic bit flip mutation**

1. copy parent genotype gp as child genotype gc

2. for each bit in gc, flip it (0→1or1→0) with p probability

Commonly, p = 0.01

gp = 001010011101010101100100101

gc = 001010111101010101101100101

For each bit, there is a small probability to be flipped.

Bit string is a common representation for individuals.

**Crossover for bit strings genotypes**

We have genotypes g1 and g2, we want to obtain the child genotype gc.

g1

gc

g2

Many options:

* one-point crossover
* two-points crossover
* n-points crossover
* uniform crossover
* ...

**One -, two-, n-points crossover**

Assume parents with equal genotype size (it is not always true):

1. choose randomly one (two, n) cut points in the genotype (indexes i such that i < |gp1 | = |gp2 |)

2. child bits before the cut point comes from parent 1, child bits

after the cut point comes from parent 2

In general, j-th bit comes from parent 1 iff closest larger cut point is even, from 2, otherwise.

One-point:

gp1 =00101001110101010|1100100101

gp2 =11101010101001010|0101110111

gc = 00101001110101010 0101110111

Two-points:

gp1 =0010100|1110101010|1100100101

gp2 =1110101|0101001010|0101110111

gc = 0010100 0101001010 1100100101

**Uniform crossover**: a cut point is placed at each index, with probability p=0.5 of inheriting the bit from each of the two parents.

**Crossover with variable length (bits) string genotype**

There are many variants:

* **one-, two-points crossover**:
* cut points may be different within parents
* child genotype size may be larger or smaller than parents’ sizes
* …

One-point:

gp1 =00101001110101010|1100100101

gp2 =111010101|010010100101110111

gc = 00101001110101010 010010100101110111

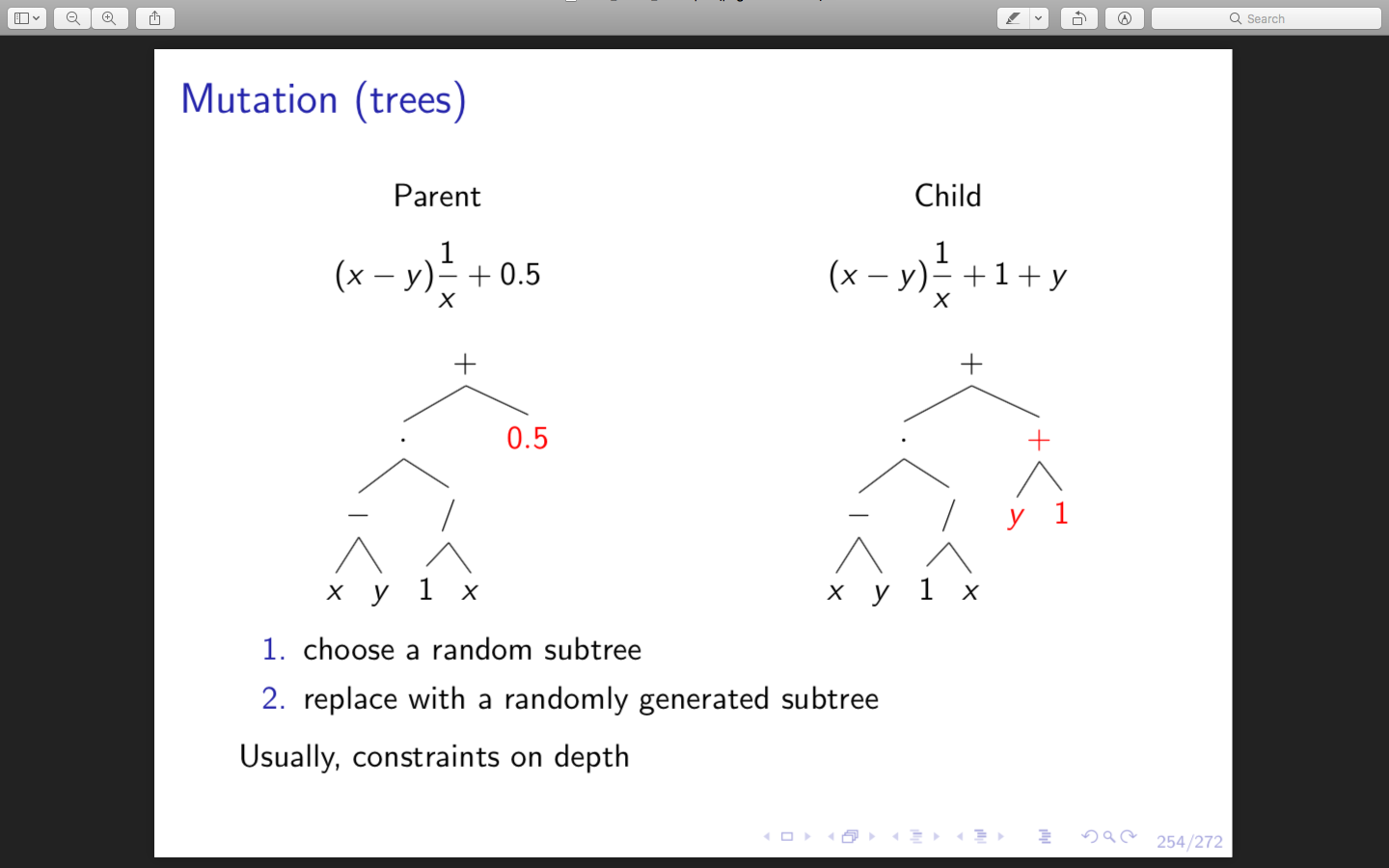
Genotype-phenotype mapping must allow for variable length genotypes!

