Evolutionary Computing

Chapter 1

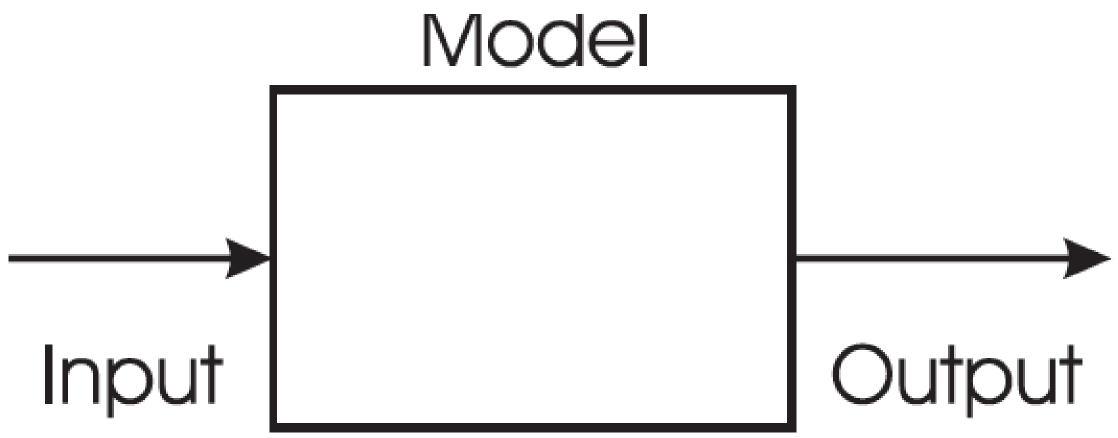
Angelo Gargantini
LaPeC 24

Chapter 1: Problems to be solved

Problems can be classified in different ways:

- Black box model
- Search problems
- Optimisation vs constraint satisfaction
- NP problems

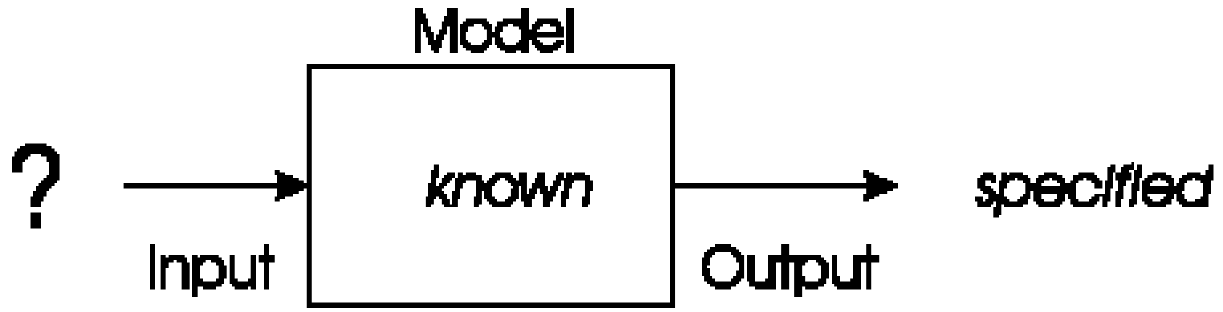
“Black box” model



- “Black box” model consists of 3 components
- When one component is unknown: new problem type

“Black box” model: Optimisation

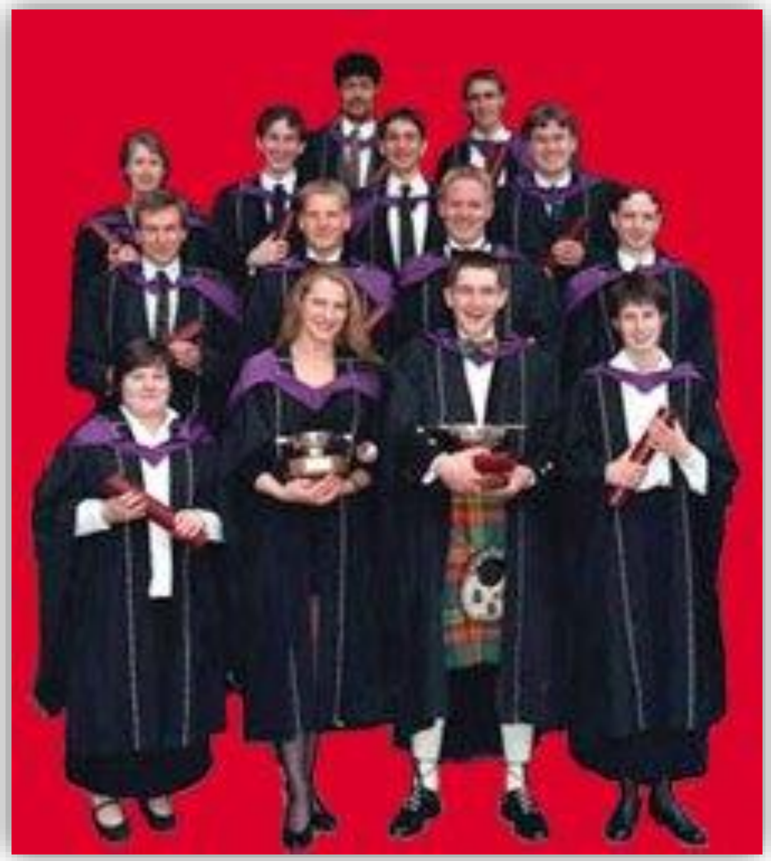
- Model and desired output is known, task is to find inputs

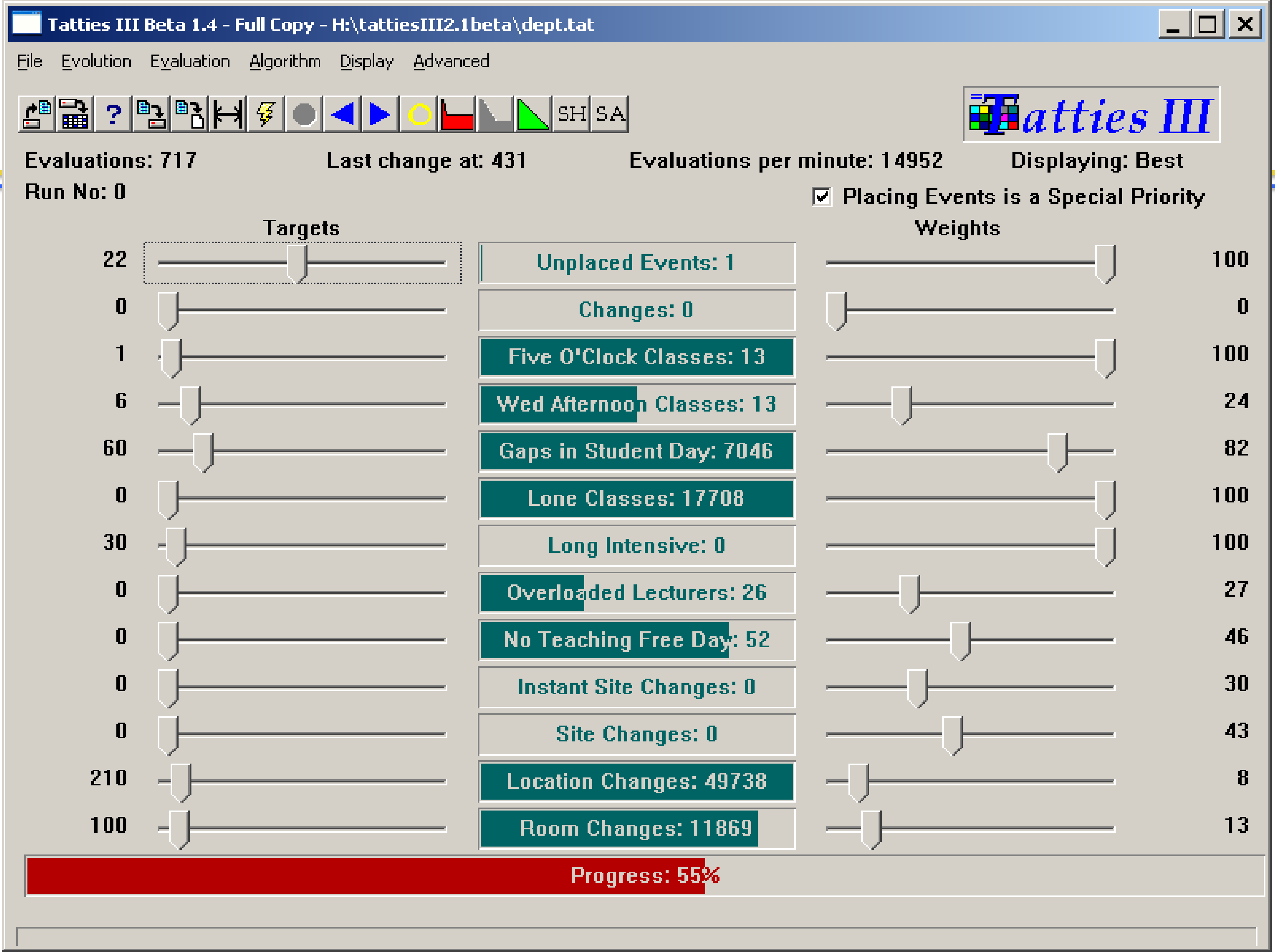


- Examples:
  - Time tables for university, call center, or hospital
  - Design specifications
  - Traveling salesman problem (TSP)
  - Eight-queens problem, etc.

“Black box” model:
Optimisation example 1: university timetabling

- Enormously big search space
- Timetables must be *good*
- “Good” is defined by a number of competing criteria
- Timetables must be feasible
- Vast majority of search space is infeasible




Tatties III Beta 1.4 - Full Copy - H:\tattiesIII2.1beta\dept.tat
File
Evolution
Evaluation
Algorithm
Display
Advanced
SH
SA
Tatties III

Evaluations: 717 Last change at: 431 Evaluations per minute: 14952 Displaying: Best

Run No: 0

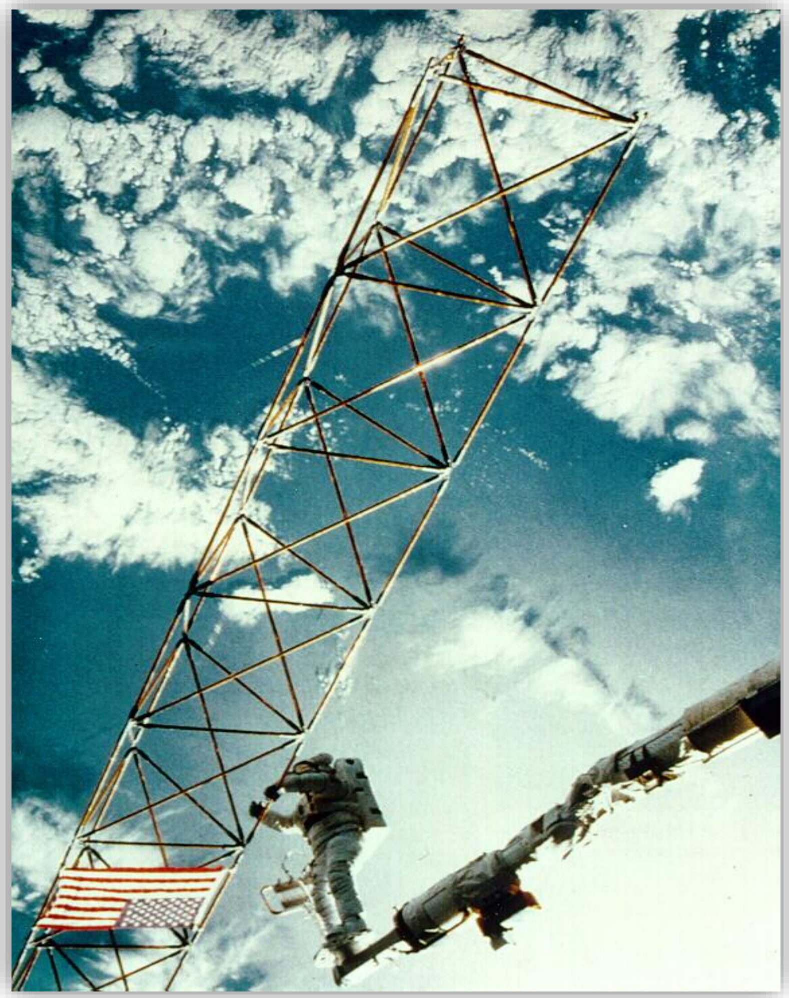
[✓] Placing Events is a Special Priority

Targets		Weights
22	Unplaced Events: 1	100
0	Changes: 0	0
1	Five O'Clock Classes: 13	100
6	Wed Afternoon Classes: 13	24
60	Gaps in Student Day: 7046	82
0	Lone Classes: 17708	100
30	Long Intensive: 0	100
0	Overloaded Lecturers: 26	27
0	No Teaching Free Day: 52	46
0	Instant Site Changes: 0	30
0	Site Changes: 0	43
210	Location Changes: 49738	8
100	Room Changes: 11869	13

Progress: 55%

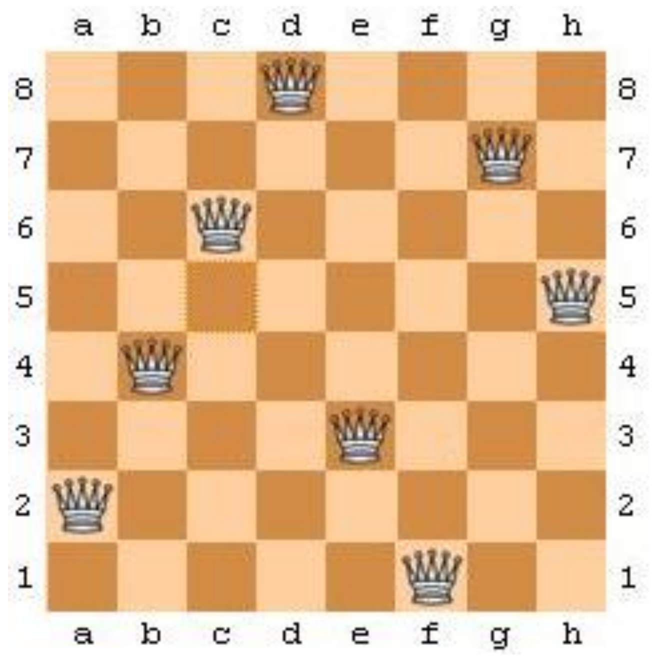
“Black box” model:
Optimisation example 2: satellite structure

- Optimised satellite designs for NASA to maximize vibration isolation
- Evolving: design structures
- Fitness: vibration resistance
- Evolutionary “creativity”



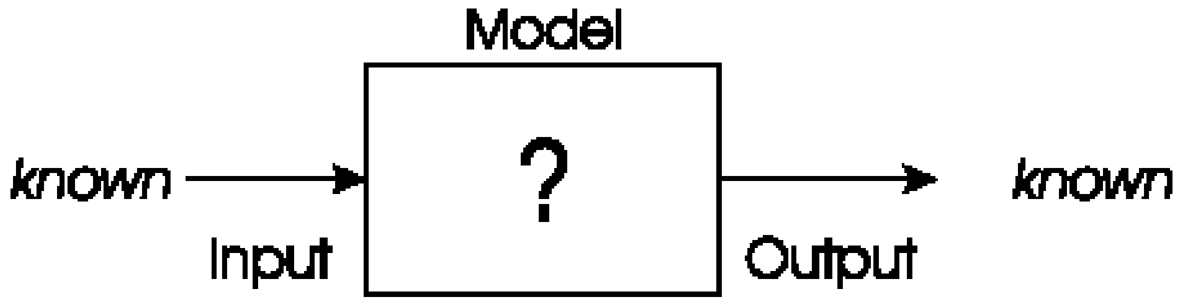
"Black box" model: Optimisation example 3: 8 queens problem

- Given an 8-by-8 chessboard and 8 queens
- Place the 8 queens on the chessboard without any conflict
- Two queens conflict if they share same row, column or diagonal
- Can be extended to an n queens problem (n>8)



“Black box” model: Modelling

- We have corresponding sets of inputs & outputs and seek model that delivers correct output for every known input



“Black box” model:
Modelling example: load applicant creditibility

- British bank evolved creditability model to predict loan paying behavior of new applicants

- Evolving: prediction models

- Fitness: model accuracy on historical data

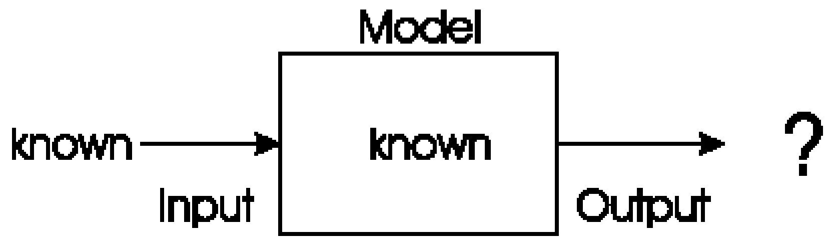
Other examples

- Evolutionary machine learning
- Predicting stock exchange
- Voice control system for smart homes

- Note: modelling problems can be transformed into optimisation problems

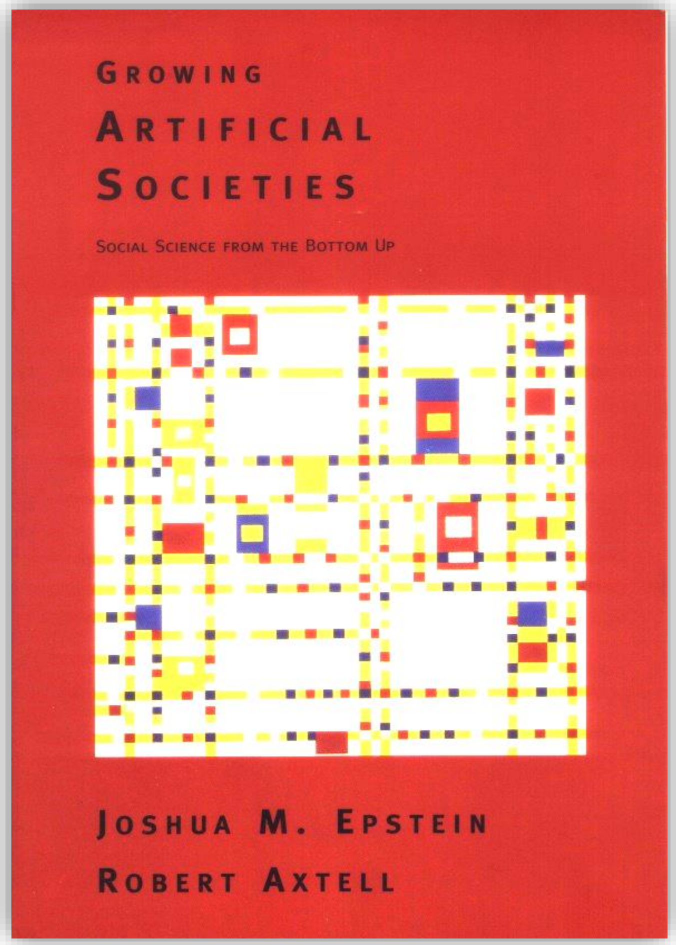
“Black box” model: Simulation

- We have a given model and wish to know the outputs that arise under different input conditions



- Often used to answer “what-if” questions in evolving dynamic environments
  - Evolutionary economics, Artificial Life
  - Weather forecast system
  - Impact analysis new tax systems

“Black box” model:
Simulation example: evolving artificial societies



Simulating trade, economic competition, etc. to calibrate models

Use models to optimise strategies and policies

Evolutionary economy

Survival of the fittest is universal (big/small fish)

"Black box" model:
Simulation example 2: biological interpretations

Picture censored

Incest prevention keeps evolution from rapid degeneration

(we knew this)

Multi-parent reproduction, makes evolution more efficient

(this does not exist on Earth in carbon)

2nd sample of Life

Search problems

- Simulation is different from optimisation/modelling
- Optimisation/modelling problems search through huge space of possibilities
- Search space: collection of all objects of interest including the desired solution
- Question: how large is the search space for different tours through *n* cities?

Benefit of classifying these problems: distinction between
- search problems, which define search spaces, and
- problem-solvers, which tell how to move through search spaces.

Optimisation vs. constraint satisfaction (1/2)

- Objective function: a way of assigning a value to a possible solution that reflects its quality on scale
  - Number of un-checked queens (maximize)
  - Length of a tour visiting given set of cities (minimize)
- Constraint: binary evaluation telling whether a given requirement holds or not
  - Find a configuration of eight queens on a chessboard such that no two queens check each other
  - Find a tour with minimal length where city X is visited after city Y

Optimisation vs. constraint satisfaction (2/2)

- When combining the two:

	Objective function	
Constraints	**Yes**	**No**
Yes	Constrained optimisation problem	Constraint satisfaction problem
No	Free optimisation problem	No problem

- Where do the examples fit?
- Note: constraint problems can be transformed into optimisation problems
- Question: how can we formulate the 8-queens problem in to a FOP/CSP/COP?

