Artificial Intelligence 4 Games

Short Introduction to Evolutionary Algorithms

2020

Jakub Kowalski, University of Wrocław

Evolutionary Algorithms

The Vision

We have (game) agents that are thrown into the (game) environment where they gradually learn from the scratch by themselves to act intelligently, survive (and win).

It is enough to give means, and the evolution will find the right purpose.

The hopes are for survival of the fittest leading to an open-ended evolution.

The Practice

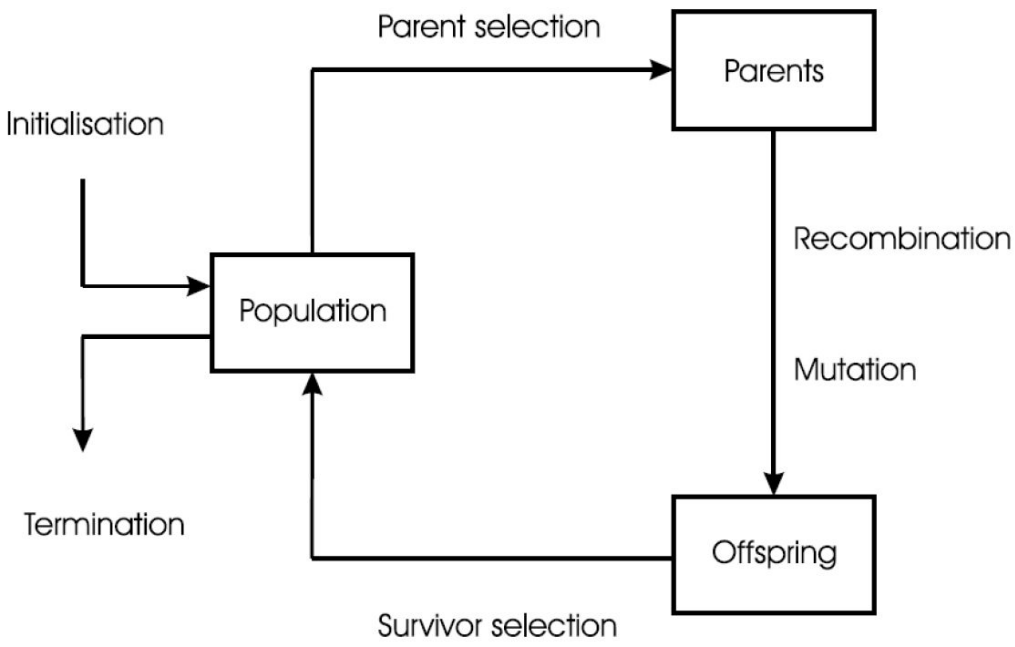
A family of stochastic global optimization algorithms, (loosely) inspired by Darwinian natural evolution.

We try to help randomness to successfully produce something worthwhile.

Extremely domain-general.

Computationally expensive.