**Mutation trees**

1. choose a random subtree

2. replace with a randomly generated subtree

Usually, constraints on depth.



One common way of using trees is to describe mathematical expressions:

We define F = {+, \*, -, /, sqrt()}

T = {x, y, z, 0.1, 1, 10}

Phenotypes is the space of mathematical expressions. We need genetic operators for combining and mutating trees.

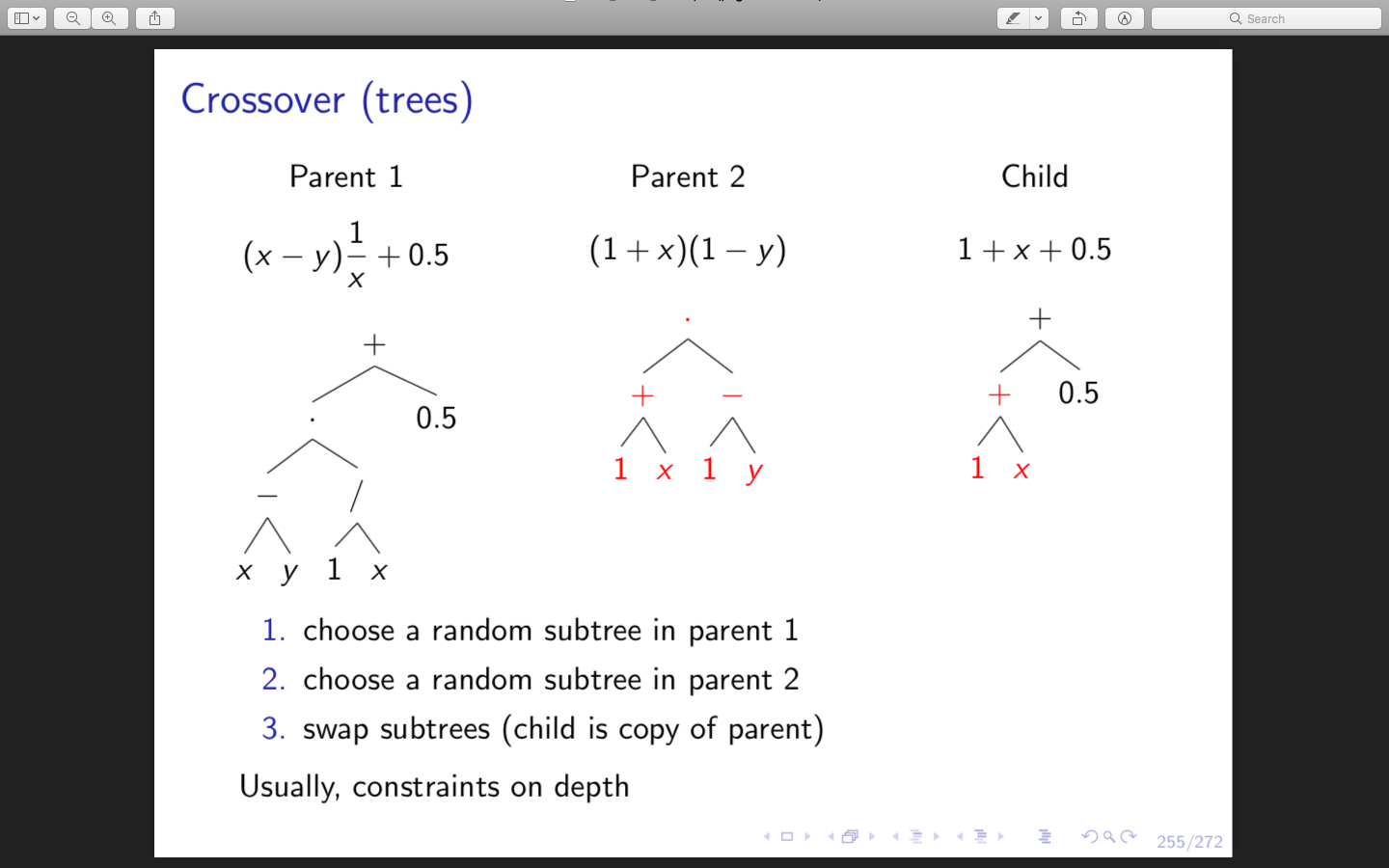
**Crossover trees**

1. choose a random subtree in parent 1

2. choose a random subtree in parent 2

3. swap subtrees (child is copy of parent)

Usually, constraints on depth



OSS: crossover is more about exploration while mutation is more about exploitation (mutation generally doesn’t move far away from the parent).

The population initialization can be totally random or follow more specific approaches, it depends on the genotype form.