NP problems

- We only looked at classifying the problem, not discussed problem solvers
- This classification scheme needs the properties of the problem solver
- Benefit of this scheme: possible to tell how difficult the problem is
- Explain the basics of this classifier for combinatorial optimisation problems (booleans or integers search space)

NP problems:
Key notions

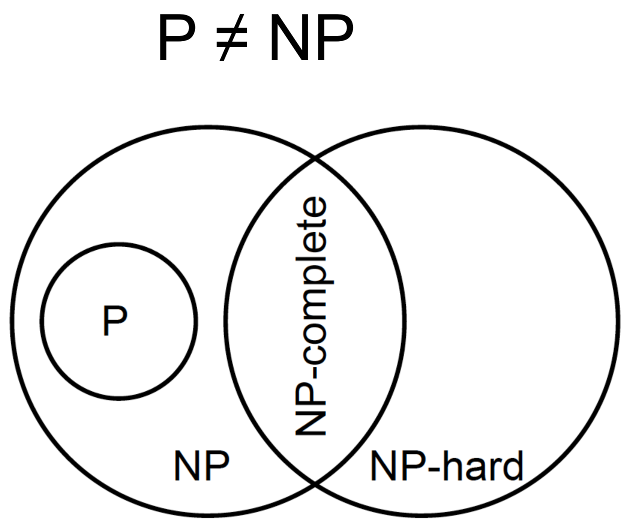
- Problem size: dimensionality of the problem at hand and number of different values for the problem variables
- Running-time: number of operations the algorithm takes to terminate
  - Worst-case as a function of problem size
  - Polynomial, super-polynomial, exponential
- Problem reduction: transforming current problem into another via mapping

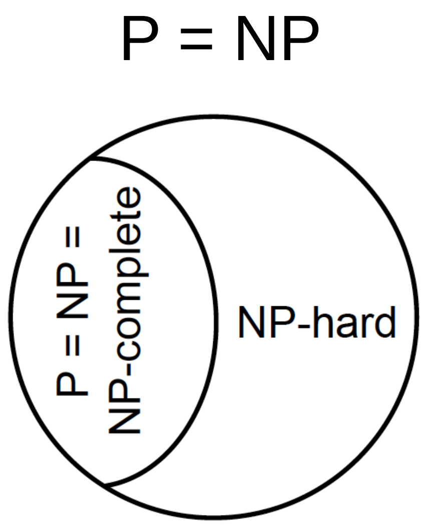
NP problems: Class

- The difficultness of a problem can now be classified:
  - Class P: algorithm can solve the problem in polynomial time (worst-case running-time for problem size n is less than F(n) for some polynomial formula F)
  - Class NP: problem can be solved and any solution can be verified within polynomial time by some other algorithm (P subset of NP)
  - Class NP-complete: problem belongs to class NP and any other problem in NP can be reduced to this problem by al algorithm running in polynomial time
  - Class NP-hard: problem is at least as hard as any other problem in NP-complete but solution cannot necessarily be verified within polynomial time

NP problems: Difference between classes

- P is different from NP-hard
- Not known whether P is different from NP





- For now: use of approximation algorithms and metaheuristics

Evolutionary Computing

Chapter 2

Angelo Gargantini
LaPeC 24

Chapter 2: Evolutionary Computing: the Origins

- Historical perspective
- Biological inspiration:
  - Darwinian evolution theory (simplified!)
  - Genetics (simplified!)
- Motivation for EC

Historical perspective (1/3)

- 1948, Turing:
  proposes “genetical or evolutionary search”
- 1962, Bremermann:
  optimization through evolution and recombination
- 1964, Rechenberg:
  introduces evolution strategies
- 1965, L. Fogel, Owens and Walsh:
  introduce evolutionary programming
- 1975, Holland:
  introduces genetic algorithms
- 1992, Koza:
  introduces genetic programming

Historical perspective (2/3)

- 1985: first international conference (ICGA)
- 1990: first international conference in Europe (PPSN)
- 1993: first scientific EC journal (MIT Press)
- 1997: launch of European EC Research Network EvoNet

Historical perspective (3/3)

EC in the early 21st Century:

- 3 major EC conferences, about 10 small related ones
- 4 scientific core EC journals
- 1000+ EC-related papers published last year(estimate)
- uncountable (meaning: many) applications
- uncountable (meaning: ?) consultancy and R&D firms
- part of many university curricula

Darwinian Evolution (1/3): Survival of the fittest

- All environments have finite resources
  (i.e., can only support a limited number of individuals)
- Life forms have basic instinct/ lifecycles geared towards reproduction
- Therefore some kind of selection is inevitable
- Those individuals that compete for the resources most effectively have increased chance of reproduction
- Note: fitness in natural evolution is a derived, secondary measure, i.e., we (humans) assign a high fitness to individuals with many offspring

Darwinian Evolution (2/3): Diversity drives change

- Phenotypic traits:
  - Behaviour / physical differences that affect response to environment
  - Partly determined by inheritance, partly by factors during development
  - Unique to each individual, partly as a result of random changes
- If phenotypic traits:
  - Lead to higher chances of reproduction
  - Can be inherited

then they will tend to increase in subsequent generations, leading to new combinations of traits …

Darwinian Evolution (3/3): Summary

- Population consists of diverse set of individuals
- Combinations of traits that are better adapted tend to increase representation in population

Individuals are “units of selection”

- Variations occur through random changes yielding constant source of diversity, coupled with selection means that:

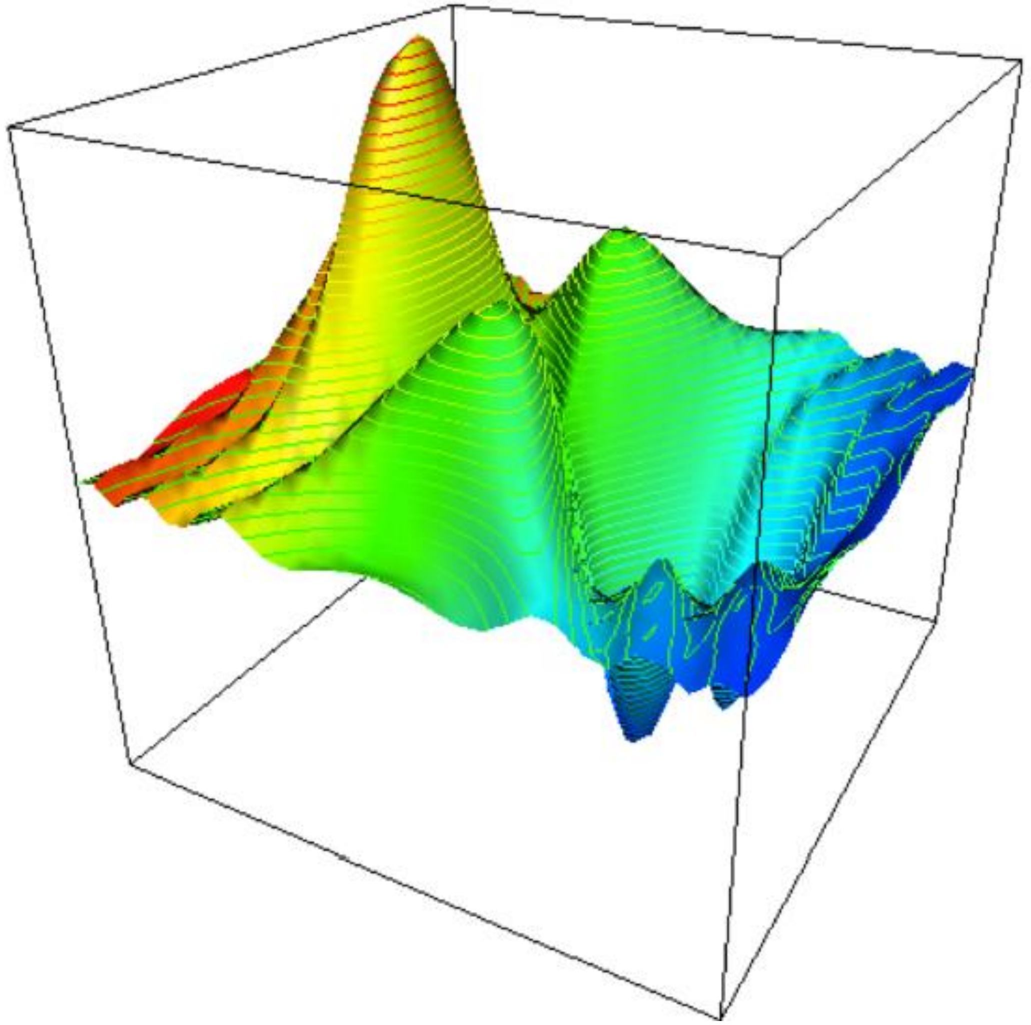
Population is the “unit of evolution”

- Note the absence of “guiding force”

Adaptive landscape metaphor (Wright, 1932)

- Can envisage population with *n* traits as existing in a *n+1*-dimensional space (landscape) with height corresponding to fitness
- Each different individual (phenotype) represents a single point on the landscape
- Population is therefore a “cloud” of points, moving on the landscape over time as it evolves – adaptation

Adaptive landscape metaphor (Wright, 1932)



Adaptive landscape metaphor (cont'd)

- Selection "pushes" population up the landscape

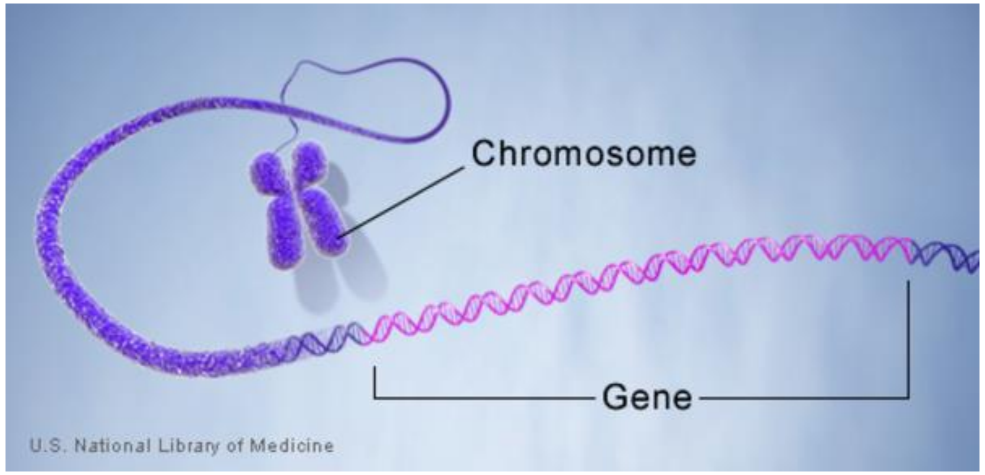
- Genetic drift:
  - random variations in feature distribution
  - (+ or -) arising from sampling error
  - can cause the population "melt down" hills, thus crossing valleys and leaving local optima

Genetics: Natural

- The information required to build a living organism is coded in the DNA of that organism
  - In humans, genes vary in size from a few hundred DNA bases to more than 2 million bases.
  - An international research effort called the Human Genome Project, which worked to determine the sequence of the human genome and identify the genes that it contains, estimated that humans have between 20,000 and 25,000 genes.
- Genotype (DNA inside) determines phenotype
- Genes $\rightarrow$ phenotypic traits is a complex mapping
  - One gene may affect many traits (pleiotropy)
  - Many genes may affect one trait (polygeny)
- Small changes in the genotype lead to small changes in the organism (e.g., height, hair colour)

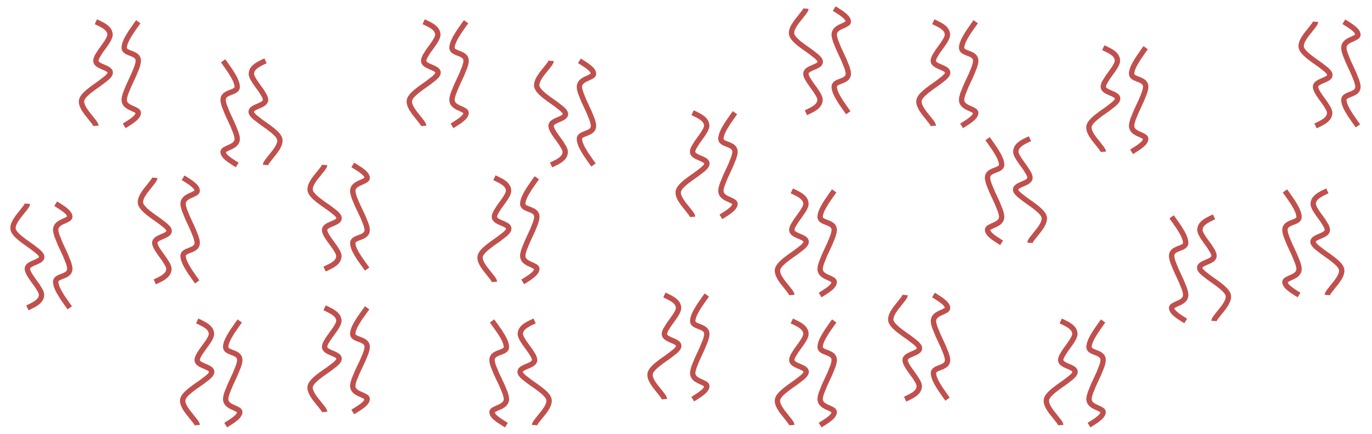
Genetics:
Genes and the Genome

- Genes are encoded in strands of DNA called chromosomes
- In most cells, there are two copies of each chromosome (diploidy)
- The complete genetic material in an individual's genotype is called the Genome
- Within a species, most of the genetic material is the same



Genetics:
Example: Homo Sapiens

- Human DNA is organised into chromosomes
- Human body cells contains 23 pairs of chromosomes which together define the physical attributes of the individual:

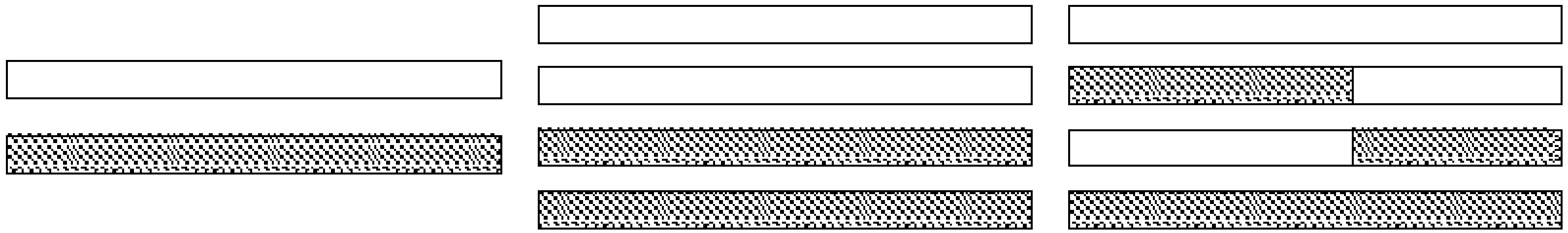


Genetics:
Reproductive Cells

- Gametes (sperm and egg cells) contain 23 individual chromosomes rather than 23 pairs
- Cells with only one copy of each chromosome are called haploid
- Gametes are formed by a special form of cell splitting called meiosis
- During meiosis the pairs of chromosome undergo an operation called *crossing-over*

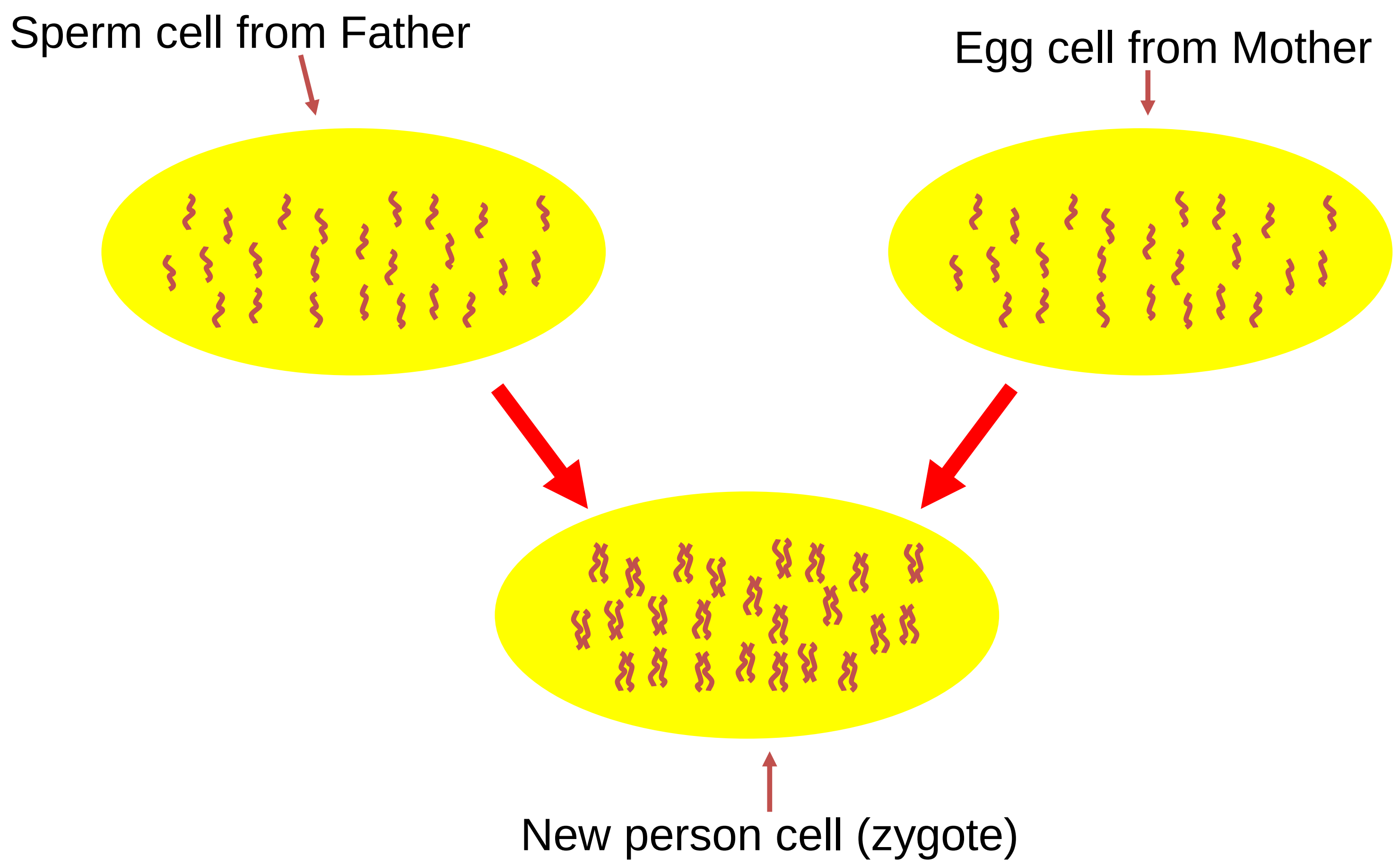
Genetics:
Crossing-over during meiosis

- Chromosome pairs align and duplicate
- Inner pairs link at a *centromere* and swap parts of themselves



- Outcome is one copy of maternal/paternal chromosome plus two entirely new combinations
- After crossing-over one of each pair goes into each gamete

Genetics: Fertilisation



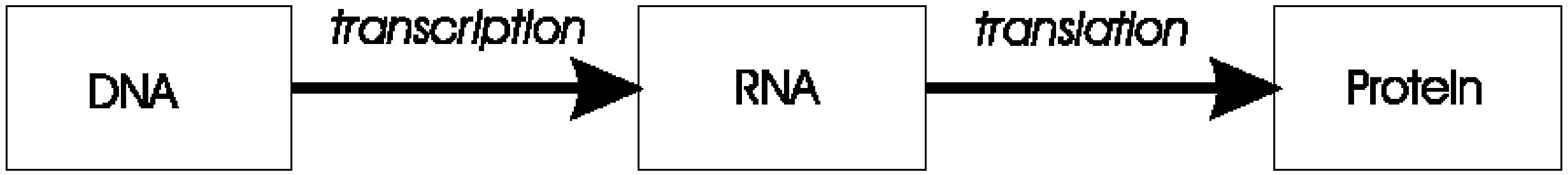
Genetics:
After fertilisation

- New zygote rapidly divides etc creating many cells all with the same genetic contents
- Although all cells contain the same genes, depending on, for example where they are in the organism, they will behave differently
- This process of differential behaviour during development is called ontogenesis
- All of this uses, and is controlled by, the same mechanism for decoding the genes in DNA

Genetics:
Genetic code

- All proteins in life on earth are composed of sequences built from 20 different amino acids
- DNA is built from four nucleotides in a double helix spiral: purines A,G; pyrimidines T,C
- Triplets of these from *codons*, each of which codes for a specific amino acid
- Much redundancy:
  - purines complement pyrimidines
  - the DNA contains much rubbish
  - 4^3=64 codons code for 20 amino acids
  - genetic code = the mapping from codons to amino acids
- For all natural life on earth, the genetic code is the same !

Genetics:
Transcription, translation



A central claim in molecular genetics: only one way flow

$$\text{Genotype} \longrightarrow \text{Phenotype}$$

$$\text{Genotype} \not\longleftarrow \text{Phenotype}$$

Lamarckism (saying that acquired features can be inherited) is thus wrong!

Genetics: Mutation

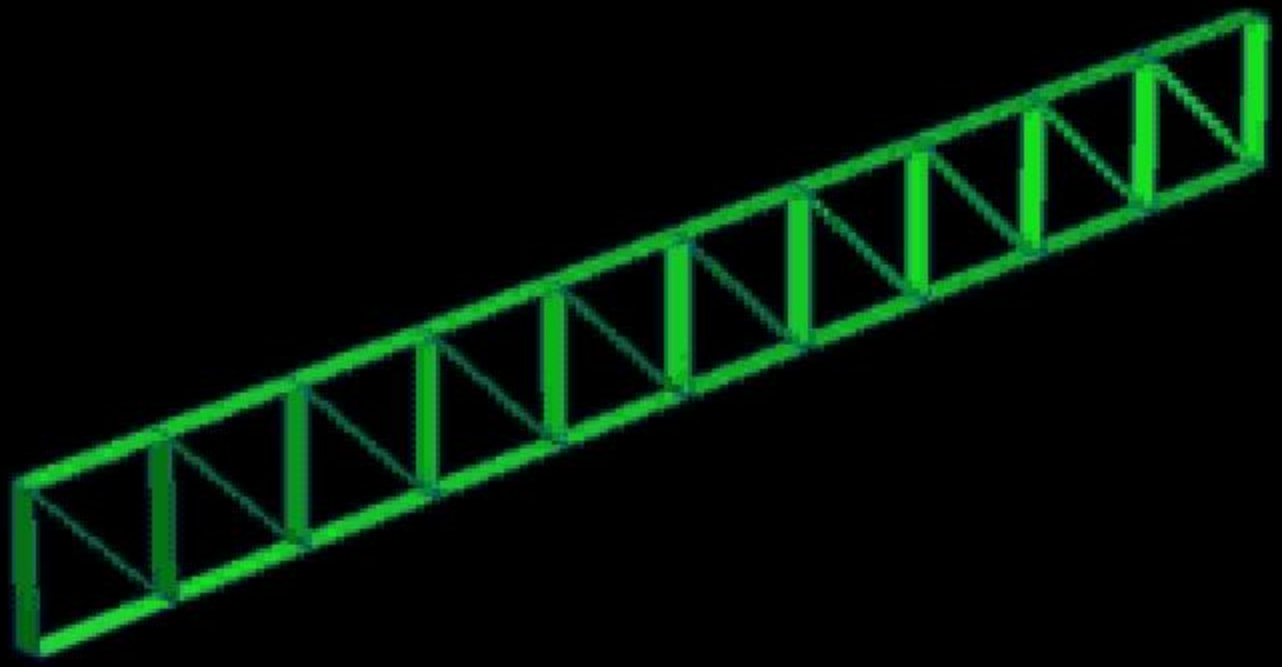
- Occasionally some of the genetic material changes very slightly during this process (replication error)
- This means that the child might have genetic material information not inherited from either parent
- This can be
  - catastrophic: offspring in not viable (most likely)
  - neutral: new feature not influences fitness
  - advantageous: strong new feature occurs
- Redundancy in the genetic code forms a good way of error checking

Motivation for evolutionary computing (1/2)

- Nature has always served as a source of inspiration for engineers and scientists
- The best problem solver known in nature is:
  - the (human) brain that created “the wheel, New York, wars and so on” (after Douglas Adams’ Hitch-Hikers Guide)
  - the evolution mechanism that created the human brain (after Darwin’s Origin of Species)
- Answer 1 → neurocomputing
- Answer 2 → evolutionary computing

Motivation for evolutionary computing (2/2)

- Developing, analyzing, applying problem solving methods a.k.a. algorithms is a central theme in mathematics and computer science
- Time for thorough problem analysis decreases
- Complexity of problems to be solved increases
- Consequence: ROBUST PROBLEM SOLVING technology needed



Evolutionary Computing

Chapter 3

Angelo Gargantini
LaPeC 24

Recap of EC metaphor (1/2)