**Fitness**

The fitness of an individual is the ability to solve the problem of interest.

* errors on several fitness cases by execution/simulation/application

A **fitness function** is a mapping from the phenotype space to the fitness space.

Common cases to measure the fitness:

* one numerical index
* more than one numerical indexes
* ...

It is closely related with selectors

**Many indexes: multiobjective optimization scenario**

NB: Multiobjective: less than 10, manyobjective: more than 10.

f(i) = ⟨f1(i), ..., fn(i)⟩

There are different ways to compare two individuals i1 and i2:

1. linearization

* f(i)=α1f1(i)+···+αnfn(i)

1. lexicographical order

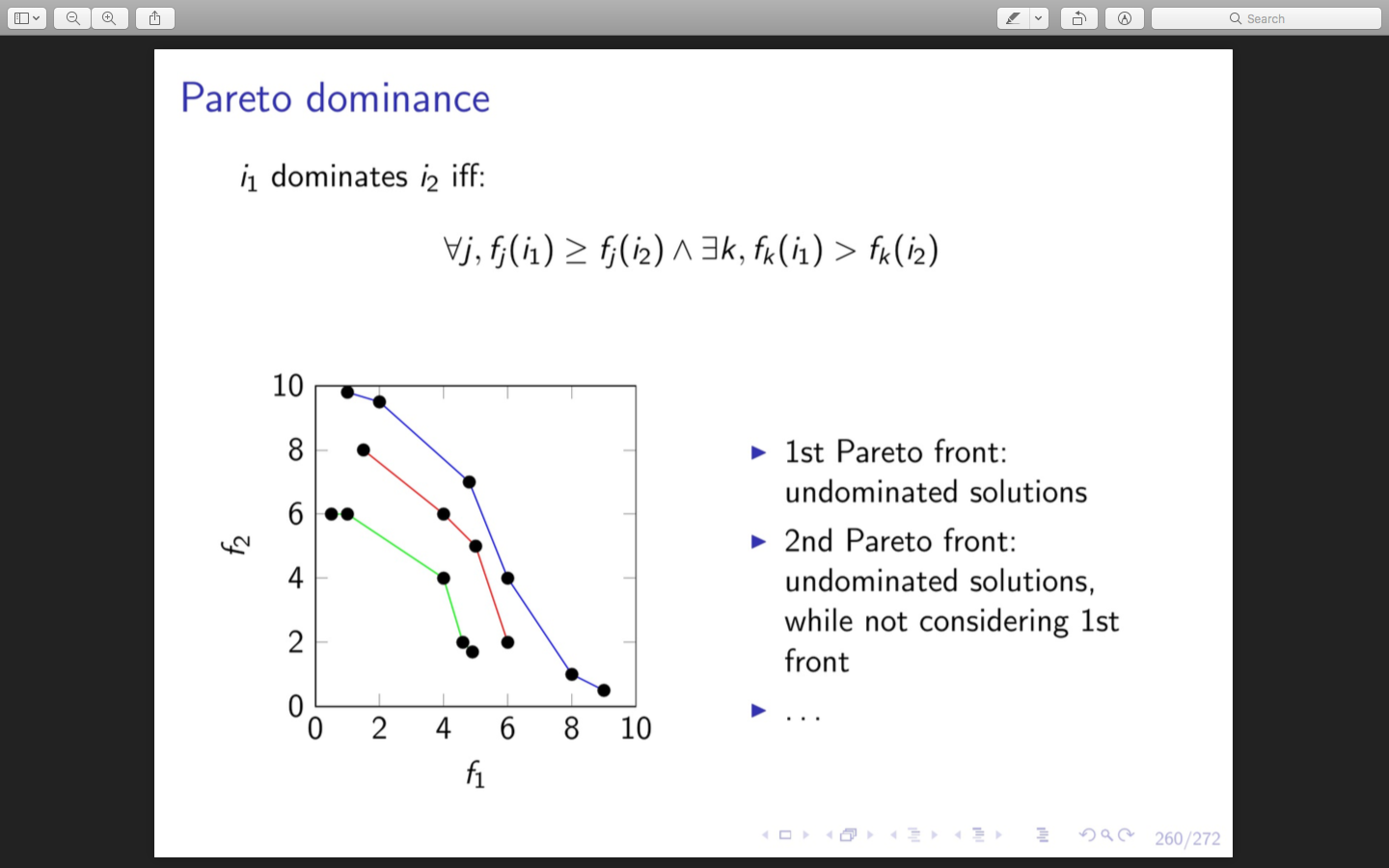
* compare f1(i1) > f1(i2); if tie, f2(i1) > f2(i2); . . .

1. Pareto dominance
2. ...

Q: with which selectors?

**Pareto dominance**

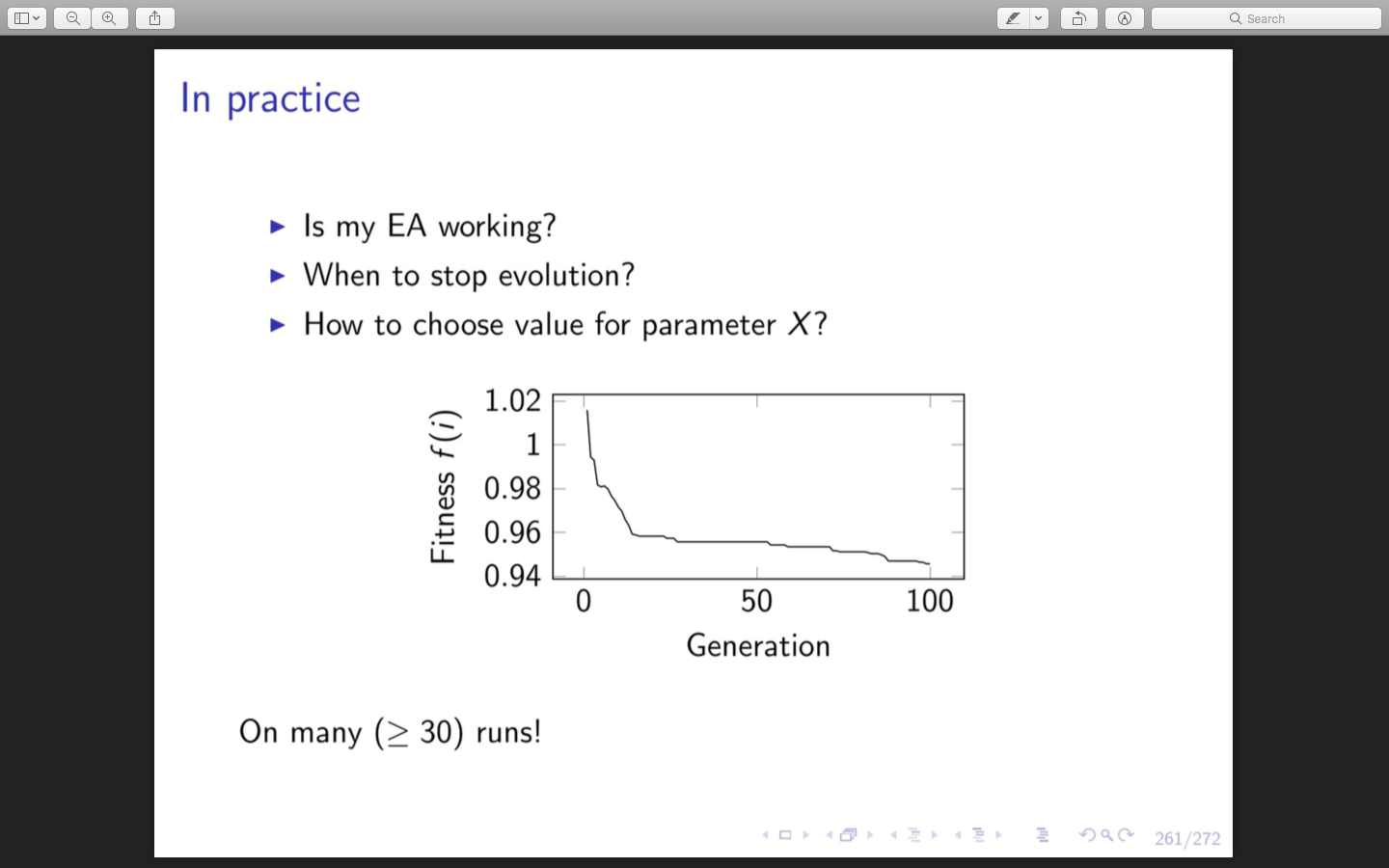
i1 dominates i2 iff: ∀j, fj (i1 ) ≥ fj (i2 ) ∧ ∃k, fk (i1 ) > fk (i2 )