- A population of individuals exists in an environment with limited resources
- ***Competition*** for those resources causes selection of those ***fitter*** individuals that are better adapted to the environment
- These individuals act as seeds for the generation of new individuals through recombination and mutation
- The new individuals have their fitness evaluated and compete (possibly also with parents) for survival.
- Over time ***Natural selection*** causes a rise in the fitness of the population

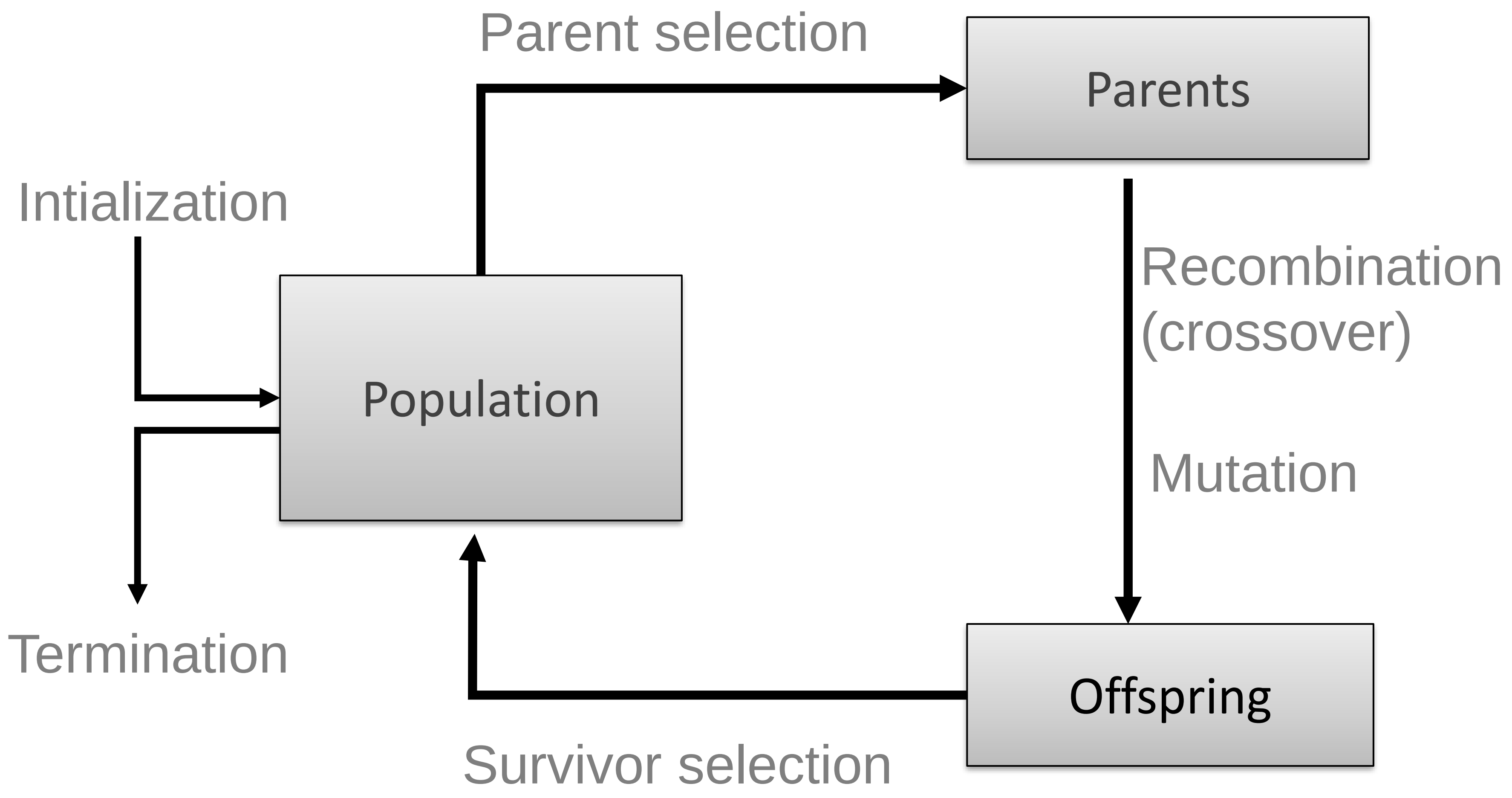
Recap of EC metaphor (2/2)

- EAs fall into the category of “generate and test” algorithms
- They are stochastic, population-based algorithms
- Variation operators (recombination and mutation) create the necessary diversity and thereby facilitate novelty
- Selection reduces diversity and acts as a force pushing quality

Chapter 3:
What is an Evolutionary Algorithm?

- Scheme of an EA
- Main EA components:
  - **Representation** / **evaluation** / population
  - Parent **selection** / survivor selection
  - Recombination / mutation
- Examples: eight-queens problem
- Typical EA behaviour
- EAs and global optimisation
- EC and neighbourhood search

Scheme of an EA:
General scheme of EAs



Scheme of an EA:
EA scheme in pseudo-code

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Scheme of an EA:
Common model of evolutionary processes

- Population of individuals
- Individuals have a fitness
- Variation operators: crossover, mutation
- Selection towards higher fitness
  - "survival of the fittest" and
  - "mating of the fittest"

Neo Darwinism:

Evolutionary progress towards higher life forms

=

Optimization according to some fitness-criterion
(optimization on a fitness landscape)

Scheme of an EA: Two pillars of evolution

There are two competing forces

Increasing population **diversity** by genetic operators

- mutation
- recombination

Push towards **novelty**

Decreasing population **diversity** by selection

- of parents
- of survivors

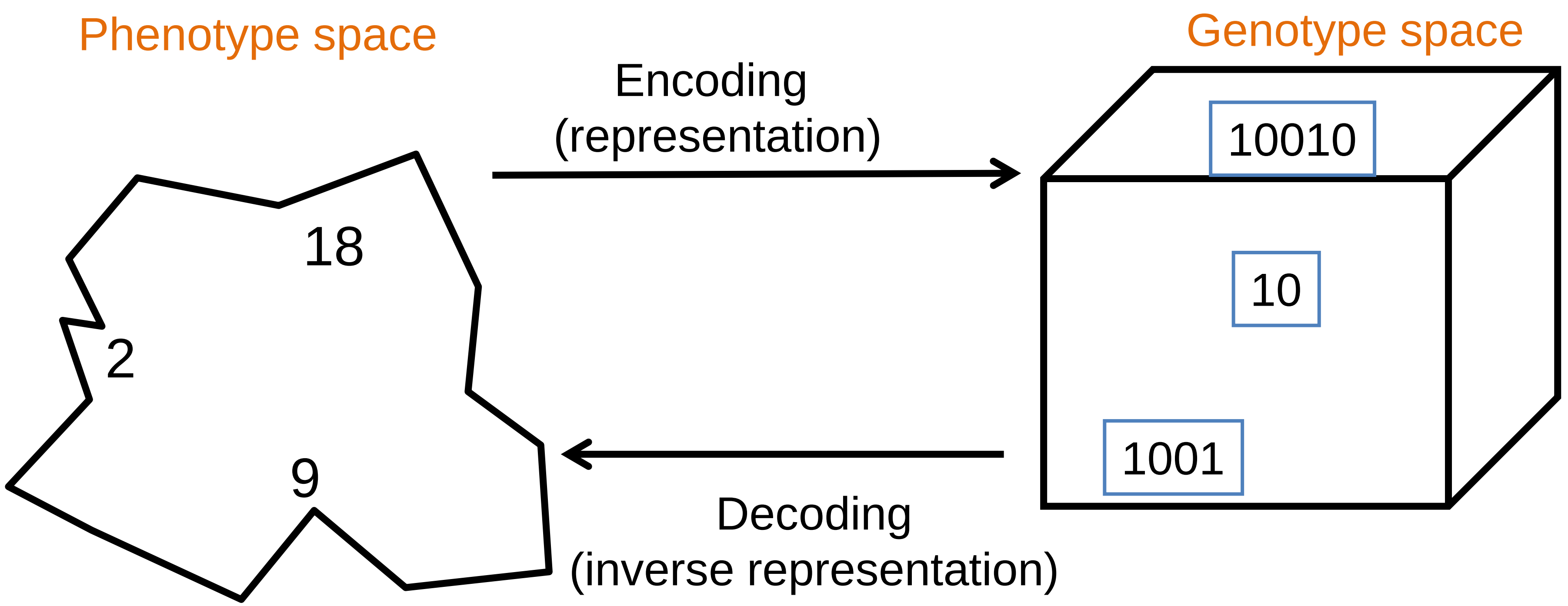
Push towards **quality**

Main EA components: Representation (1/2)

- Role: provides code for candidate solutions that can be manipulated by variation operators
- Leads to two levels of existence
  - phenotype: object in original problem context, the outside
  - genotype: code to denote that object, the inside (chromosome, “digital DNA”)
- Implies two mappings:
  - Encoding : phenotype=> genotype (not necessarily one to one)
  - Decoding : genotype=> phenotype (must be one to one)
- Chromosomes contain genes, which are in (usually fixed) positions called loci (sing. locus) and have a value (allele)

Main EA components: Representation (2/2)

Example: represent integer values by their binary code



In order to find the global optimum, every feasible solution must be represented in genotype space

Main EA components: Evaluation (fitness) function

- Role:
  - Represents the task to solve, the requirements to adapt to (can be seen as “the environment”)
  - Enables selection (provides basis for comparison)
  - e.g., some phenotypic traits are advantageous, desirable, e.g. big ears cool better, these traits are rewarded by more offspring that will expectedly carry the same trait
- A.k.a. *quality* function or *objective* function
- Assigns a single real-valued fitness to each phenotype which forms the basis for selection
  - So the more discrimination (different values) the better
- Typically we talk about fitness being maximised
  - Some problems may be best posed as minimisation problems, but conversion is trivial

Main EA components: Population (1/2)

- Role: holds the candidate solutions of the problem as individuals (genotypes)
- Formally, a population is a multiset of individuals, i.e. repetitions are possible
- Population is the basic unit of evolution, i.e., the population is evolving, not the individuals
- Selection operators act on population level
- Variation operators act on individual level

Main EA components: Population (2/2)

- Some sophisticated EAs also assert a spatial structure on the population e.g., a grid
- Selection operators usually take whole population into account i.e., reproductive probabilities are *relative* to *current* generation
- Diversity of a population refers to the number of different fitnesses / phenotypes / genotypes present (note: not the same thing)

Main EA components: Selection mechanism (1/3)

Role:

- Identifies individuals
  - to become parents
  - to survive
- Pushes population towards higher fitness
- Usually probabilistic
  - high quality solutions more likely to be selected than low quality
  - but not guaranteed
  - even worst in current population usually has non-zero probability of being selected
- This *stochastic* nature can aid escape from local optima

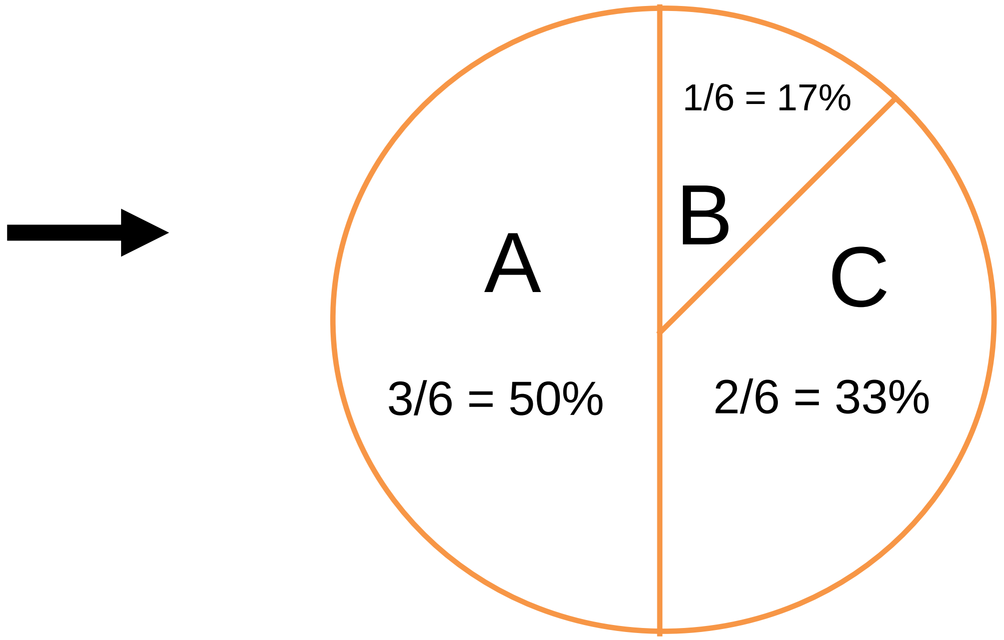
Main EA components: Selection mechanism (2/3)

Example: roulette wheel selection

fitness(A) = 3

fitness(B) = 1

fitness(C) = 2



In principle, any selection mechanism can be used for parent selection as well as for survivor selection

Main EA components: Selection mechanism (3/3)

- Survivor selection A.k.a. ***replacement***
- Most EAs use fixed population size so need a way of going from (parents + offspring) to next generation
- Often deterministic (while parent selection is usually stochastic)
  - Fitness based : e.g., rank parents + offspring and take best
  - Age based: make as many offspring as parents and delete all parents
- Sometimes a combination of stochastic and deterministic (elitism)

Main EA components: Variation operators

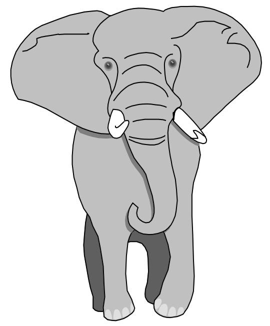
- Role: to generate new candidate solutions
- Usually divided into two types according to their arity (number of inputs):
  - Arity 1 : mutation operators
  - Arity >1 : recombination operators
  - Arity = 2 typically called crossover
  - Arity > 2 is formally possible, seldom used in EC
- There has been much debate about relative importance of recombination and mutation
  - Nowadays most EAs use both
  - Variation operators must match the given representation

Main EA components: Mutation (1/2)

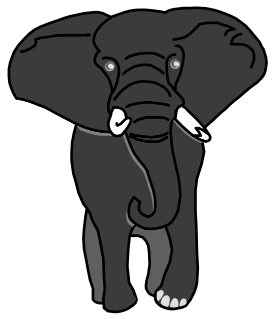
- Role: causes small, random variance
- Acts on one genotype and delivers another
- Element of randomness is essential and differentiates it from other unary heuristic operators
- Importance ascribed depends on representation and historical dialect:
  - Binary GAs – background operator responsible for preserving and introducing diversity
  - EP for FSM's / continuous variables – only search operator
  - GP – hardly used
- May guarantee connectedness of search space and hence convergence proofs

Main EA components: Mutation (2/2)

before 1 1 1 1 1 1 1 →



after 1 1 1 0 1 1 1 →

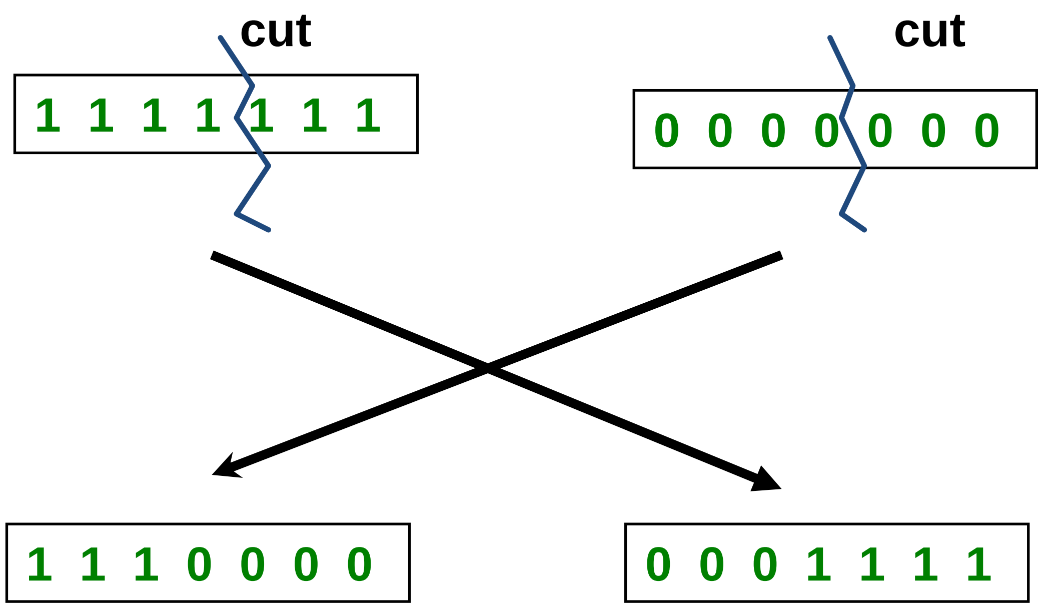


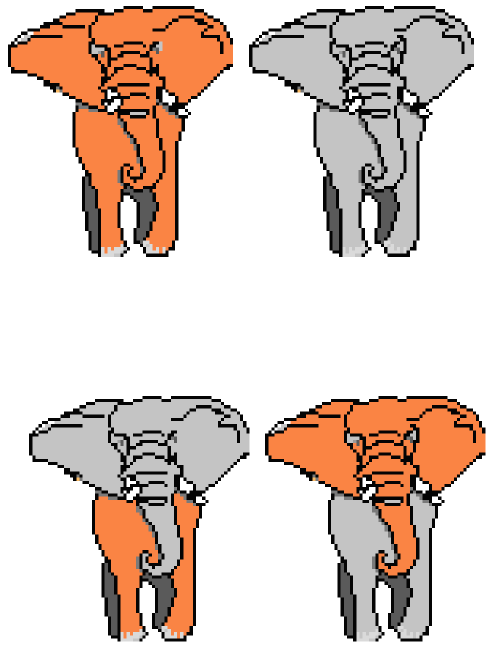
Main EA components: Recombination (1/2)

- Role: merges information from parents into offspring
- Choice of what information to merge is stochastic
- Most offspring may be worse, or the same as the parents
- Hope is that some are better by combining elements of genotypes that lead to good traits
- Principle has been used for millennia by breeders of plants and livestock

Main EA components: Recombination (2/2)

Parents





Offspring

Main EA components: Initialisation / Termination

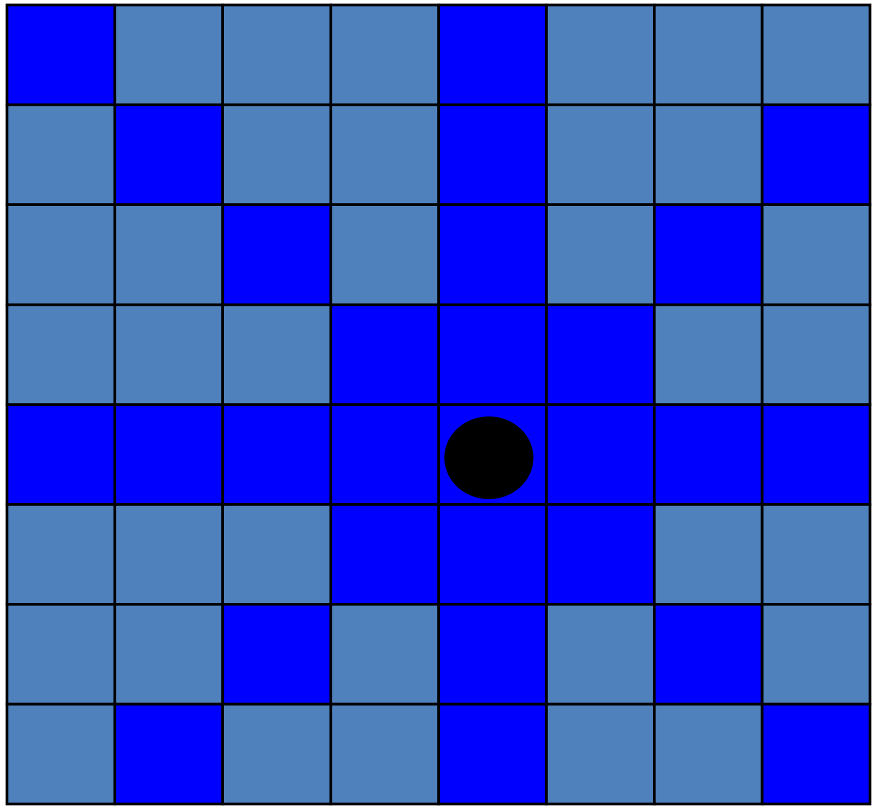
- Initialisation usually done at random,
  - Need to ensure even spread and mixture of possible allele values
  - Can include existing solutions, or use problem-specific heuristics, to “seed” the population

- Termination condition checked every generation
  - Reaching some (known/hoped for) fitness
  - Reaching some maximum allowed number of generations
  - Reaching some minimum level of diversity
  - Reaching some specified number of generations without fitness improvement

Main EA components:
What are the different types of EAs

- Historically different flavours of EAs have been associated with different data types to represent solutions
  - Binary strings : Genetic Algorithms
  - Real-valued vectors : Evolution Strategies
  - Finite state Machines: Evolutionary Programming
  - LISP trees: Genetic Programming
- These differences are largely irrelevant, best strategy
  - choose representation to suit problem
  - choose variation operators to suit representation
- Selection operators only use fitness and so are independent of representation

Example:
The 8-queens problem

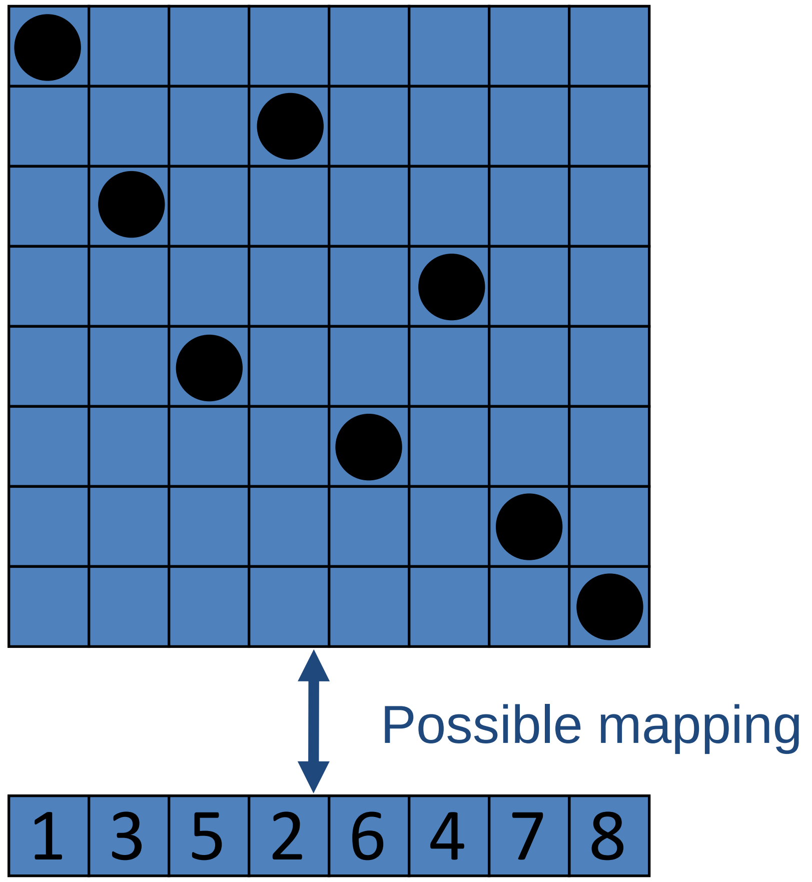


Place 8 queens on an 8x8 chessboard in
such a way that they cannot check each other