****

Objectives in the same line are equivalent. This method can be use e.g. in tournament selection but not in fitness proportional selection.

In practice, we try to understand if the evolutionary algorithm works, when to stop evolution and how to choose a value for the parameter X. ???

We typically check if the algorithm is working in an experimental way, by plotting the following chart on many runs (> 30):



This can be done only for one objective systems. If there are few we can plot a chart for each objective.

We could also go beyond 100 generation, is depends on the system.

We should do it many times because many systim's components are stochastic.

Let’s analyse some issues:

* diversity in population
* variational inheritance
* expressiveness

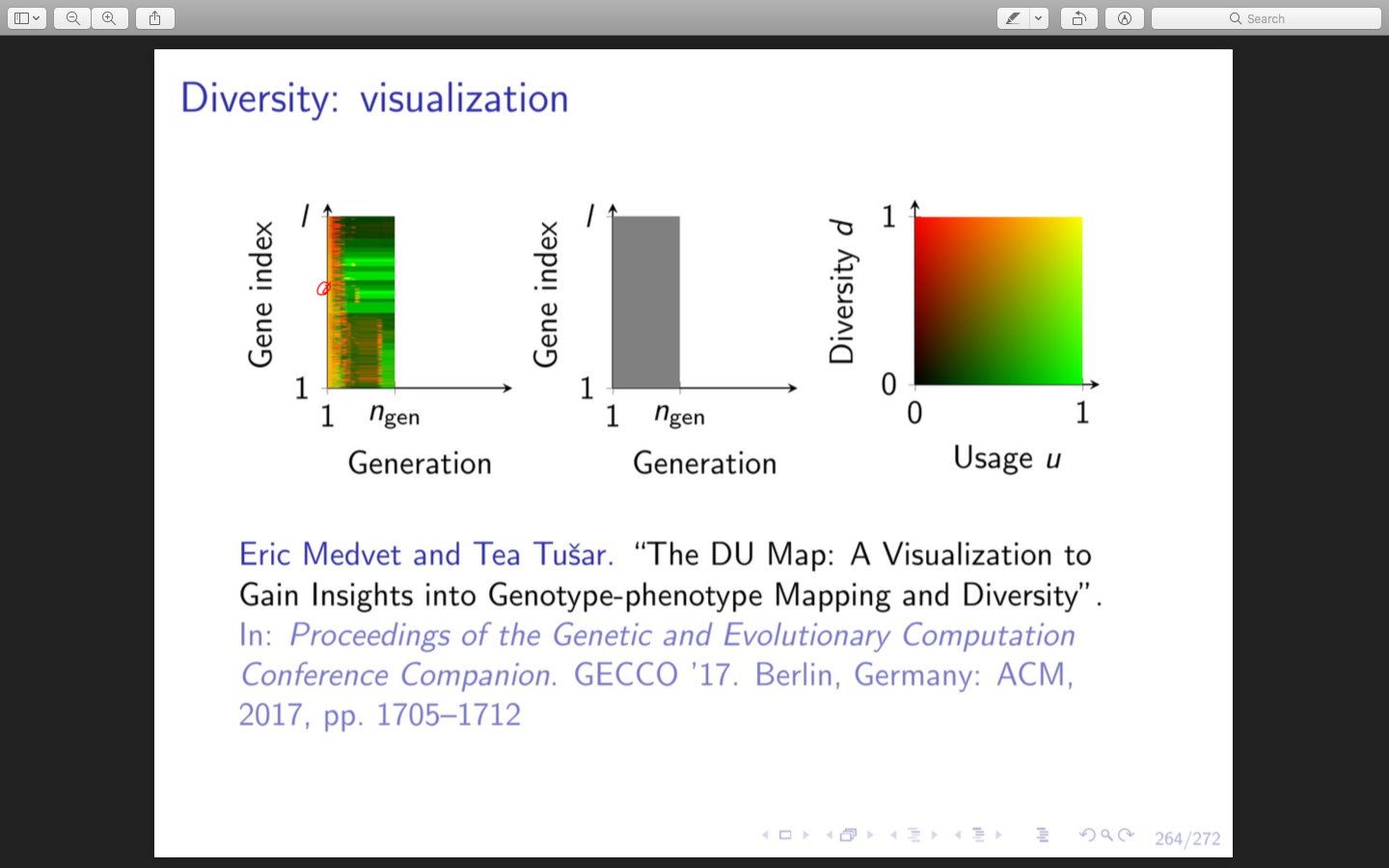
**Diversity**: is the population diverse enough?