The 8-queens problem: Representation

Phenotype:
a board configuration

Genotype:
a permutation of
the numbers 1–8



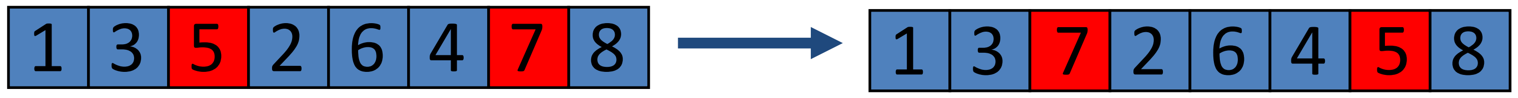
The 8-queens problem: Fitness evaluation

- Penalty of one queen: the number of queens she can check
- Penalty of a configuration: the sum of penalties of all queens
- Note: penalty is to be minimized
- Fitness of a configuration: inverse penalty to be maximized

The 8-queens problem: Mutation

Small variation in one permutation, e.g.:

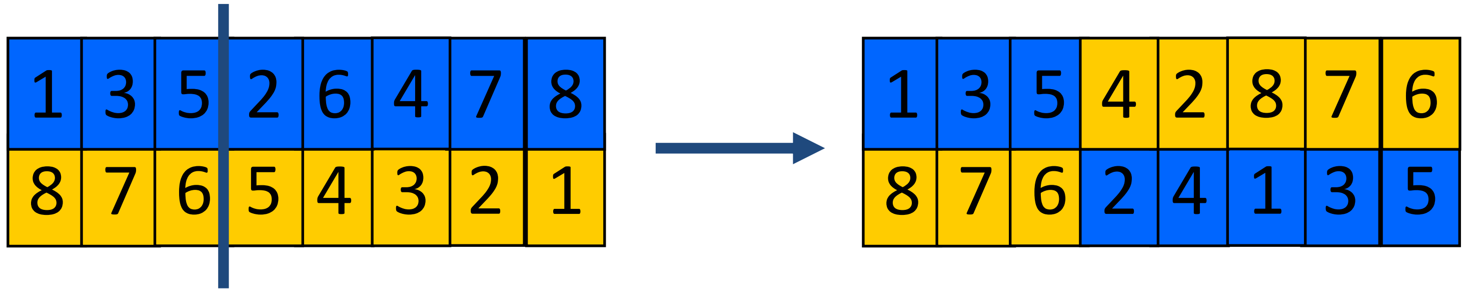
- swapping values of two randomly chosen positions,



The 8-queens problem: Recombination

Combining two permutations into two new permutations:

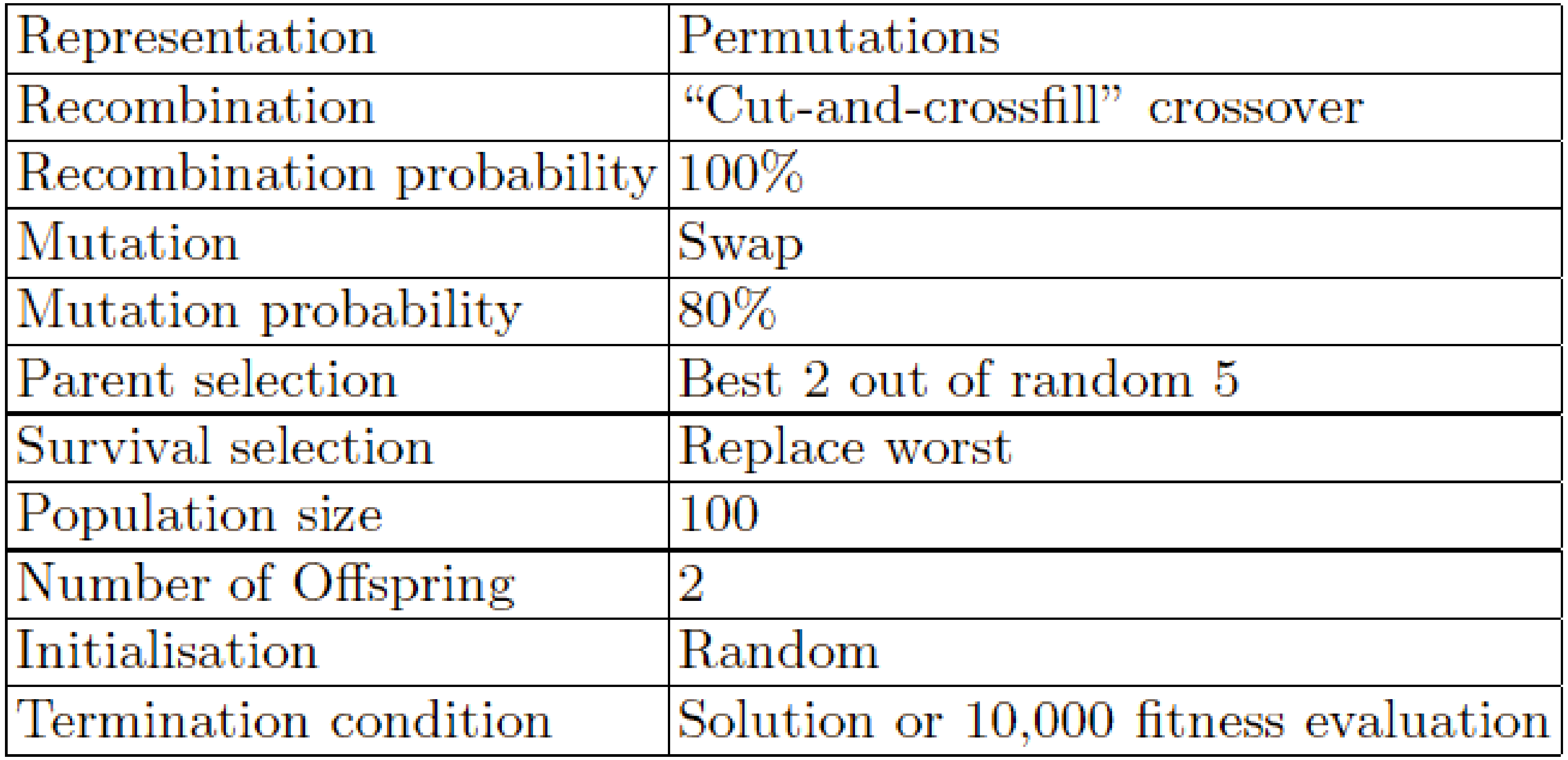
- choose random crossover point
- copy first parts into children
- create second part by inserting values from other parent:
    - in the order they appear there
    - beginning after crossover point
    - skipping values already in child



The 8-queens problem: Selection

- Parent selection:
  - Pick 5 parents and take best two to undergo crossover
- Survivor selection (replacement)
  - When inserting a new child into the population, choose an existing member to replace by:
  - sorting the whole population by decreasing fitness
  - enumerating this list from high to low
  - replacing the first with a fitness lower than the given child

The 8-queens problem: Summary

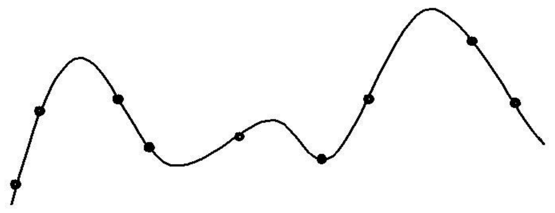


Representation	Permutations
Recombination	“Cut-and-crossfill” crossover
Recombination probability	100%
Mutation	Swap
Mutation probability	80%
Parent selection	Best 2 out of random 5
Survival selection	Replace worst
Population size	100
Number of Offspring	2
Initialisation	Random
Termination condition	Solution or 10,000 fitness evaluation

Note that is ***only one possible***
set of choices of operators and parameters

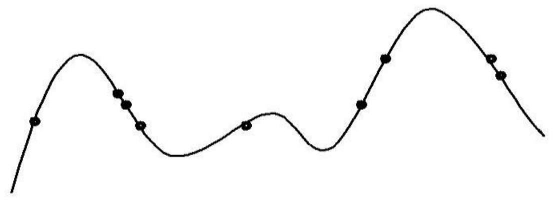
Typical EA behaviour: Stages

Stages in optimising on a 1-dimensional fitness landscape



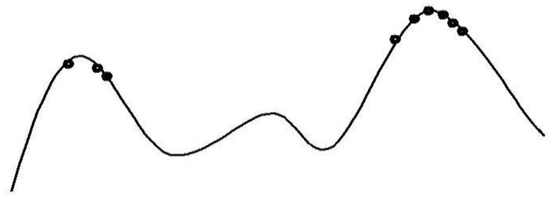
Early stage:

quasi-random population distribution



Mid-stage:

population arranged around/on hills

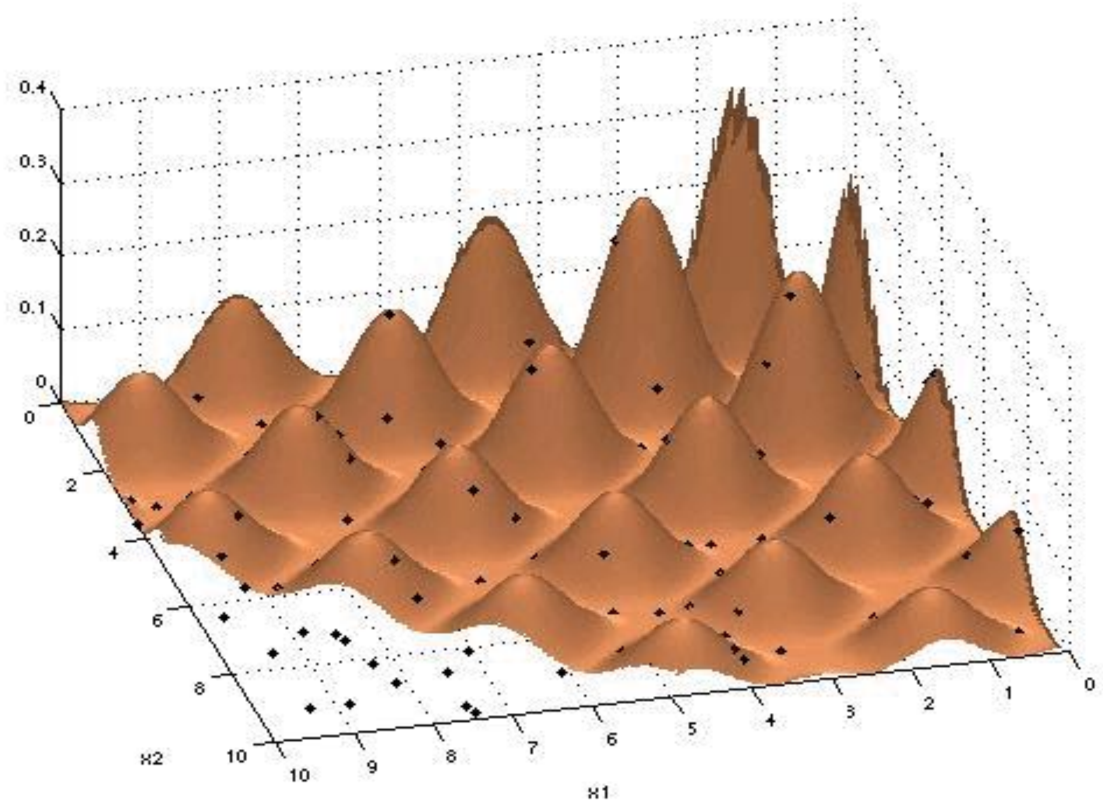


Late stage:

population concentrated on high hills

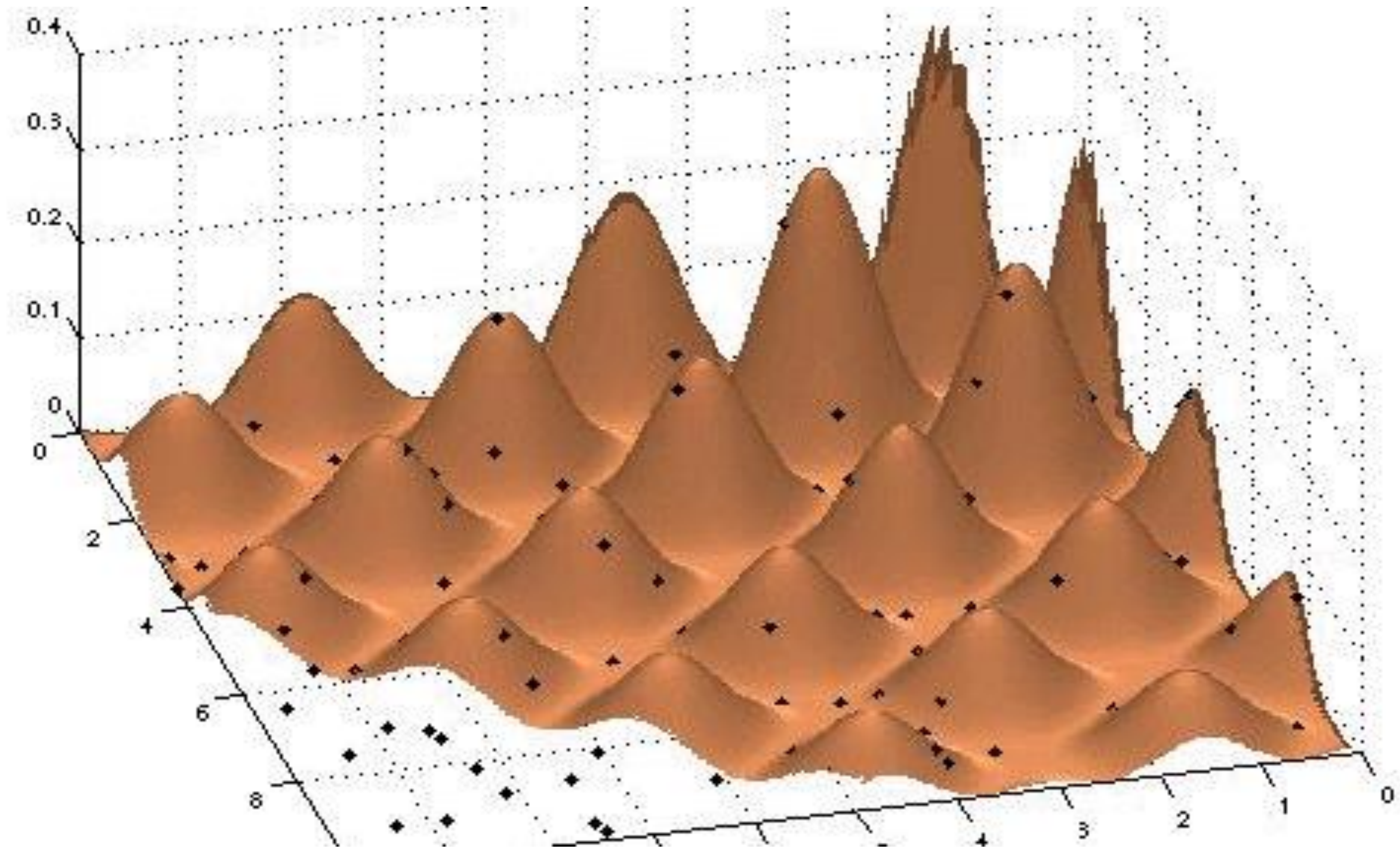
Typical EA behaviour: Working of an EA demo (1/2)

Searching a fitness landscape without "niching"

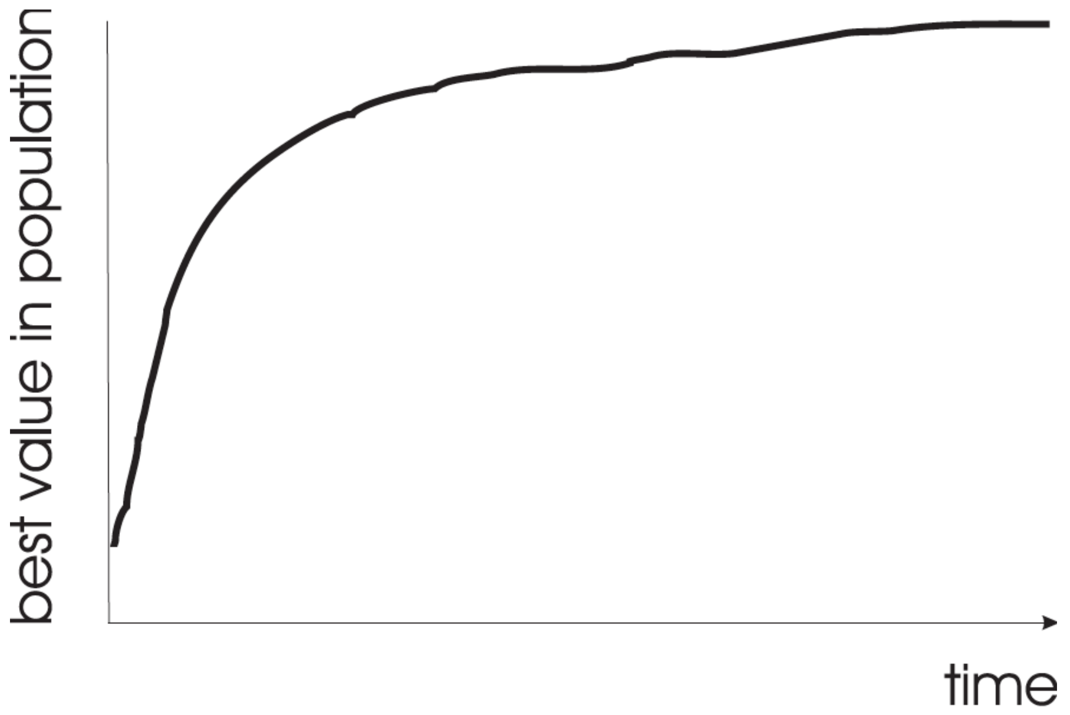


Typical EA behaviour: Working of an EA demo (2/2)

Searching a fitness landscape with “niching”



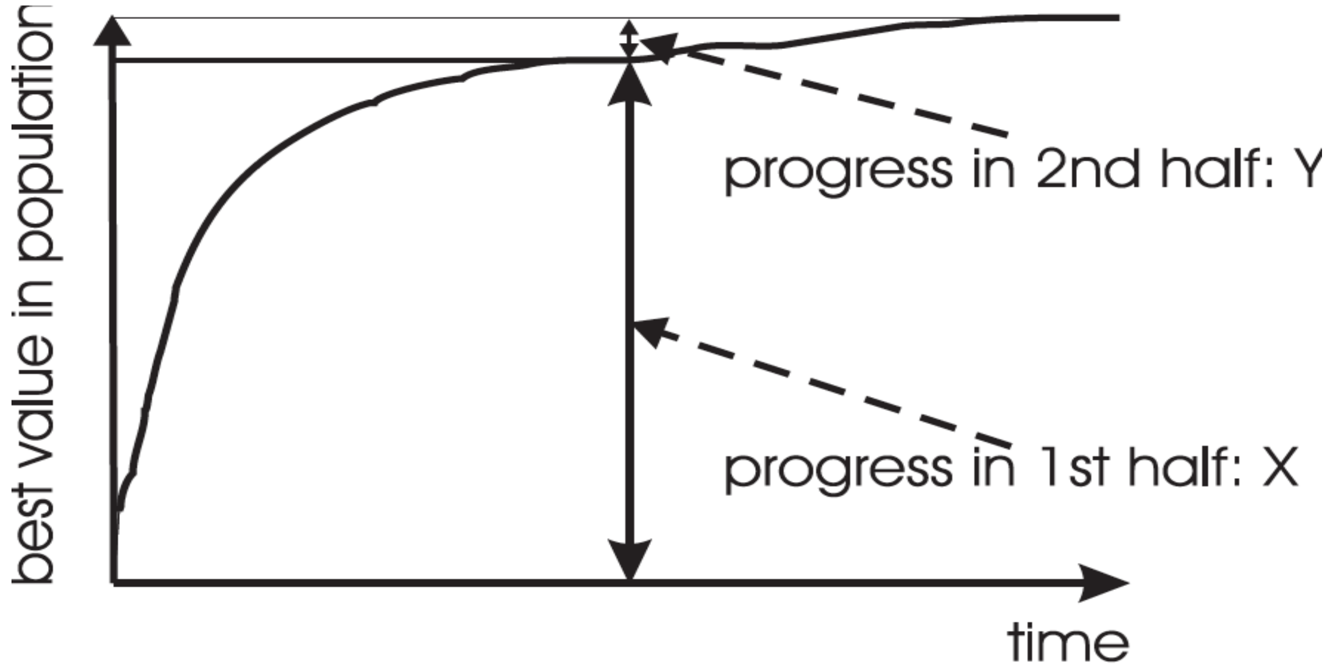
Typical EA behaviour:
Typical run: progression of fitness



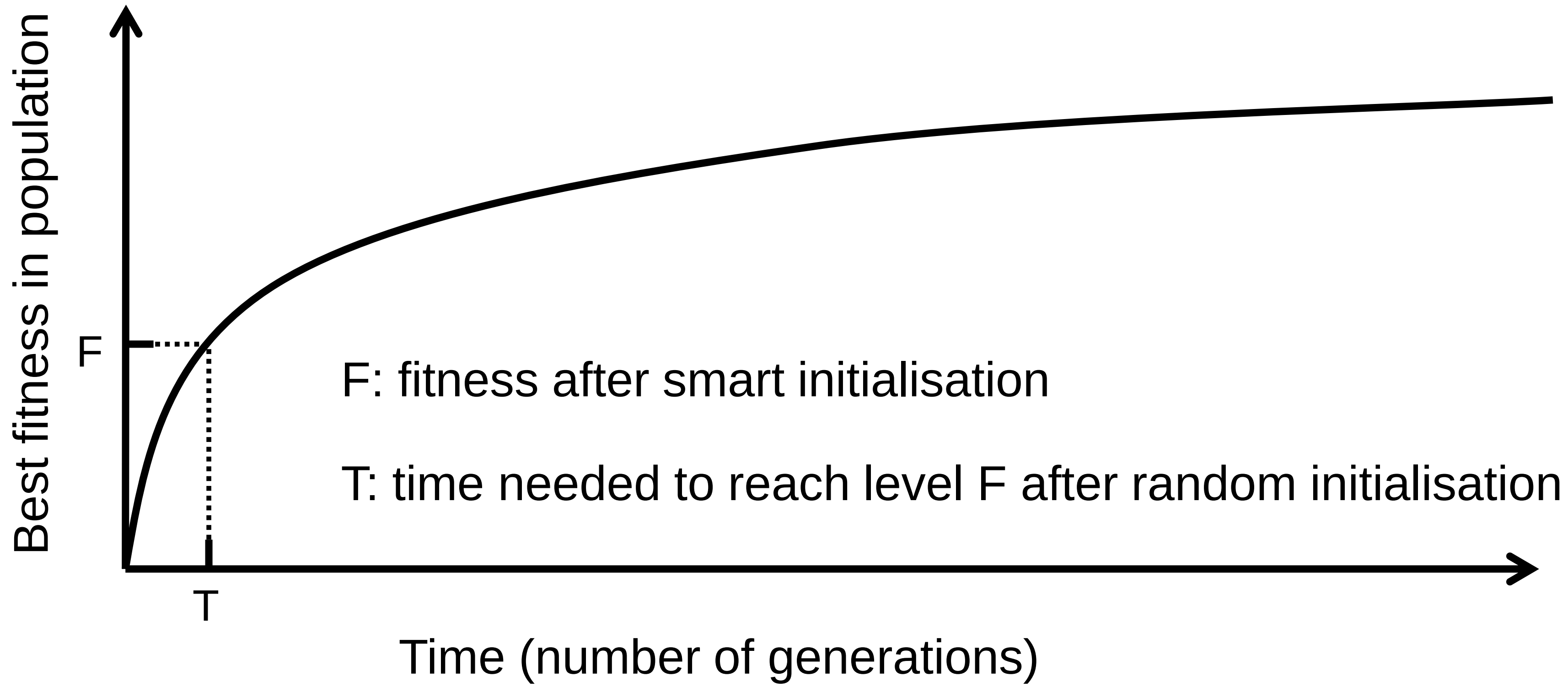
Typical run of an EA shows so-called "anytime behavior"

Typical EA behaviour: Are long runs beneficial?

- Answer:
  - It depends on how much you want the last bit of progress
  - May be better to do more short runs



Typical EA behaviour:
Is it worth expending effort on smart initialisation?

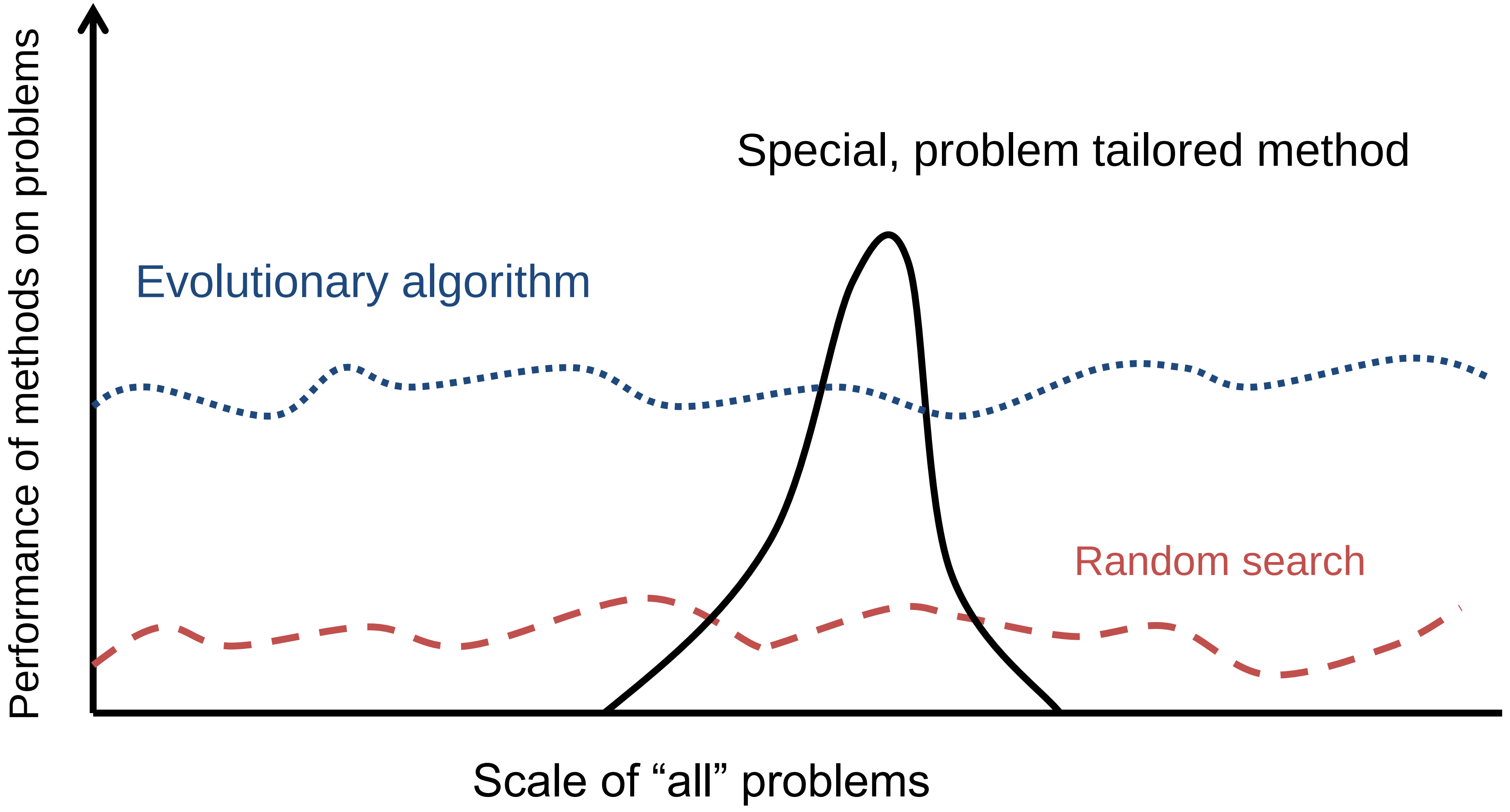


- Answer: it depends.
  - Possibly good, if good solutions/methods exist.
  - Care is needed, see chapter/lecture on hybridisation.

Typical EA behaviour: Evolutionary Algorithms in context

- There are many views on the use of EAs as robust problem solving tools
- For most problems a problem-specific tool may:
  - perform better than a generic search algorithm on most instances,
  - have limited utility,
  - not do well on all instances
- Goal is to provide robust tools that provide:
  - evenly good performance
  - over a range of problems and instances

Typical EA behaviour:
EAs as problem solvers: Goldberg view (1989)



Typical EA behaviour: EAs and domain knowledge

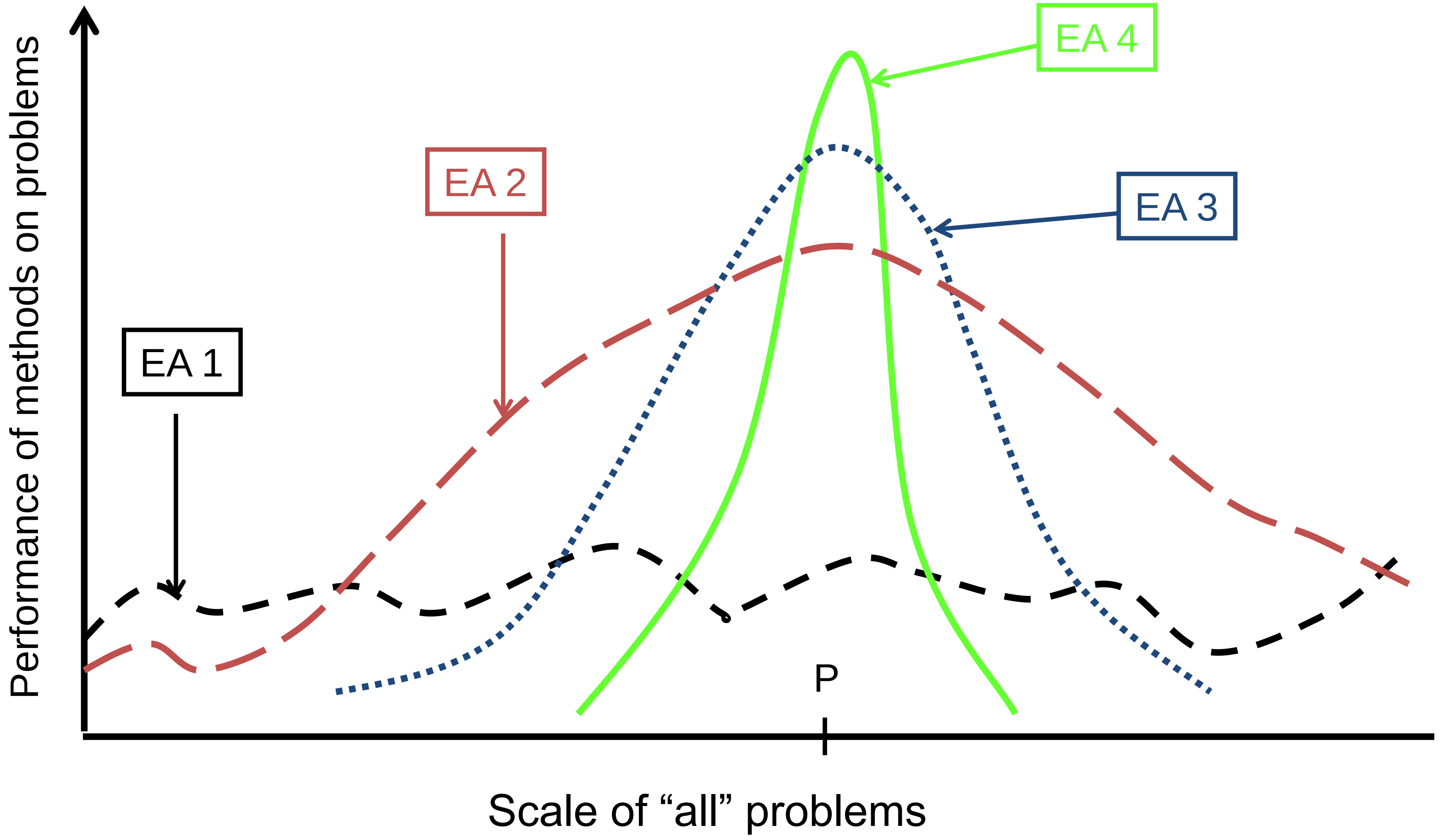
- Trend in the 90's:

  adding problem specific knowledge to EAs

  (special variation operators, repair, etc)
- Result: EA performance curve "deformation":
  - better on problems of the given type
  - worse on problems different from given type
  - amount of added knowledge is variable

- Recent theory suggests the search for an "all-purpose" algorithm may be fruitless

Typical EA behaviour:
EAs as problem solvers: Michalewicz view (1996)



EC and global optimisation

- Global Optimisation: search for finding best solution x^* out of some fixed set S
- Deterministic approaches
  - e.g. box decomposition (branch and bound etc)
  - Guarantee to find x^* ,
  - May have bounds on runtime, usually super-polynomial
- Heuristic Approaches (generate and test)
  - rules for deciding which $x \in S$ to generate next
  - no guarantees that best solutions found are globally optimal
  - no bounds on runtime

- “I don’t care if it works as long as it converges”

  vs.

- “I don’t care if it converges as long as it works”

EC and neighbourhood search

- Many heuristics impose a neighbourhood structure on *S*
- Such heuristics may guarantee that best point found is *locally optimal* e.g. Hill-Climbers:
  - **But** problems often exhibit many local optima
  - Often very quick to identify good solutions
- EAs are distinguished by:
  - Use of population,
  - Use of multiple, stochastic search operators
  - Especially variation operators with arity >1
  - Stochastic selection

- Question: what is the neighbourhood in an EA?

Evolutionary Computation

Laboratorio di Programmazione Evoluta e Competitiva

2024

Angelo Gargantini

(8h)

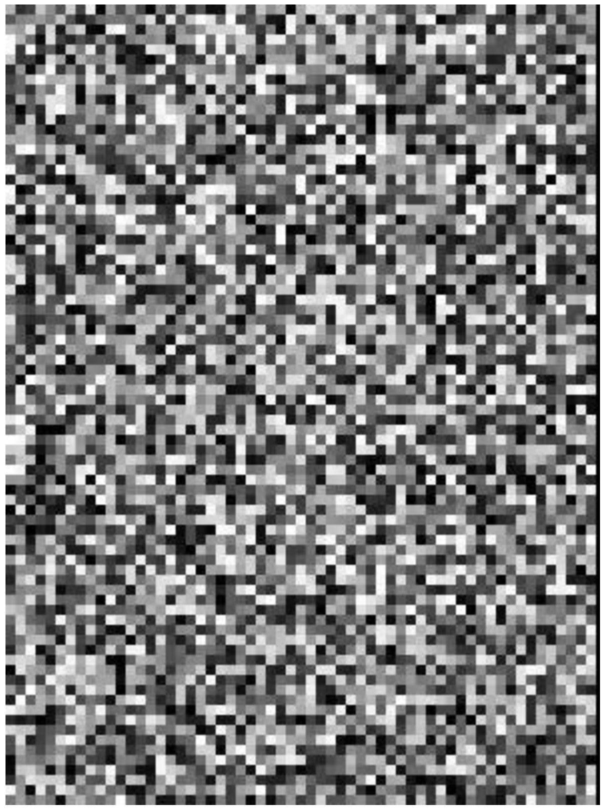
Contenuto: EC

- **evolutionary computation** is a family of algorithms for global optimization inspired by biological evolution. In technical terms, they are a family of population-based trial and error problem solvers with a metaheuristic or stochastic optimization character.

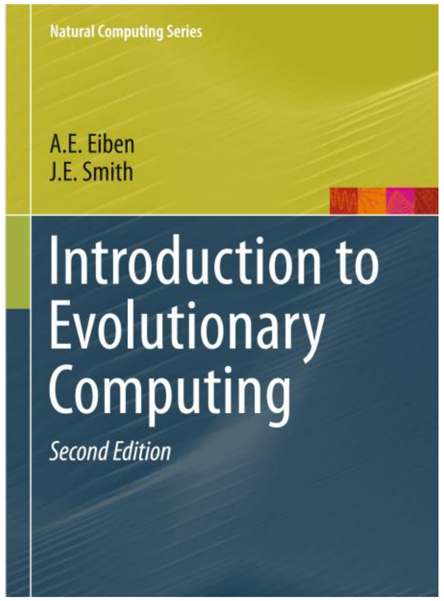
- Basic concepts of EC and its practical use in several problems

EC in brief

- In evolutionary computation, 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. As a result, the population will gradually evolve to increase in fitness, in this case the chosen fitness function of the algorithm.



Due pilastri

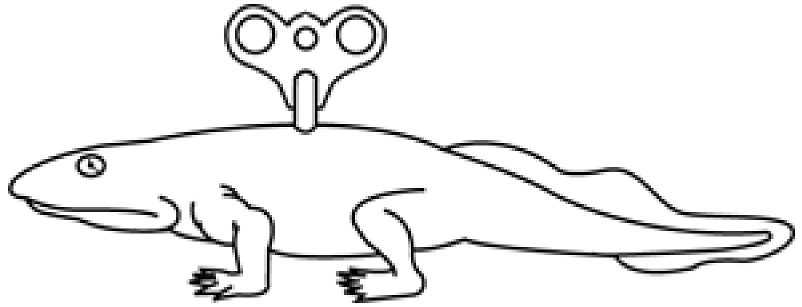

Natural Computing Series

A.E. Eiben
J.E. Smith

Introduction to Evolutionary Computing

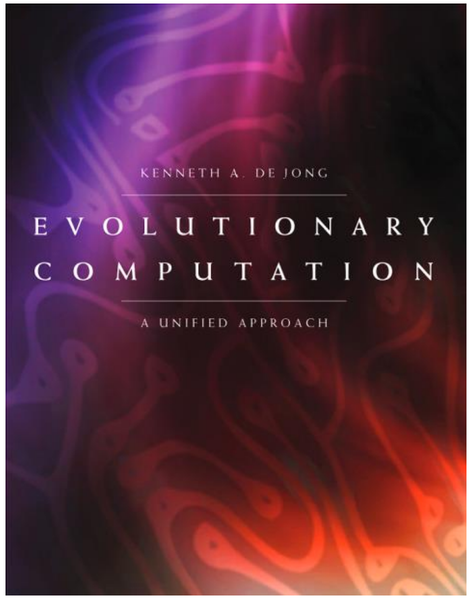
Second Edition

Parte teorica
Concetti come popolazione, selezione, rappresentazione, gene…..



Parte pratica
Evolutionary algorithms con watchmaker
https://watchmaker.uncommons.org/

Se volete saperne di più



Altro libro molto completo
Evolutionary Computation: A Unified Approach
Kenneth A. De Jong

Altra libreria in Java
https://jenetics.io/
Jenetics is a Genetic Algorithm, Evolutionary Algorithm, Grammatical Evolution, Genetic Programming, and Multi-objective Optimization library, written in modern-day Java.

DISTRIBUTED EVOLUTIONARY ALGORITHMS IN PYTHON

Mondo python

https://github.com/deap/deap
Uso avanzato con Python:
https://pymoo.org/

Esame

- Esercizio da implementare
  - 1h
  - In laboratorio
  - Con eclipse !!!

- Conta quanti punti? 5

Strumenti - programmazione

- Java
  - Inclusi i generici, anche <? super A>   ….
  - Classi anonime

- Maven
  - Per la libreria
- Eclipse (io e lab/esame)

Evolutionary computation in Java con watchmaker

Angelo Gargantini

LaPeC 2024

Strumenti che useremo

- Java
  - Perché????
    - Uso consistente dei tipi e dei generici
- Eclipse
  - Come editor – puoi usare quello che vuoi ma poi in lab useremo eclipse
- Maven
  - Per gestire le dipendenze della libreria (jar) – se vuoi puoi includere il jar
- watchmaker

Come installare eclipse

- Eclipse è semplicemente uno zip che si scarica e si unzippa

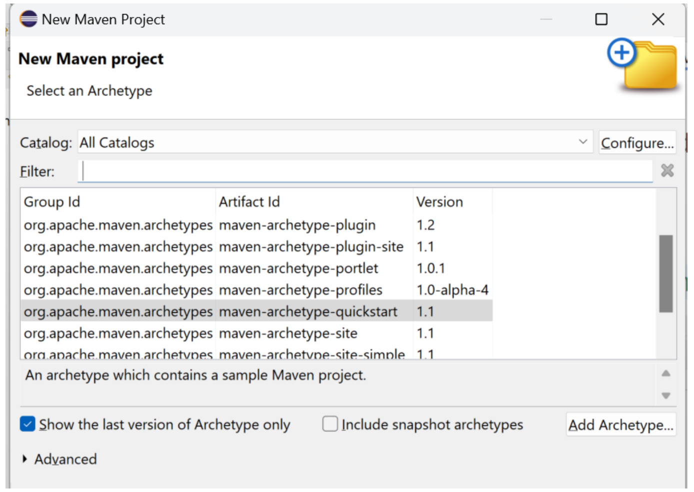
- Usiamo il package java:
  - https://www.eclipse.org/downloads/packages/release/2024-03/r/eclipse-ide-java-developers

- Scarica, unzippa, esegui eclipse.exe
- Contiene anche l’ultima versione di java (21)

Come creare un progetto maven in eclipse

- Per creare un progetto maven in eclipse puoi:

1. Nuovo progetto
2. Maven project
3. Selezionare quickstart come archetipo



New Maven project

Select an Archetype

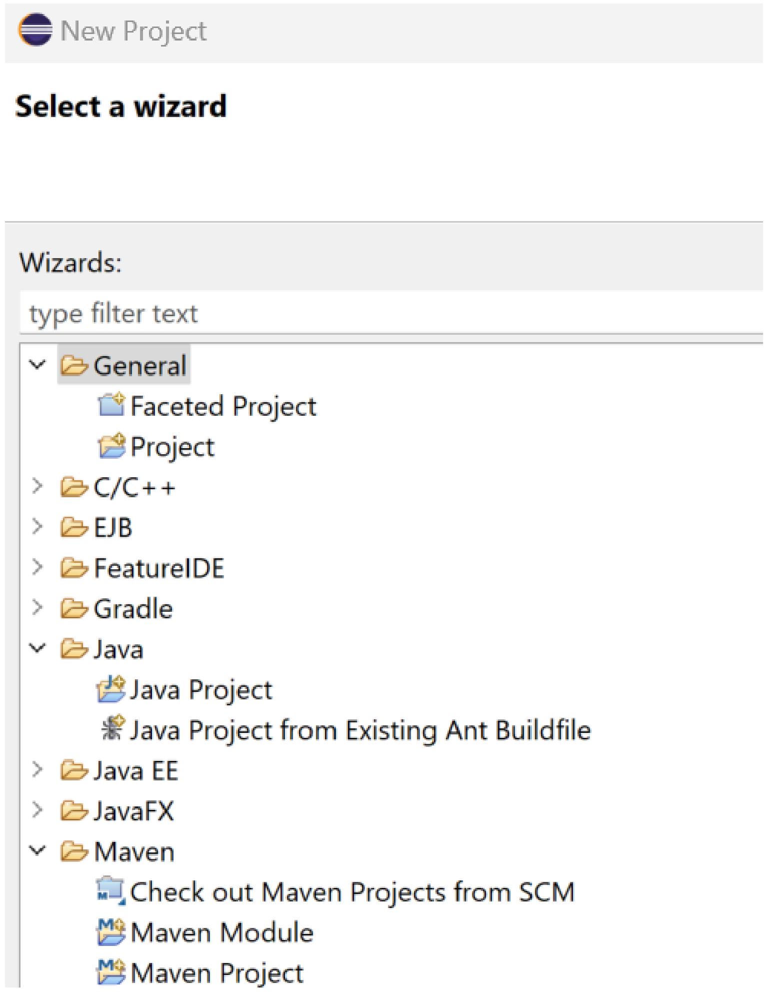
Catalog: All Catalogs

Filter:

Group Id	Artifact Id	Version
org.apache.maven.archetypes	maven-archetype-plugin	1.2
org.apache.maven.archetypes	maven-archetype-plugin-site	1.1
org.apache.maven.archetypes	maven-archetype-portlet	1.0.1
org.apache.maven.archetypes	maven-archetype-profiles	1.0-alpha-4
org.apache.maven.archetypes	maven-archetype-quickstart	1.1
org.apache.maven.archetypes	maven-archetype-site	1.1
org.apache.maven.archetypes	maven-archetype-site-simple	1.1

An archetype which contains a sample Maven project.