“No” → too much exploitation → local minimum

“Yes” → in principle, no drawbacks

* + - how to measure diversity?
    - how to enforce/promote diversity?

If there is no diversity we are moving one point in the space and is more probable that the point ends up in the wrong place (local min/max).

Diversity visualization:  


Plot that shows what happens in evolution from two points of view:

* genotypic diversity at the level of the single bit
* contribution (usage) of the single bit to the mapping

and how they vary (interact) during the evolution.

Each point in the grey space (central plot) represent a given bit in a certain generation. In the left-hand plot, each of these points is coloured in base of the usage-diversity grade, according the right-hand plot.

The greener is the point the most is used this bit in a certain generation. The redder it is, the more diversity is present.

**Variational inheritance**: are children similar but not identical to parents?

* “Too much similar” → too much exploitation → local minimum, no/slow evolution
* “Too much different” → no exploitation, just coarse exploration (random walk)
* How to measure? (locality, redundancy, degeneracy, uniformity, ...)
* How to tackle? Operators, mapping, both?

**Expressivness**: Is the representation (phenotype) expressive enough?

* “Low expressiveness” → good/optimal solution might not be representable, or might not be reachable
* “Large expressiveness” → large search space → very long or infinity convergence time

**Fitness landscape**

How are genotype and fitness spaces related? What does a small step on one correspond to on the other?

Q: is phenotype space relevant?

**Genetic Algorithms (GA)**

It is the most widely used and study kind of evolutionary algorithm. Characterized by:

* Genotype = phenotype = bits string  (direct mapping)
* m=n≈1000
* no overlapping
* Fitness-proportional selection, or multiobjective (Pareto-based) selection
* Genotypes often encodes numerical parameters

**Genetic programming (GP)**

Popular for symbolic regression (type of regression analysis that searches the space of mathematical expressions to find the model that best fits a given dataset, both in terms of accuracy and simplicity).

In this case, individuals are programs.

* Genotype = phenotype = tree (tree-based GP) or list of instructions (linear GP)
* m=n≈1000, overlapping
* Tournament selection

We have to deal with syntactic/semantic validity, because we are talking about programs.

**Grammatical evolution**

Context free grammar are ways to define languages, it makes sure that every phenotype is a valid program. Mapping procedure is very important.