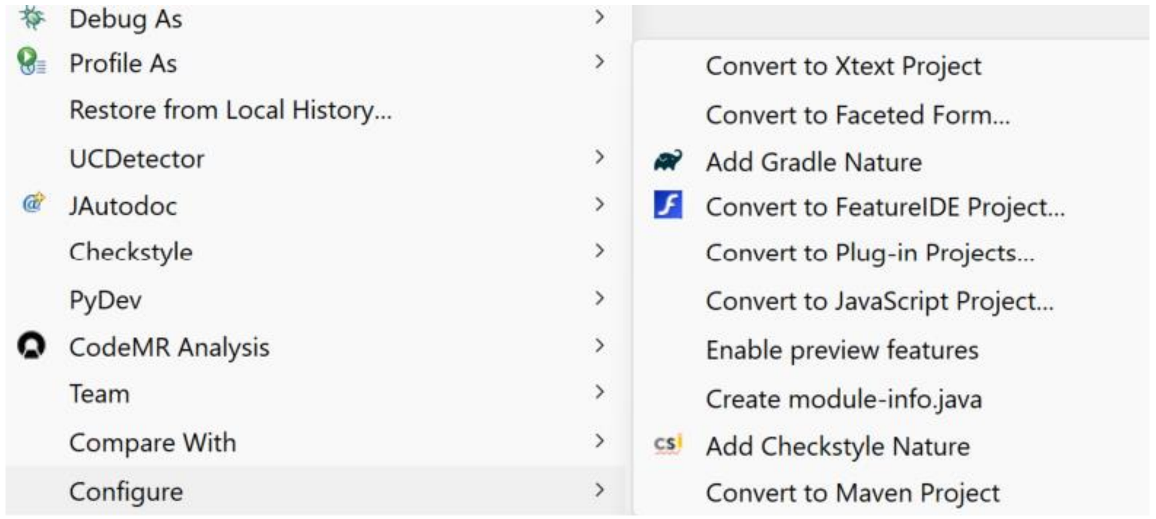
[✓] Show the last version of Archetype only
[ ] Include snapshot archetypes



- oppure

Convertire un progetto eclipse java a maven

- Puoi creare un progetto java normale e convertirlo in maven



Watchmaker

WATCHMAKER FRAMEWORK
FOR EVOLUTIONARY COMPUTATION

- https://watchmaker.uncommons.org/

- Repo github: https://github.com/dwdyer/watchmaker

- User Manul → tipo tutorial
- APIs → utili per sapere cosa fanno le classi

Come includere la libreria watchmaker

- Il modo più semplice è aggiungere nel pom le dipendenze:

```
<dependencies>
<dependency>
<groupId>org.uncommons.watchmaker</groupId>
<artifactId>watchmaker-framework</artifactId>
<version>0.7.1</version>
</dependency>
</dependencies>
```

- In alternativa si possono includere il jar (ma anche uncommons-maths-1.2.1. e google-collect-1.0.jar)

Altri tools per EC (java)

- https://jenetics.io/
  - Molto usato, basato molto sull’idea di Geni/Cromosomi…. Uso gli stream
- https://cs.gmu.edu/~eclab/projects/ecj/
  - ECJ is a research EC system written in Java. It was designed to be highly flexible, with nearly all classes (and all of their settings) dynamically determined at runtime by a user-provided parameter file. All structures in the system are arranged to be easily modifiable. Even so, the system was designed with an eye toward efficiency.
- https://github.com/jMetal/jMetal
  - Per l’ottimizzazione multi obiettivo

Generics

- Definizioni di classi generiche
  - class List<T>
- Instanzazione di T
  - List<Integer> list
  - (attenti non posso usare i tipi primitive – List<int> non è possibile)
- Creazione
  - list = new LinkedList<>();

- Java sa (in compilazione) che T è Integer (in esecuzione no, type erasure)

Bounds dei generici

- Posso mettere dei bound:
  - public <T extends Number> List<T> fooo(T[] a) …
- Posso usare wildcards
  - void foo(List<? extends Person> lp){…}

Classi anonime

- Posso dichiarare sottoclassi senza introdurre una nuova dichiarazione
  - Se ho una classe LibClass, posso fare (anche se LibClass ha metodo astratti)

1. class MyClass extends LibClass
   - Questa MyClass può ridefinire alcuni metodi di LibClass

2. oppure

- LibClass l = new LibClass ([argument-list]) { class-body }

- Anche con le interface (interface LibInterface)

  - LibInterface l = new LibInterface() { class-body }

- Dovrò implementare i metodi

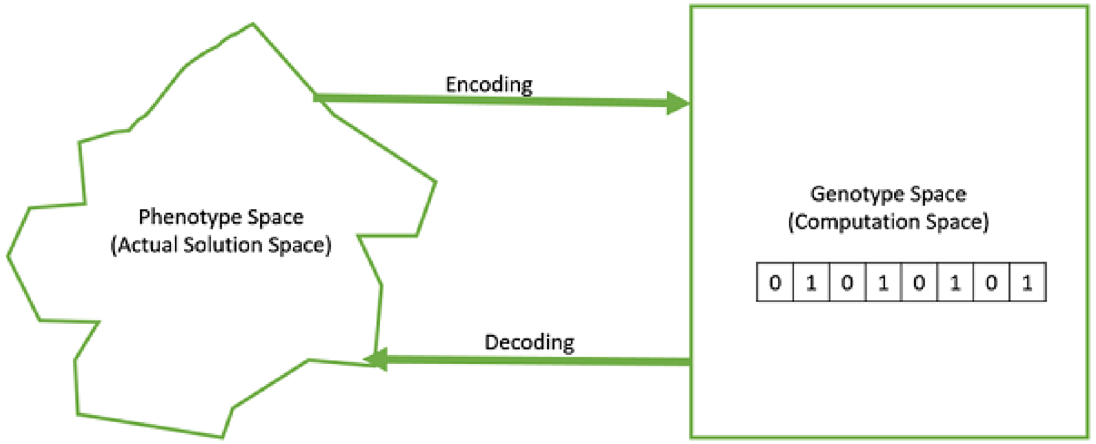
Example – HELLO WORD

- Suppose that we want to use evolution to generate a particular character string, for example "**HELLO WORLD**".
  - assume that we don't know how to create such a string and that evolution is the best approach available to us.
- Each **candidate solution** will be a string. We'll use a fixed-length representation so that each string is 11 characters long. Each character in a string will be one of the 27 valid characters (the upper case letters 'A' to 'Z' plus the space character).
- our population will be a set of Strings, each with 11 chars

Note - rappresentazione

- Popolazione: tutte le stringhe con 11 caratteri
- In questo caso Genotipo e Phenotipo praticamente coincidono

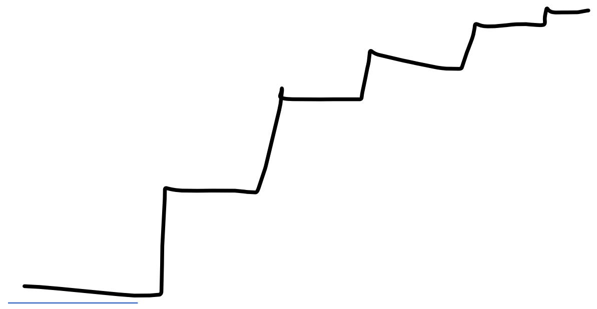
- Assicuriamoci che:
  1. Esiste un genotipo che rappresenta la soluzione ottima?
  2. Tutti i genotipi sono significativi



Note Fitness

- For the **fitness function** we'll use the simple approach of assigning a candidate solution one point for each position in the string that has the correct character. For the string "HELLO WORLD" this gives a maximum possible fitness score of 11 (the length of the string).

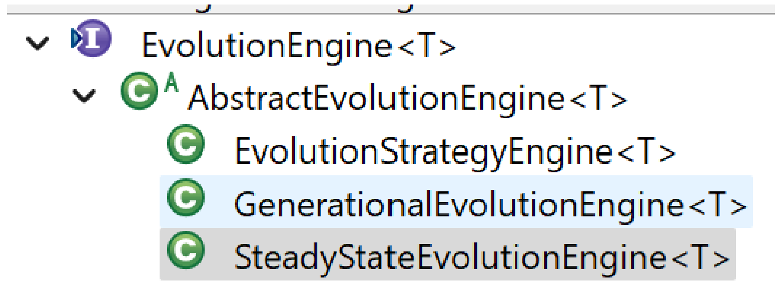
H	E	L	L	O		W	O	R	L	D	fitness
J	K	L	L	K	P	Z	O	K	P	U	3



EvolutionEngine

- The core component of the Watchmaker Framework is the **EvolutionEngine**. To write an evolutionary program using the Watchmaker Framework you have to instantiate an **EvolutionEngine** and invoke one of its evolve or evolvePopulation methods.
- The framework provides multiple implementations of the EvolutionEngine interface, but the one that you will usually want to use is **GenerationalEvolutionEngine**. This is a general-purpose implementation of the evolutionary algorithm outline from chapter 1.

EvolutionEngine



EvolutionEngine<T>
AbstractEvolutionEngine<T>
EvolutionStrategyEngine<T>
GenerationalEvolutionEngine<T>
SteadyStateEvolutionEngine<T>

- An **EvolutionEngine** has a single generic type parameter that indicates the type of object that it can evolve. For the "Hello World" program we need to be able to evolve Java strings. Code that creates an engine that can evolve strings would look something like this:

```
CandidateFactory<String> candidateFactory = getCandidateFactory();
EvolutionaryOperator<String> evolutionaryOperator = …;
FitnessEvaluator<? super String> fitnessEvaluator = …;
SelectionStrategy<? super String> selectionStrategy = …;
Random rng = …;
EvolutionEngine<String> engine =
new GenerationalEvolutionEngine<>(candidateFactory, evolutionaryOperator, fitnessEvaluator,
selectionStrategy, rng);
```

Other objects

1. A Candidate Factory
2. An Evolutionary Operator
3. A Fitness Evaluator
4. A Selection Strategy
5. A Random Number Generator

5. Random

- The final dependency that must be satisfied in order to create an evolution engine is the **random number generator** (RNG).
- An ideal RNG is both fast and statistically random. We could use the standard Java RNG, java.util.Random, but its output is not as random as it should be. The other RNG in the standard library, java.security.SecureRandom is much better in this respect but can be slow.
- Fortunately, the Watchmaker Framework provides alternatives. The org.uncommons.maths.random.**MersenneTwisterRNG** random number generator is both fast and statistically sound. It is usually the best choice when creating an evolution engine.

1. Candidate Factory

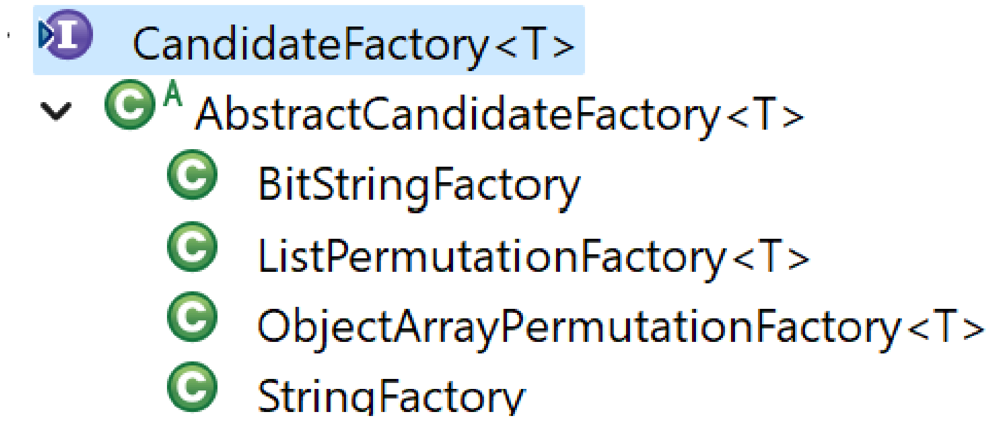
- The first object that needs to be plugged into the evolution engine is a candidate factory.
- Every evolutionary simulation must start with an initial population of candidate solutions and the **CandidateFactory** interface is the mechanism by which the evolution engine creates this population.

Esempio di popolazione iniziale

- The first task for the evolutionary algorithm is to randomly generate the initial population. We can use any size population that we choose.
- Typical EA population sizes can vary from tens to thousands of individuals. For this example we will use a population size of 10. After the initialisation of the population we might have the following candidates (fitness scores in brackets):

1. GERZUNFXCEN (1)
2. HSFDAHDMUYZ (1)
3. UQ IGARHGJN (0)
4. ZASIB WSUVP (2)
5. XIXROIUAZBH (1)
6. VDLGCWMBFYA (1)
7. SY YUHYRSEE (0)
8. EUSVBIVFHFK (0)
9. HHENRFZAMZH (1)
10. UJBBDFZPLCN (0)

StringFactory



CandidateFactory<T>
AbstractCandidateFactory<T>
BitStringFactory
ListPermutationFactory<T>
ObjectArrayPermutationFactory<T>
StringFactory

- Using a StringFactory

```
// Define the set of permitted characters (A-Z plus
space).
char[] chars = new char[27];
for (char c = 'A'; c <= 'Z'; c++) chars[c-'A'] = c;
chars[26] = ' ';
// Factory for random 11-character Strings.
CandidateFactory<String> candidateFactory = new
StringFactory(chars, 11);
```

Altri factory predefiniti

- `StringFactory`
  - General-purpose candidate factory for EAs that use a fixed-length String encoding Generates random strings of a fixed length from a given alphabet.
- `BitStringFactory`
- `ListPermutationFactory<T>` Generates random candidates from a set of elements. Each candidate is a random permutation of the full set of elements.

CandidateFactory in generale

In generale devo implementare il mio Factory di candidati di tipo T.
Basta implementare T `generateRandomCandidate`
Attenzione: ogni volta che chiamo il metodo meglio restituire candidati diversi ma possibilmente buoni (**the seed is strong**)
Ad esempio

```
// alternativa con una stringa random
class MyHelloWordInitilCandidateFactory extends AbstractCandidateFactory<String> {

        @Override
        public String generateRandomCandidate(Random rng) {
                String res = "";
                for (int i = 0; i < 11; i++)
                        res += (char) rng.nextInt();
                return res;
        }
}
```

Meglio o peggio del precedente Stringfactory?

Con classe anonima?

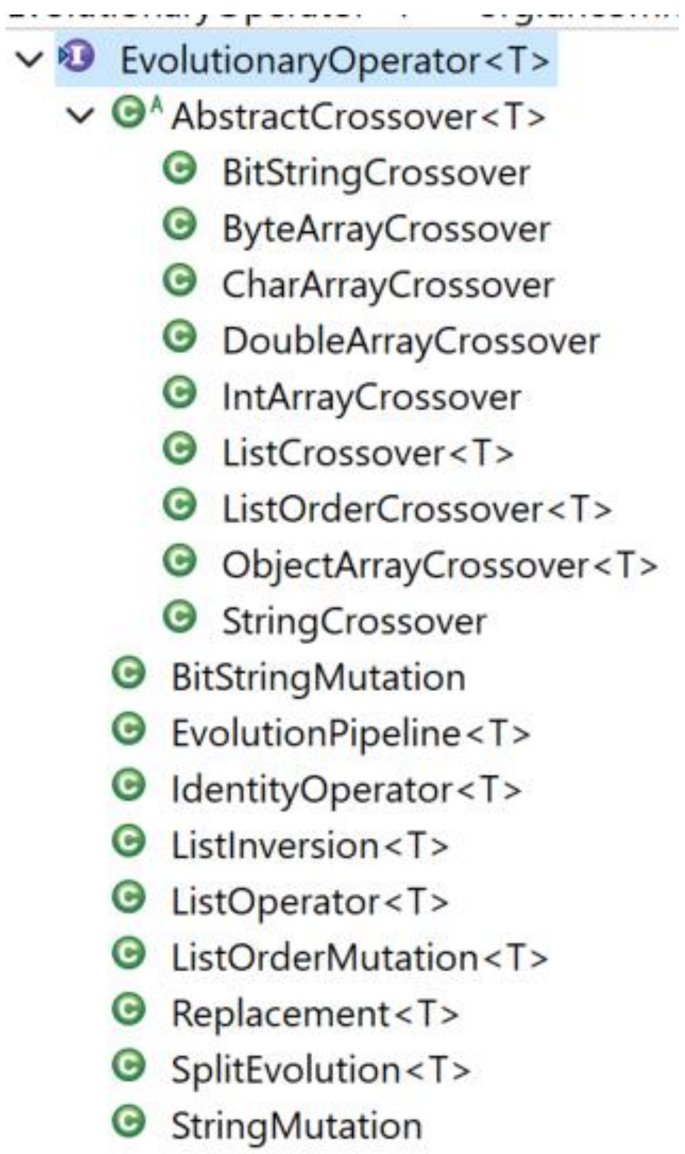
- Invece di
- `class MyHelloWordInitilCandidateFactory extends AbstractCandidateFactory<String>`

```
new AbstractCandidateFactory<String>(){
    // metodi da implementare
}
```

2. Evolutionary operators

- Evolutionary operators are the components that perform the actual evolution of a population. Cross-over is an evolutionary operator, as is mutation.
- evolutionary operators are defined in terms of the **EvolutionaryOperator** interface.
  - This interface declares a single method that takes a list of selected individuals and returns a list of evolved individuals.
  - Some operators (i.e. mutation) will process one individual at a time, whereas others will process individuals in groups (cross-over processes two individuals at a time).
- As with candidate factories, evolutionary operators have associated types that must be compatible with the type of the evolution engine that they are used with.
- the framework provides several ready-made operators for common types. These can be found in the org.uncommons.watchmaker.framework.operators package.

Evolutionary operators



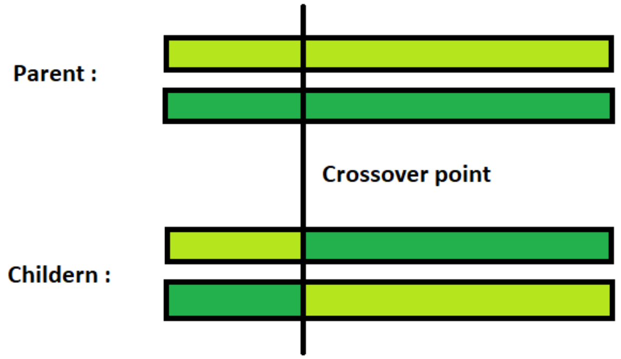
- EvolutionaryOperator<T>
  - AbstractCrossover<T>
    - BitStringCrossover
    - ByteArrayCrossover
    - CharArrayCrossover
    - DoubleArrayCrossover
    - IntArrayCrossover
    - ListCrossover<T>
    - ListOrderCrossover<T>
    - ObjectArrayCrossover<T>
    - StringCrossover
  - BitStringMutation
  - EvolutionPipeline<T>
  - IdentityOperator<T>
  - ListInversion<T>
  - ListOperator<T>
  - ListOrderMutation<T>
  - Replacement<T>
  - SplitEvolution<T>
  - StringMutation

- Many Evolutionary operators

- The cross-over and mutation operators that we need for our "Hello World" program are provided by the **StringCrossover** and **StringMutation** classes.

1. StringCrossover

- `new StringCrossover()`
- `* Variable-point (fixed or random) cross-over for String candidates.`
- `* This implementation assumes that all candidate Strings are the same`
- `* length. If they are not, an exception will be thrown at runtime.`



```
Arrays.asList("CIAO!", "HELLO", "CASSA")
→
[CASS!, CIAOA, HELLO]
[CILLO, HEAO!, CASSA]
[HELL!, CIAOO, CASSA]
[CASSO, HELLA, CIAO!]
[HELLA, CASSO, CIAO!]
[CILLO, HEAO!, CASSA]
[CIAO!, CASSA, HELLO]
[CAAO!, CISSA, HELLO]
[CALLO, HESSA, CIAO!]
[CASLO, HELSA, CIAO!]
[CILLO, HEAO!, CASSA]
[CIAO!, CASSA, HELLO]
[HASSA, CELLO, CIAO!]
[CILLO, HEAO!, CASSA]
[CASS!, CIAOA, HELLO]
[CELLO, HIAO!, CASSA]
```

2. StringMutation

```
new StringMutation(chars, new Probability(0.02))
```

```
* Creates a mutation operator that is applied with the given
* probability and draws its characters from the specified alphabet.
* @param alphabet The permitted values for each character in a string.
* @param mutationProbability The probability that a given character is changed.
```

```
char[] chars = {'A', 'B', 'C', 'D'};
StringMutation sm = new StringMutation(chars, new Probability(0.2));

Arrays.asList("CIAO!", "HELLO", "CASSA");→
[CIAOC, HELLO, CASSA] [CIAO!, HELLO, CACSA] [CCAO!, HCCLO, BASSA]
[CIAA!, HELCD, DASSA] [CAAO!, HELLO, CABSA] [DIBO!, HELBA, CASSD]
[CIAO!, CELLB, CASSA] [ADAO!, HAALO, CASSB] [CIAB!, HEALO, ABSSA]
```

Join evolutionary operators

- the evolution engine constructor only accepts a single evolutionary operator.
- So how can we use both cross-over and mutation? The answer is provided by the EvolutionPipeline operator. This is a compound evolutionary operator that chains together multiple operators of the same type.

```
List<EvolutionaryOperator<String>> operators = new LinkedList<EvolutionaryOperator<String>>();
operators.add(new StringCrossover());
operators.add(new StringMutation(chars, new Probability(0.02)));
EvolutionaryOperator<String> pipeline = new EvolutionPipeline<String>(operators);

A compound evolutionary operator that applies multiple operators (of the same type) in series.
By combining EvolutionPipeline operators with {@link SplitEvolution} operators,
elaborate evolutionary schemes can be constructed.
```

3. Fitness Evaluator

- a fitness function is written by implementing the **FitnessEvaluator** interface. The getFitness method of this interface takes a candidate solution and returns its **fitness score as a Java double**. The method actually takes two arguments, but we can ignore the second for now.
  - Ha anche un metodo
  - `boolean isNatural();`

```
Specifies whether this evaluator generates <i>natural</i> fitness scores or not.
Natural fitness scores are those in which the fittest individual in a population has
the highest fitness value. In this case the algorithm is attempting to maximise
fitness scores. There need not be a specified maximum possible value.
In contrast, <i>non-natural</i> fitness evaluation results in fitter individuals being
assigned lower scores than weaker individuals. In the case of non-natural fitness, the
algorithm is attempting to minimise fitness scores.

@return True if a high fitness score means a fitter candidate or false if a low
fitness score means a fitter candidate.
```

Fitness evaluator 1

- we'll use the simple approach of assigning a candidate solution one point for each position in the string that has the correct character.

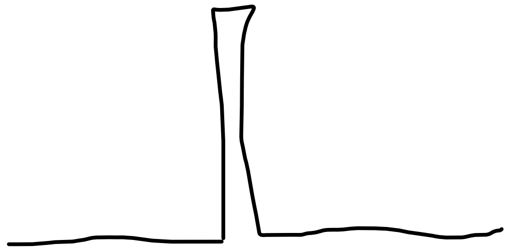
```
public class StringEvaluator implements FitnessEvaluator<String> {
 private final String targetString = "HELLO WORLD";
 public double getFitness(String candidate, List<? extends String> population) {
  int matches = 0;
   for (int i = 0; i < candidate.length(); i++)
        if (candidate.charAt(i) == targetString.charAt(i)) ++matches;
  return matches;
 }
 public boolean isNatural() { return true; }
}
```

nota: questa ignora la popolazione

Example

With strings generated by the `CandidateFactory`

Altre valutatori della fitness????



1. 0 se sbagliata 100 se giusta?

2. La somma delle differenze tra il carattere in posizione i-esima dalla candidata e il carattere alla posizione i-esima del target????

```
IWDHSTQFRSJ-->1.0
YNZSNZQS CI-->0.0
DQZGLQUDWAJ-->0.0
HKTEAYUINQI-->1.0
ZPA QUJRKZG-->0.0
UAJFBBKVIBZ-->0.0
VHCOEPCICBM-->0.0
FXKZEMTNCDK-->0.0
HOAJJBZPLWR-->1.0
PARBXVDRSNB-->0.0
JFGSNYNBRDF-->1.0
NCDJHZOYNDU-->0.0
DRWMAOXXCPL-->0.0
UYIHJMWJFS -->0.0
EBFENQROBCK-->1.0
NF UZHDBGIR-->0.0
LXWOGIFKJAJ-->0.0
RWPQWXOCILU-->1.0
BCLXHAQOPGT-->2.0
HIYFMOWJNJP-->2.0
VCDYGDJYWVO-->0.0
GPYWDMHLPCX-->0.0
FARMTLRAWQD-->1.0
RCTXRUUYNXI-->0.0
EKIRFGCJWQ -->1.0
UXC LPMXICI-->0.0
HGBRQ JQSUB-->2.0
```

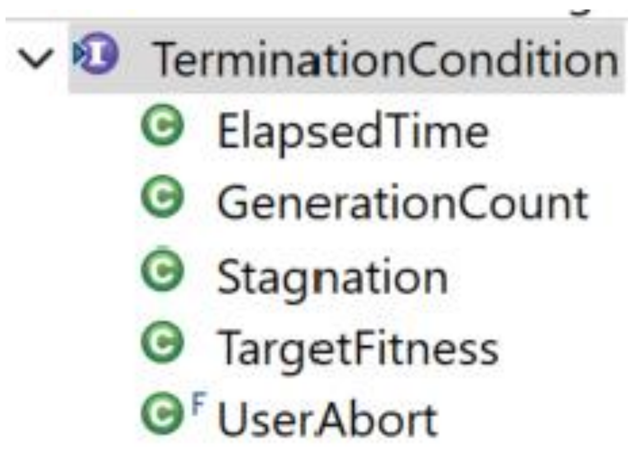
4. Selection Strategy

- Selection is a key ingredient in any evolutionary algorithm. It's what determines which individuals survive to reproduce and which are discarded.
- Some selection strategies work better than others for certain problems. Often a little trial-and-error is required to pick the best option.
- for now we will just create an instance of the **RouletteWheelSelection** class and use that for our "Hello World" application.
- **Roulette wheel selection is the most common type of fitness-proportionate selection. It gives all individuals a chance of being selected but favours the fitter individuals since an individual's selection probability is derived from its fitness score.**

Executing the evalution

- We have now to perform the evolution.
- For that we need to call the **evolve** method
  - takes three parameters.
  - The first is the **size of the population**. This is the number of candidate solutions that exist at any time. A bigger population will often result in a satisfactory solution being found in fewer generations. On the other hand, the processing of each generation will take longer because there are more individuals to deal with. For the "Hello World" program, a population size of 10 is fine.
  - The second parameter is concerned with **elitism**. For now, just use a value of zero.
  - The final varargs parameter specifies one or more **termination conditions**.

Termination condition



```
ElapsedTime
Terminates evolution after a pre-determined period of time has elapsed.
GenerationCount
Terminates evolution after a set number of generations have passed.
Stagnation
A {@link TerminationCondition} that halts evolution if no improvement
in fitness is observed within a specified number of generations.
TargetFitness
Terminates evolution once at least one candidate in the population has
equalled or bettered a pre-determined fitness score.
UserAbort
{@link TerminationCondition} implementation that allows for user-initiated
termination of an evolutionary algorithm. This condition can be used, for
instance, to provide a button on a GUI that terminates execution.
```

```
TerminationCondition stop = new GenerationCount(100);
String res = engine.evolve(100, 0, stop );
```

Proviamo!!!

Evolution Observers

- The **evolve** method gives no hints as to its evolutionary nature. We can make the program more interesting by adding an EvolutionObserver to report on the progress of the evolution at the end of each generation.
- Add the following code to your program before the call to the evolve method (note the use of anonymous class)

```
engine.addEvolutionObserver(new EvolutionObserver<String>() {
        public void populationUpdate(PopulationData<? extends String> data) {
        System.out.printf("Generation %d: %s\n", data.getGenerationNumber(),
        data.getBestCandidate());
        }
});
```

Esercizi

Esercizio 1 - Prova l’esercizio hello world

1. Prova hello world e vedi se funziona
2. Se invece di “hello world” prendessi una stringa molto lunga (divina commedia)?
3. Prova a cambiare la condizione di terminazione – ad esempio staganazione. Termina prima?
4. Se cambiassimo la fitness
    - Se canidato uguale a “HELLO WORLD” allora 1 altrimenti 0 ?

Esercizio 2

- Generare una stringa con lunghezza N di tua scelta che sia **palindroma**
  - Fissa la lunghezza
  - Usa pure sia mutation che crossover
  - Definisci una fitness

Evolutionary computation in Java
con watchmaker

Selection Strategies & Elitism

Angelo Gargantini

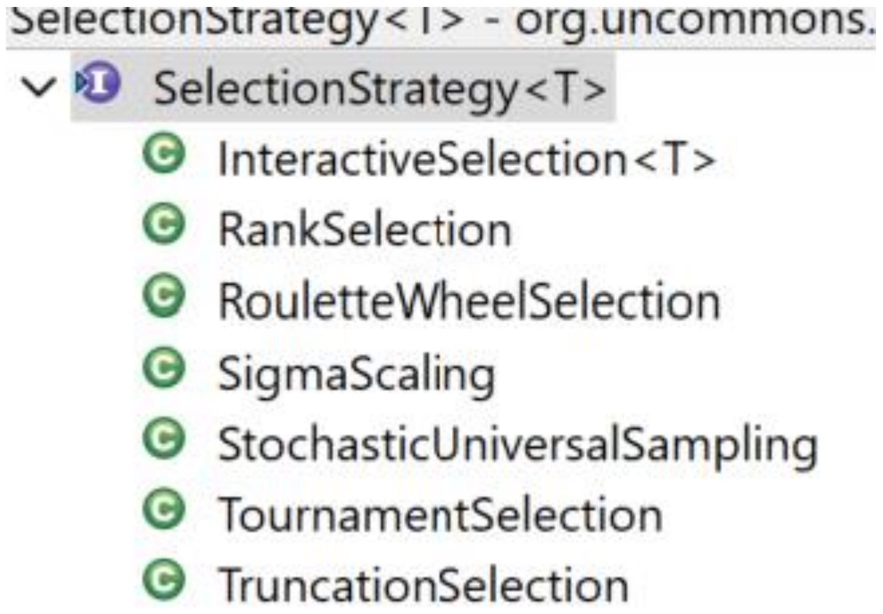
LaPeC 2024

Selection strategies

- **Selection** is an important part of an evolutionary algorithm. Without selection directing the algorithm towards fitter solutions there would be no progress.

- Selection must favour fitter candidates over weaker candidates but beyond that there are no fixed rules. Furthermore, there is no one strategy that is best for all problems.

Selection strategies



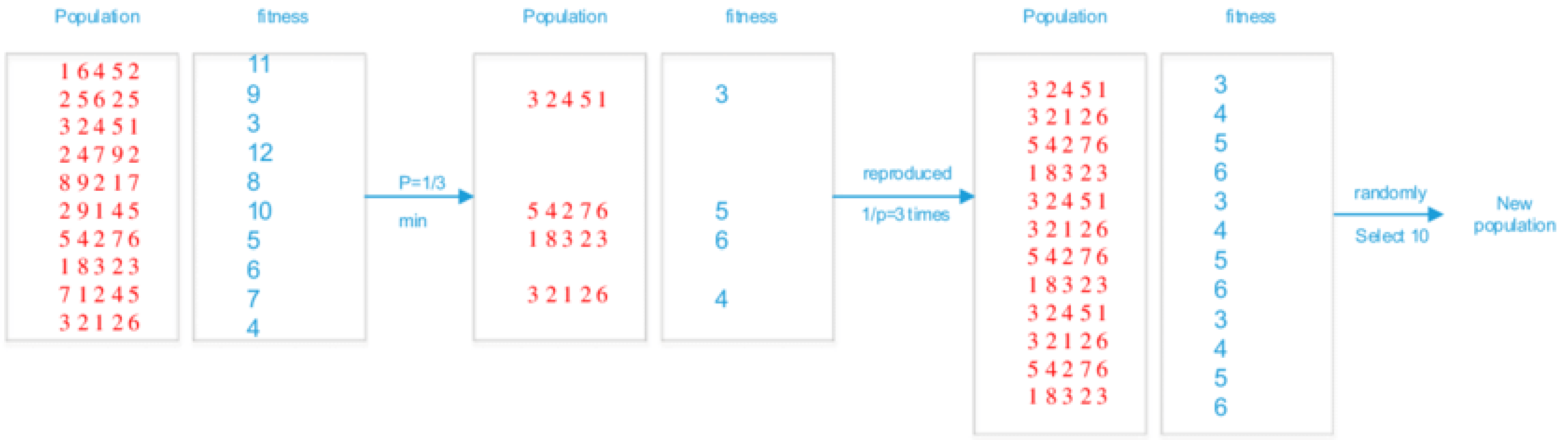
SelectionStrategy<T> - org.uncommons.

SelectionStrategy<T>
- InteractiveSelection<T>
- RankSelection
- RouletteWheelSelection
- SigmaScaling
- StochasticUniversalSampling
- TournamentSelection
- TruncationSelection

Truncation Selection

- Truncation selection is the simplest and arguably least useful selection strategy. Truncation selection simply retains the fittest x% of the population. These fittest individuals are duplicated so that the population size is maintained.

For example, we might select the fittest 25% from a population of 100 individuals. In this case we would create four copies of each of the 25 candidates in order to maintain a population of 100 individuals.



Fitness non naturale

In Hello world – evolve più rapidamente!!

```
selectionStrategy = new TruncationSelection(0.2);
…
Generation 0: MH JO WZDSM
Generation 1: MH JO EOAPD
Generation 2: VEILJ EOAPD
Generation 3: RH JO WJRLD
Generation 4: VELYO WJRLD
Generation 5: VELYO WORLD
Generation 6: VELYO WORLD
Generation 7: HELYO WORLD
Generation 8: AELLO WORLD
Generation 9: HELLO WORLD
```

- Attenzione: potrebbe convergere prematuramente (troppa exploitation e poca exploration)

Fitness-Proportionate Selection

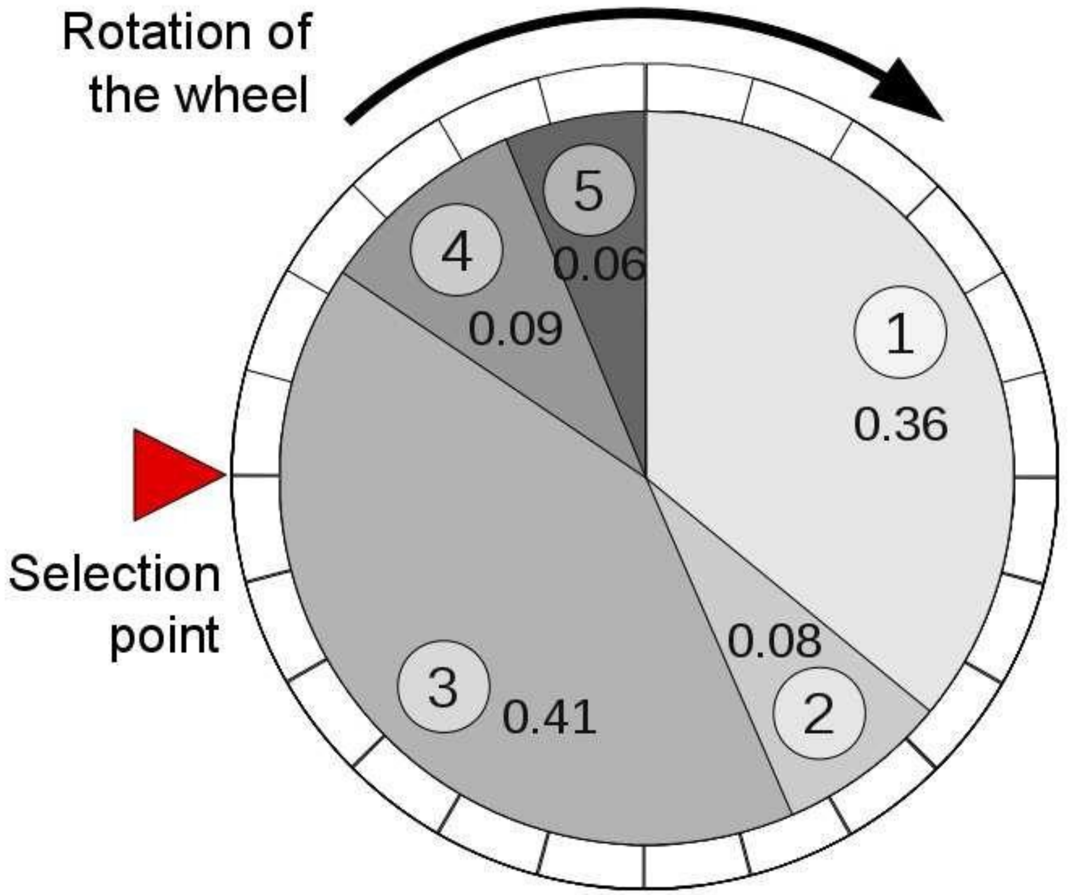
- A better approach to selection is to give every individual a chance of being selected to breed but to make fitter candidates more likely to be chosen than weaker individuals. This is achieved by making an individual's survival probability a function of its fitness score. Such strategies are known as *fitness-proportionate selection*.

Roulette Wheel Selection

- The most common fitness-proportionate selection technique is called *Roulette Wheel Selection*. Conceptually, each member of the population is allocated a section of an imaginary roulette wheel.
- Unlike a real roulette wheel the sections are different sizes, proportional to the individual's fitness, such that the fittest candidate has the biggest slice of the wheel and the weakest candidate has the smallest. The wheel is then spun and the individual associated with the winning section is selected. The wheel is spun as many times as is necessary to select the full set of parents for the next generation.

esempio

Individuo	Fitness	Prob
1	252	.36
2	56	.08
3	287	.41
4	63	.09
5	42	.06
	TOT = 700	



Multiple selections

- Using this technique it is possible (probable) that one or more individuals is selected multiple times.
- That's OK, it's what we want to happen. Remember that we are not selecting the members of the next generation, **we are selecting their parents** and it is possible for an individual to be a parent multiple times. If there is a particularly fit member of the population we would expect it to be more successful at producing offspring than a weaker rival.

Stochastic Universal Sampling

- *Stochastic Universal Sampling* is an elaborately-named variation of roulette wheel selection. Stochastic Universal Sampling ensures that the observed selection frequencies of each individual are in line with the expected frequencies. So if we have an individual that occupies 4.5% of the wheel and we select 100 individuals, we would expect on average for that individual to be selected between four and five times. Stochastic Universal Sampling guarantees this. The individual will be selected either four times or five times, not three times, not zero times and not 100 times. Standard roulette wheel selection does not make this guarantee.

Rank Selection

- *Rank Selection* is similar to fitness-proportionate selection except that selection probability is proportional to relative fitness rather than absolute fitness. In other words, it doesn't make any difference whether the fittest candidate is ten times fitter than the next fittest or 0.001% fitter. In both cases the selection probabilities would be the same; all that matters is the ranking relative to other individuals.
- Rank selection will tend to avoid premature convergence by tempering selection pressure for large fitness differentials that occur in early generations. Conversely, by amplifying small fitness differences in later generations, selection pressure is increased compared to alternative selection strategies.

Tournament Selection

- *Tournament Selection* is among the most widely used selection strategies in evolutionary algorithms. It works well for a wide range of problems, it can be implemented efficiently, and it is amenable to **parallelisation**.
- At its simplest tournament selection involves randomly picking two individuals from the population and staging a tournament to determine which one gets selected.
- The "tournament" isn't much of a tournament at all, it just involves generating a random value between zero and one and comparing it to a pre-determined selection probability. If the random value is less than or equal to the selection probability, the fitter candidate is selected, otherwise the weaker candidate is chosen. The probability parameter provides a convenient mechanism for adjusting the selection pressure. In practise it is always set to be greater than 0.5 in order to favour fitter candidates. The tournament can be extended to involve more than two individuals if desired.

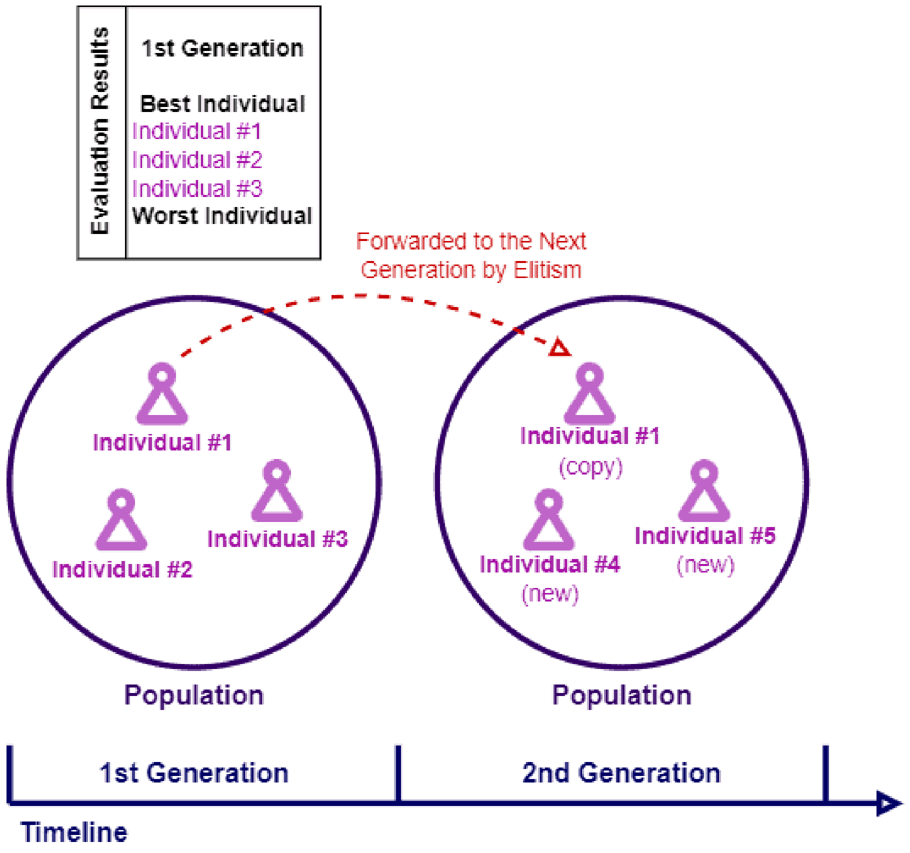
Sigma Scaling

Elitism

- Sometimes good candidates can be lost when cross-over or mutation results in offspring that are weaker than the parents. Often the EA will re-discover these lost improvements in a subsequent generation but there is no guarantee. To combat this we can use a feature known as *elitism*.
- Elitism involves copying a small proportion of the fittest candidates, unchanged, into the next generation. This can sometimes have a dramatic impact on performance by ensuring that the EA does not waste time re-discovering previously discarded partial solutions. Candidate solutions that are preserved unchanged through elitism remain eligible for selection as parents when breeding the remainder of the next generation.

Elitism

- supports elitism via the second parameter to the evolve method of an **EvolutionEngine**.
- This elite count is the number of candidates in a generation that should be copied unchanged from the previous generation, rather than created via evolution.
- Collectively these candidates are the elite. So for a population size of 100, setting the elite count to 5 will result in the fittest 5% of each generation being copied, without modification, into the next generation.



APIs

```
/**
* Execute the evolutionary algorithm until one of the termination conditions is met,
* then return the fittest candidate from the final generation.

* @param populationSize The number of candidate solutions present in the population
* at any point in time.

* @param eliteCount The number of candidates preserved via elitism. In elitism, a
* sub-set of the population with the best fitness scores are preserved unchanged in
* the subsequent generation. Candidate solutions that are preserved unchanged through
* elitism remain eligible for selection for breeding the remainder of the next
generation.
* This value must be non-negative and less than the population size. A value of zero
• means that no elitism will be applied.

* @param conditions One or more conditions that may cause the evolution to terminate.
* @return The fittest solution found by the evolutionary process.
* */
T evolve(int populationSize, int eliteCount, TerminationCondition... conditions);
```

Esercizio

Diversi selection… quale è meglio?