A form of GP based on GA, given a context-free grammar G

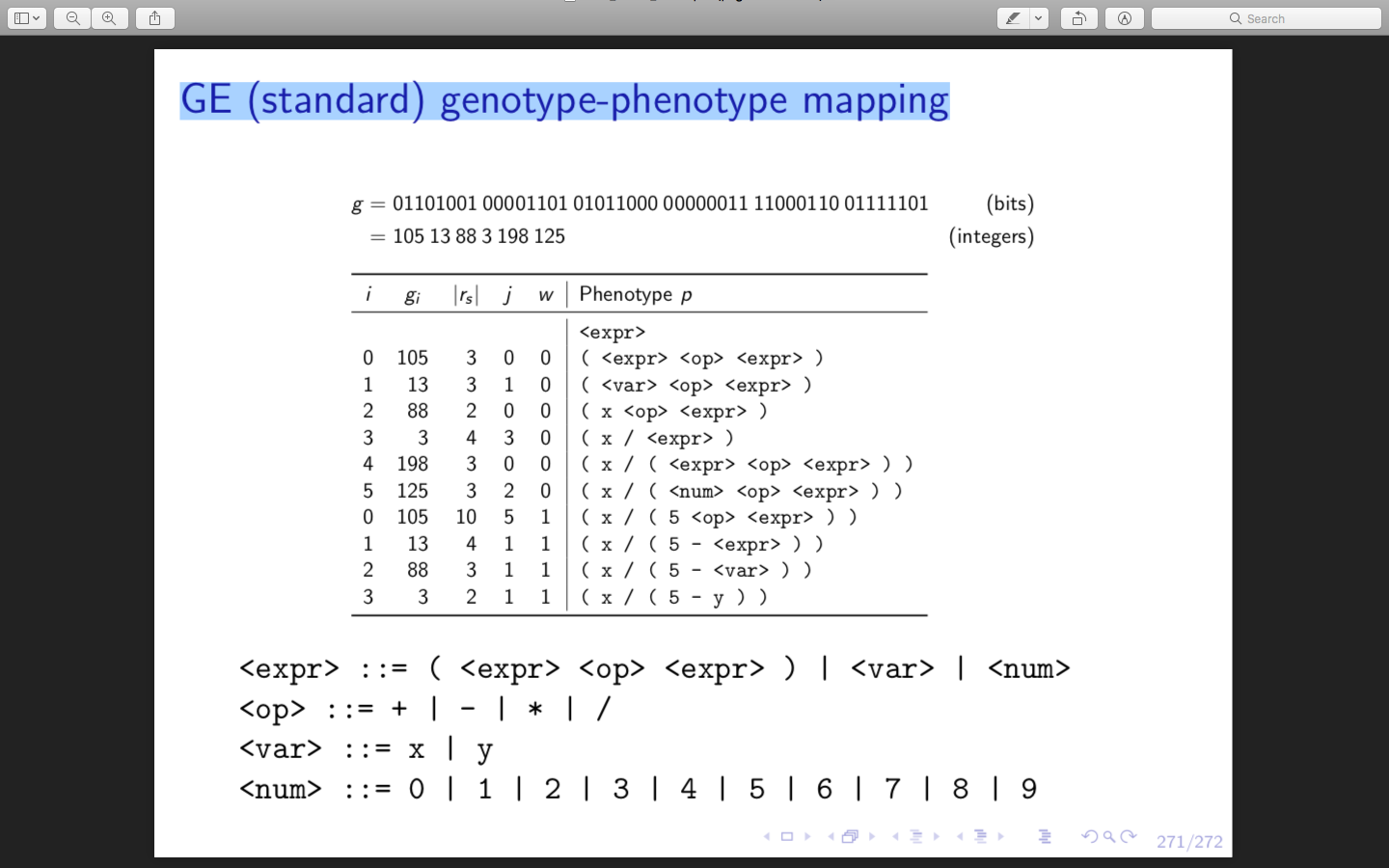
* Genotype = bits string, phenotype = string ∈ L(G), by means of a mapping procedure
* Steadystate (m≈500,n=1,overlapping) or m=n, overlapping
* Tournament selection

**GE (standard) genotype-phenotype mapping**

This is a standard way to specify grammars:

We first transform a string of bits in a string of integers. We consider the first integer, compute the remainder of the integer over the number of option available to represent the expression.

Changing a single bit can bring to a very different number: small distance in genotype 🡪 large distance in phenotype.



🡪 possible numbers

🡪 possible variables

🡪 possible operations

🡪 possible ways to describe the expression

**An alternative: WHGE genotype-phenotype mapping**