- Prova diverse selection strategies per hello world

2. Rappresentazione

Angelo Gargantini

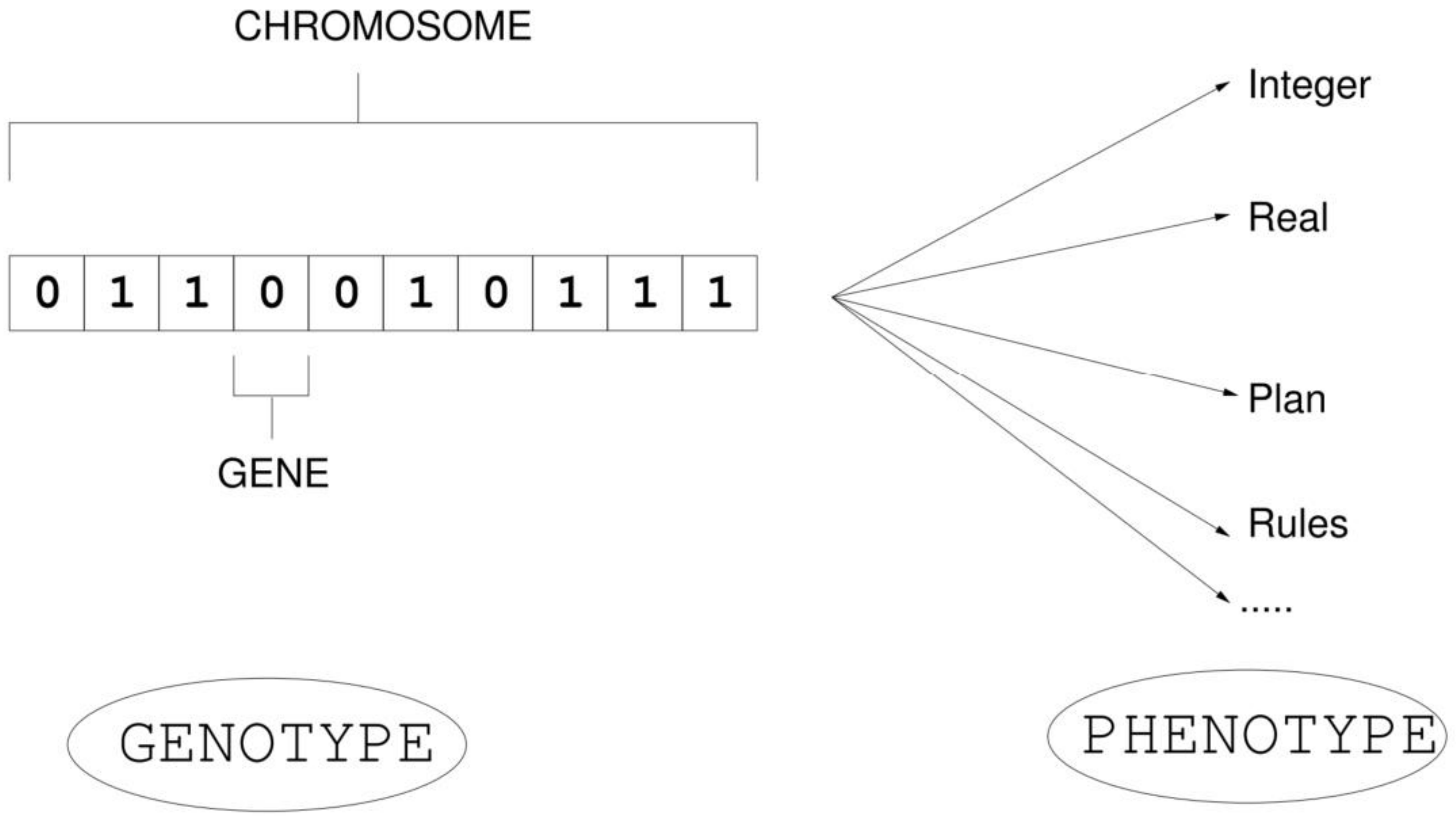
LaPeC 2024

Usiamo BitString – genetic algorithms

Una prima forma per rappresentare gli individui è quella di usare la classe BitString. Una BitString è una stringa di bit (di numero fisso – si possono fare anche indivudi con un numero di bit variabile ma si usa raramente anche per problemi del crossover)

Genotipi e fenotipi

- Solutions are coded as bit strings



Using BitStrings

- Supponiamo di avere stringhe di bits
  - Qualsiasi problema può essere rappresentato con stringhe di BIT

- Primo Esempio (banale):
- https://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf
- **The algorithm evolves bit strings and the fitness function simply counts the number of ones in the bit string. The evolution should therefore converge on strings that consist only of ones.**
- **(anche detto OneMax)**

Risolviamo One Max con Stringhe di Bit

- Primo passo usiamo stringhe di BIT

- **BitStringCrossover**(1, new Probability(0.7d)));
- **BitStringMutation**(new Probability(0.01d)));

- No elitism
  - BitStringEvaluator() →

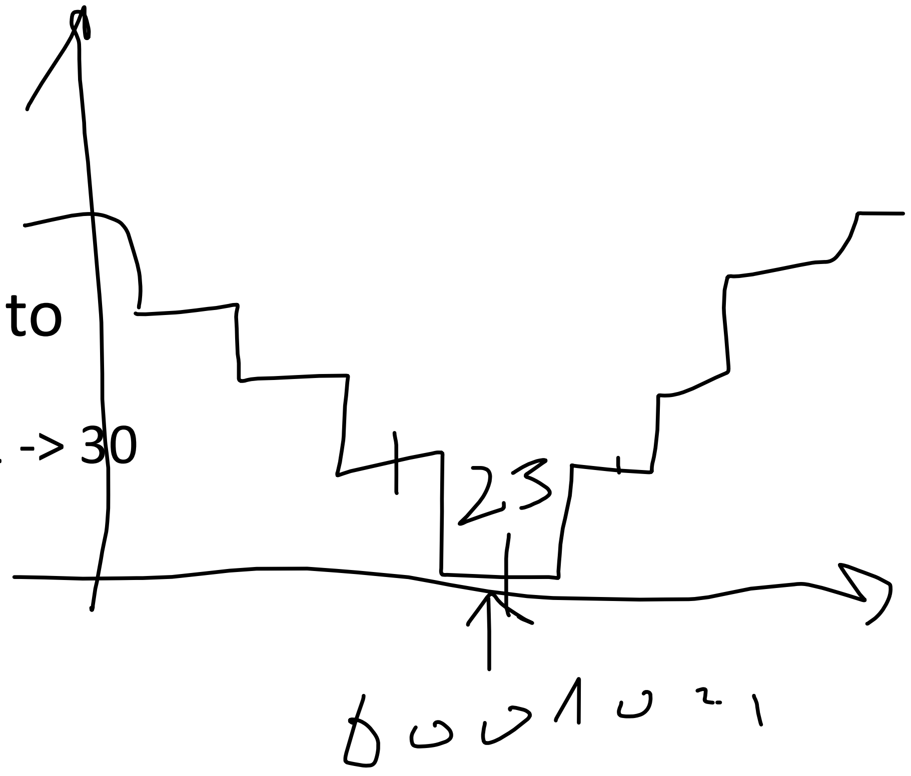
- **TargetFitness**(length, true)

Con stringhe di Bit si può rappresentare un po’ tutto

- Con stringhe di Bit si può rappresentare un po’ tutto
- Attenzione però:
  - Lo spazio del genotypo potrebbe contenere degli individui che non hanno phenotipo
    - Esempio: i float: non tutte le stringhe di bit float/double rappresentano dei numeri ci sono anche in NaN
  - La mutazione/crossover tra due elementi potrebbe generare un nuovo elemento con fitness impredicibile

Esempio 1: BitString per Interi

- Voglio trovare l’intero come sequenza di bit cui doppio di avvicina al massimo a 50.
- Rappresentazione: 16 bit
- Operatori di mutazione: crossover e mutation
- Selection: roulette
- Fitness: distanza del doppio da 50 in valore assoluto
  - 0 -> best (soluzione)
  - >0 non soluzione – esempio 10 → distanza = 50 – 10*2 -> 30
  - Fitness è naturale? Disegna la fitness
  - Attenzione che la fitneess deve essere > 0
- Come passare dai genotipo a fenotipo?
  - `int val = Integer.parseInt(candidate.toString(), 2);`



Esempio 2: BitString per double

- Voglio trovare il double che più si avvicina alla radice di 2.
- Rappresentazione: 64 bit (così trovo tutti i double possibili)
- Per passare ad bitstring a double uso:

```
double doubleVal =
   Double.longBitsToDouble(new BigInteger(candidate.toString(), 2).longValue());

Uso dei soliti operatori
Converge???
```

KnapSack Problem – con i bitstring

The Knapsack problem is an optimization problem that deals with filling up a knapsack with a bunch of items such that the value of the Knapsack is maximized. Formally, the problem statement states that, given a set of items, each with a weight and a value, determine the items we pack in the knapsack with a constrained maximum weight that the knapsack can hold, such the the total value of the knapsack is maximum.

kg	7	2	1	9
$	5	4	7	2
	A	B	C	D

Max Weight: 15kg

QUALI OGGETTI POSSO PRENDERE???

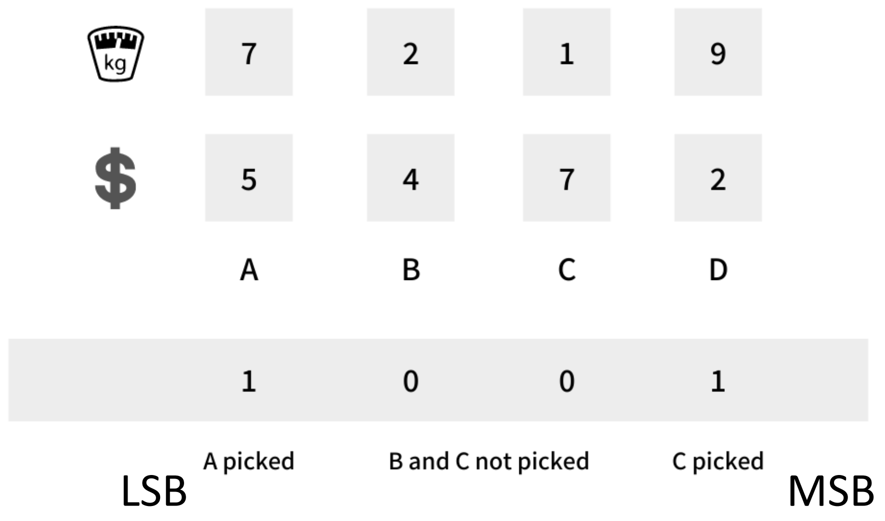
Esiste una soluzione esatta → esplora tutte le possibilità

Esempio: A, B, C -> posso (10 kg) e 16 $

Esempio B, C, D -> 12 kg e 13 $

KnapSack con binary strings

- Usiamo una rappresentazione con binary strings per rappresentare gli ogggeti presi (1) o no (0)



```java
static int peso[] = { 7, 2, 1, 9 };
static int valore[] = { 5, 4, 7, 2 };
```

Implementazione in watchmaker

- Representation Binary strings of length n
- Recombination One-point crossover (`BitStringCrossover`)
  - Recombination probability 70% (`new Probability(0.7)`)
- Mutation Each value inverted with independent probability 0.2
  - `new BitStringMutation(new Probability(0.2))`
- Parent selection Truncation selection al 25%
  - `new TruncationSelection(0.25)`
- Population size 5
- Elitism: 1
- Initialisation Random
- Termination condition- Stagnation

Alcune note

Per scorrere tutti i bit usate:

```
for (int i = 0; i < candidate.getLength(); i++)  { …. }
```

Per controllare l’i-esimo bit

```
candidate.getBit(i)
```

String->BitString->List<T>

Passiamo ora a liste di elementi generici T. Un individuo è una lista di elementi T

Usiamo come rappresentazione delle liste

- Ogni individuo è una lista di <T>
  - Con T Stringa-> lista di stringhe, intero -> lista di interi
  - Può essere una **permutazione** di un insieme stabilito di <T> oppure elementi a caso di T

	Permutazione	**Non permutazione**
CandidateFactory	ListPermutationFactory<T>	CandidateFactory<List<T>>
CrossOver	**Partially Mapped Crossover (PMX)**	ListCrossover<T> Date due liste (anche lunghezza diversa) fa il crossover
Mutation	ListInversion	ListInversion
		ListOperator (mutation of elements)
	ListOrderMutation	(ListOrderMutation)

Lista di elementi (non permutazione)

- La soluzione è una lista di elementi di un tipo predefinitio (ad esempio intero)
  - Esempio [1, 4, 7 , 8]
- Per crearlo devo implementare

```
new AbstractCandidateFactory<List<T>>() {
       @Override
       public List<T> generateRandomCandidate(Random rng) {
       List<T> l = new ArrayList<>(); riempi con elementi la lista
       return l;
       }
};
```

Mutazioni

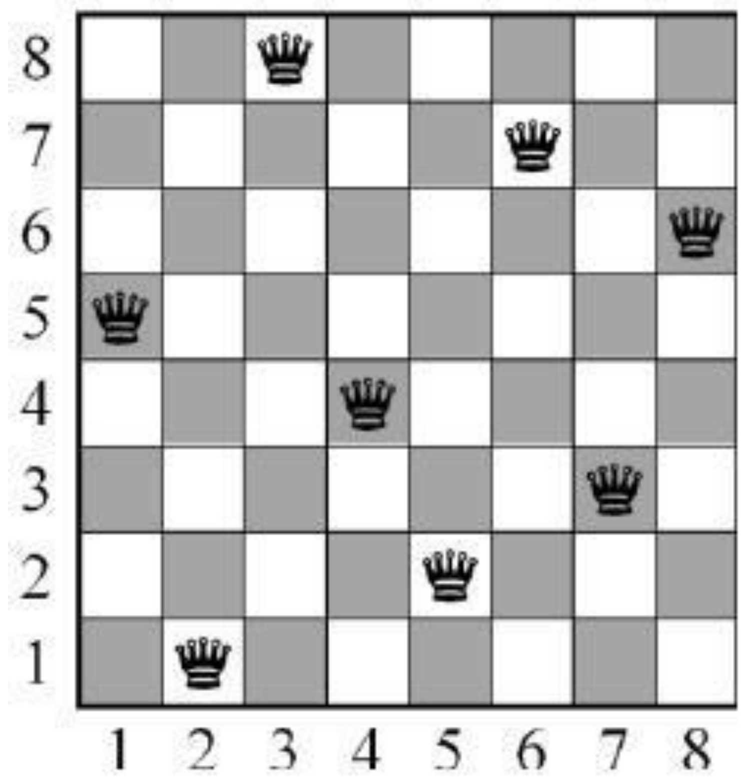
- ListCrossover<T>
  - [1, 4, 7 , 8] [5, 6, 4, 8] →
  - [5, 6, 7 , 8] [1, 4, 4, 8]

- ListOperator(operator)
  - Applica operator a tutti glie elementi

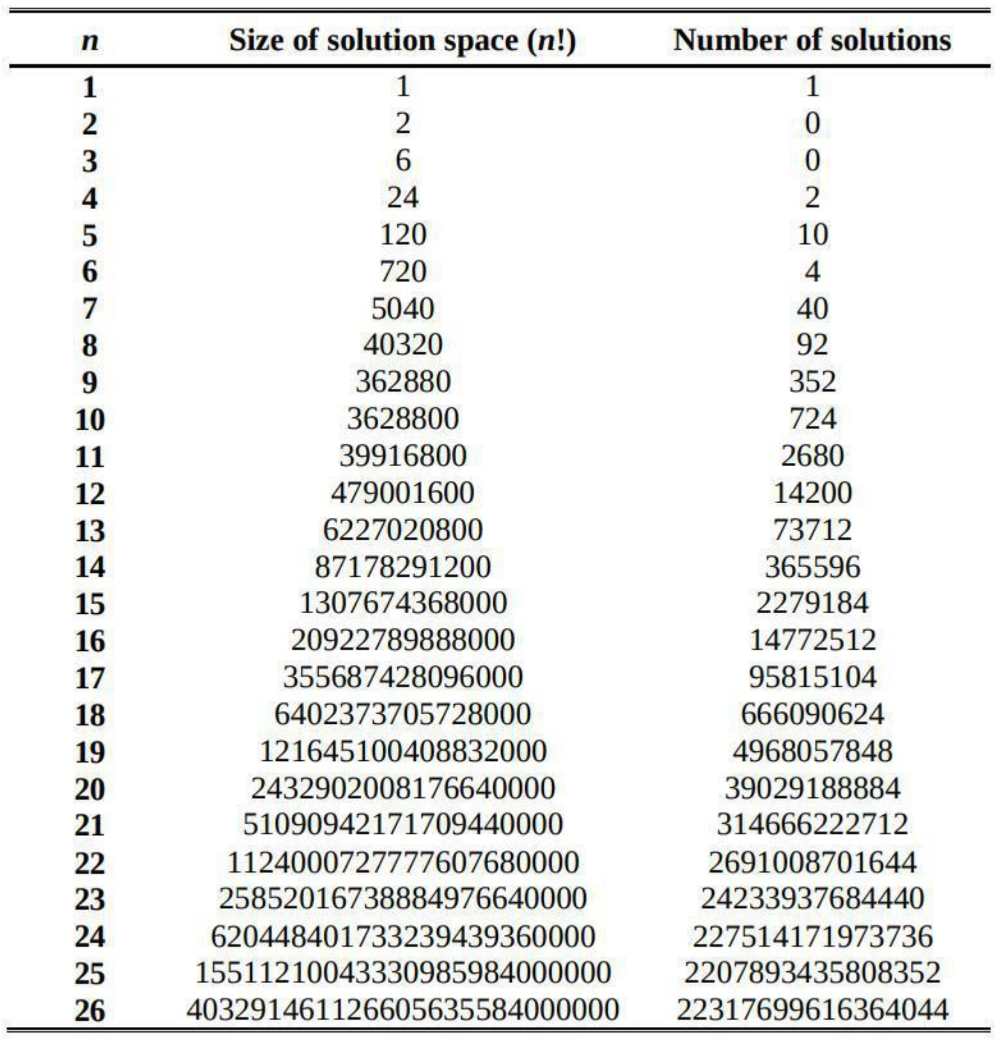
Esercizio - Problema delle 8-QUEENs

- Come disporre n queens in una scacchiera nxn senza che si mangino

- Nota che si mangiano
  - Stessa colonna
  - Stessa riga
  - Stessa diagonale



Metodi di risoluzione esatta vs ec



n	**Size of solution space (*n*!)**	**Number of solutions**
1	1	1
2	2	0
3	6	0
4	24	2
5	120	10
6	720	4
7	5040	40
8	40320	92
9	362880	352
10	3628800	724
11	39916800	2680
12	479001600	14200
13	6227020800	73712
14	87178291200	365596
15	1307674368000	2279184
16	20922789888000	14772512
17	355687428096000	95815104
18	6402373705728000	666090624
19	121645100408832000	4968057848
20	2432902008176640000	39029188884
21	51090942171709440000	314666222712
22	1124000727777607680000	2691008701644
23	25852016738884976640000	24233937684440
24	620448401733239439360000	227514171973736
25	15511210043330985984000000	2207893435808352
26	403291461126605635584000000	22317699616364044

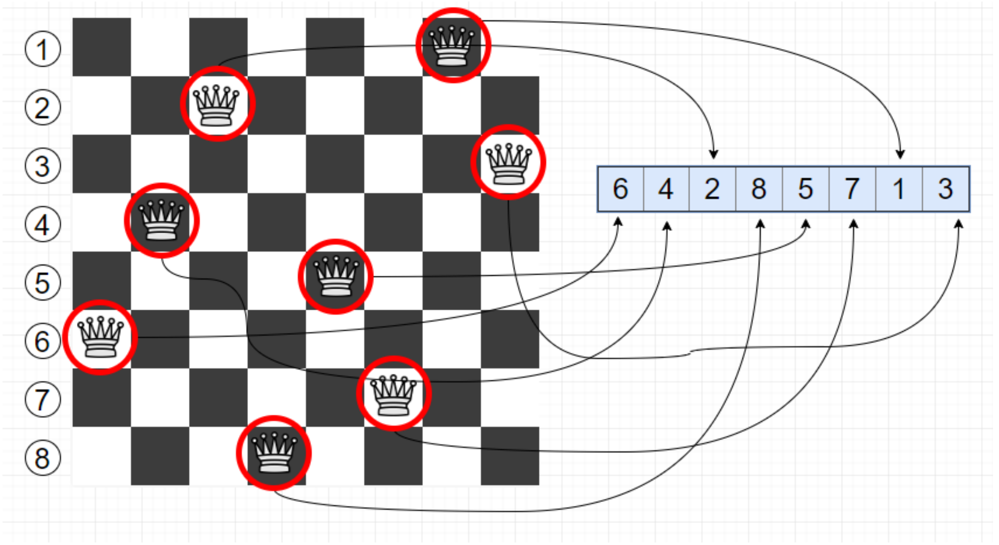
Rappresentazione della soluzione

- Numero la scacchiera da 1 a 64
- Come BitString
  - Ogni bit rappresenta se 1 che c’è una queen altrimenti no
  - Solo le bitstring con 8 bit sono valide
  - Spazio enorme e le soluzioni sono pochissime

- Una lista di 8 posizioni da 1 a 64
  - Esempio [1,2,3,6,10 ,…] –
  - Devo assicurarmi che o con la mutazione inserisca tutte le posizioni
  - O che lo faccia inizialmente

Rappresentazione 1

- In ogni colonna al massimo c’è una queen
- Una lista di 8 posizioni, ognuna indica in che posizione si trova la queen in quella colonna



- Iniziale in posizioni a caso da 1 a 8
- Poi posso fare tutte le operazioni sulla lista

```
new ListCrossover<Integer>());
new ListOrderMutation<Integer>());
```

Calcolo della fitness

- Conto quante coppie di regine si mangiano
- Una regina si mangia se è sullo stessa riga (colonna non c'è problema)
- O diagonale

Colonna->		i		j
pos[i]		Q		
pos[j]				Q

Per controllare se due regine sono sulla stessa riga è semplice: pos[i] == pos[j]

Due regine sono sulla stessa **diagonale** se j-i == ABS(pos[j] – pos[i])

Risoluzione

- Vedi

Factory di Liste di interi

- Vedi codice per creare liste di interi

Lista - permutazione

Soluzione che sono permutazioni

- In molti casi, la soluzione è una permutazione di una lista prefissata
- Inizializzazione: costruisco la lista di elementi e poi li posso ordinare a a caso (`ListPermutationFactory`)
- Quando lo muto posso solo cambiare l’ordine

```
ListOrderMutation<T> A special mutation implementation that instead of changing the genes
of the candidate, re-orders them. A single mutation involves swapping a a random element
in the list with the element immediately after it.

ListInversion An evolutionary operator that randomly reverses a subsection of a
list.
```

N queeen - Esercizio

- Con permutazione:
  se vogliamo essere sicuri che due regine non siamo mai sulla stessa riga, basta usare come individuo una lista di N interi tutti diversi.
- A questo punto la soluzione sarà una permutazione

travelling salesman problem – lista di città

- The **travelling salesman problem**, also known as the **travelling salesperson problem** (**TSP**), asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.

Usando evolutionary computing

- Data la lista di città che il commesso deve visitare
  - Esempio [Milano, Bergamo, Torino, Bologna]
- So la distanza tra due città
- Ogni soluzione indica l’ordine con cui le visita
  - In questo caso non introduco città, posso cambiare solo l’ordine
- Fitness la distanza totale

Esercizio – implementa il TSP

- Con List<T> puoi prendere List<String>
    - Inizialmente la lista dovrà contenere tutte le città

- Per la distanza puoi usare una mappa

```
Map<String, Integer> map = Map.of("Bergamo-Milano", 50, "Bergamo-Torino", 120);
```

Compito a casa (fitness complessa)

- Questa estate devi sostenere **3** esami A,B,C che hanno ognuno due appelli ma in giorni diversi. A al giorno 10 e 20, B al giorno 5 e 25, C al giorno 7 e 30.
- Vuoi sostenere tutti e tre gli esami ma vuoi decidere in quali appelli per massimizzare le media dei voti nei tre appelli. Il voto dipende da quanto tempo hai tra un appello e l’altro per studiare (studi una sola materia alla volta, quella del prossimo esame). Se studi fino a 5 giorni prendi 18, ogni giorno in più che studi prendi un voto in più (al massimo 30).
- Qual è l’ordine degli appelli che massimizza la media finale dei voti? Nota: devi sostenere tutti e tre gli esami

- Tip: io userei una lista di 3 …

Possibili rappresentazioni della soluzione

- Come una stringa di bit, un bit per ogni appello che faccio (2 appelli per Corso, sono 6 bit)
  - Esempio 100110 -> faccio il I appello di A e C, il secondo di B
  - PRO: facile rappresentare
  - CONTRO: calcolo fitness complicato, devo accettare solo le bitstring con 3 bit→ ho molti individui (quanti?) con fitness nulla

- Lista di interi

Lista di interi

- Soluzione 1:
  - Numero gli appelli da 0 a 5, poi memorizzo I tre appelli che faccio
  - Esempio [1,3,4]

  -
    ```
    appello 1(C) day 7 voto 20 appello 3(A) day 20 voto 26 appello 4(B) day 25
    voto 18 media: 21.333333333333332
    ```
  - Se non è ordinato non lo consider
  - Se ha esami ripetuti non lo consider
  - …

- Lista di 3 interi. Ogni intero la data dell’esame
  - Il primo 10 e 20, il secondo 5 e 25, il terzo 7 e 30.
  - Tutte le soluzioni sono corrette
  - Fitness …. Calcolo voto
  - Mutazioni: ad hoc
- Lista di stringhe A1, A2, B1, B2
  - Mutazioni ad hoc