Generic EA Scheme



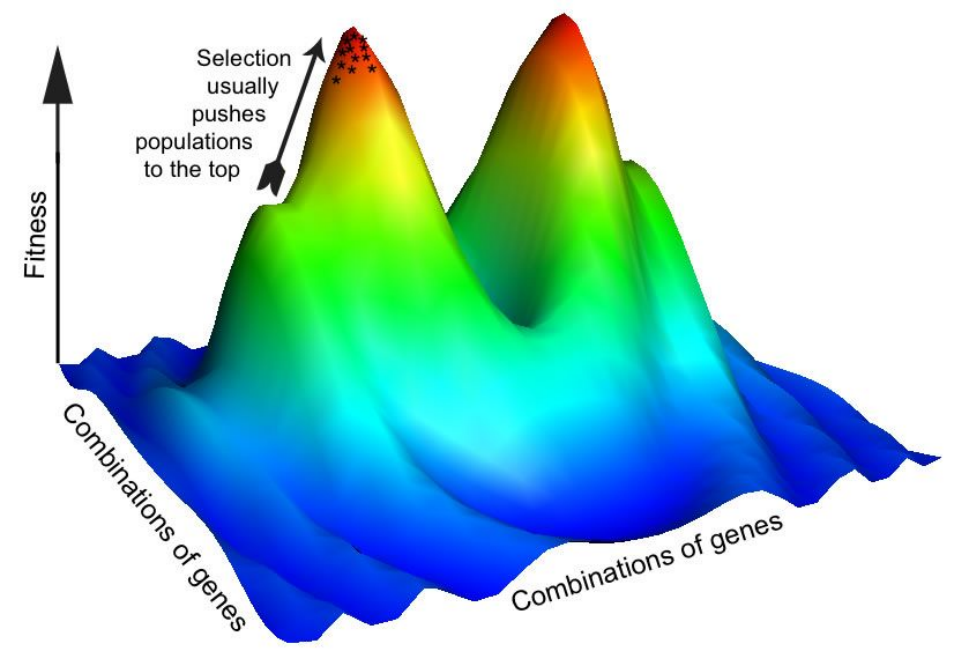
Source: Julian Togelius

Pseudocode

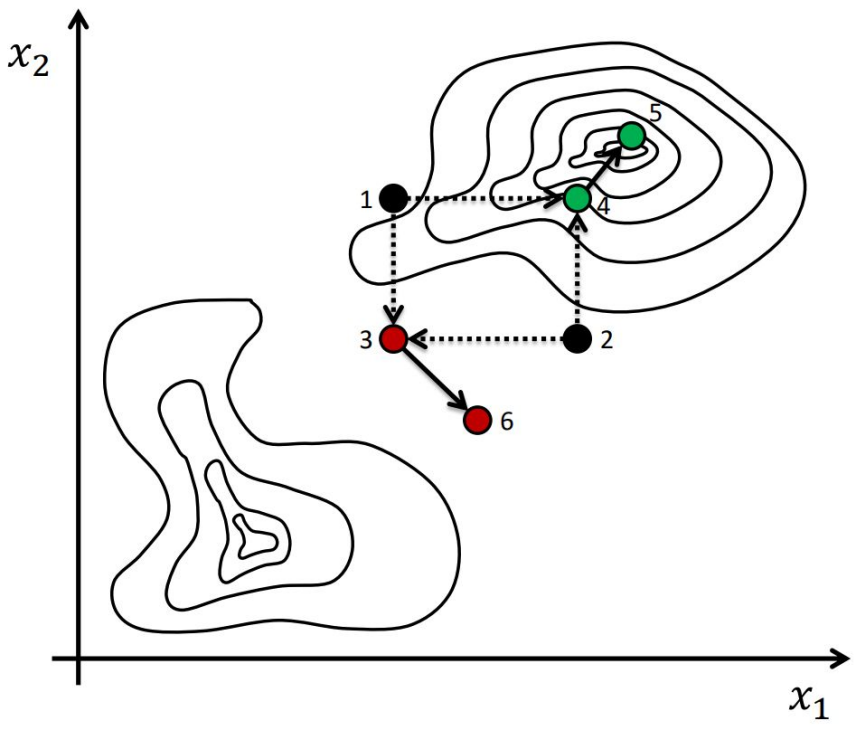
INITIALIZE population (with random solutions)
EVALUATE each candidate
UNTIL **TERMINAL CONDITION** is satisfied

> **SELECT** parents
> **RECOMBINE** pairs of parents
> **MUTATE** the resulting offspring
> **EVALUATE** new candidates
> **SELECT** individuals for the next generation

The Fitness Landscape



Source: Julian Togelius



G. N. Yannakakis, J. Togelius: Artificial Intelligence and Games. Springer, 2018.

What’s essential

Encoding of individuals

What is our domain? How to encode our candidate solutions in a terms suitable for evolutionary approach?

Fitness evaluation

What is our goal? How to calculate it so that its value can be used to compare the quality of individuals across multiple populations?

Selection

How to, based on their fitness, select individuals good enough to produce offspring? How many offspring should be produced and what number of parents should contribute?

Variation

How to introduce random variations inside the population that will likely push it to a good direction? Mutation? Crossover? How should they work and with what probabilities?

Main elements of the algorithm

- ★ Representation
- ★ Evaluation function
- ★ Population
- ★ Parent selection
- ★ Variation
- ★ Survivor Selection
- ★ Termination

Representation

A **population** is a set of individuals, which are also referred to as **genotypes**.

Each genotype is a sequence of **genes** that are the smallest modifiable units we are interested in.

All variation and replication mechanisms are performed on the genotype, so keep it as simple as possible.

E.g. We can have our individuals encoded as fixed-size arrays of doubles or words (unknown length sequences of letters).

What is evaluated by the fitness function is the **phenotype**, i.e. the semantic of our individual. It may be (but not have to) a completely different being.

Genotype-to-phenotype mapping should be fast, deterministic, and preserve locality.

E.g. the array of doubles can represent the weights of a neural network, words can represent a solution to a DNA-like sequence that encodes the characteristics of a game character.

Evaluation function

It is also called a **fitness** function, and it is used to calculate of fitness of a particular individual - the value approximating its quality.

The role of fitness is to allow comparison among different individuals (also coming from different populations).

Thus it should return a number (or ranking) of the evaluated individual (phenotype).

Usually, our goal is to find the single individual that optimizes the value of the fitness function.

We can have one, or multiple such functions.

Sometimes, we can only compare individuals against each other, without a means to create an objective reference point.

Evaluation functions should be smooth, fast to evaluate, noise-free, accurate, and relevant.

Population

Contains a (usually fixed) number of individuals (genotypes).

During the evolution, we expect the population to improve. Thus, although it is usually enough to obtain one strong (in terms of fitness) individual, the average performance in the population should also improve.

The population should be:

- ➔ diverse,
- ➔ large enough,
- ➔ small enough.

A single iteration of the evolutionary loop (referred to as **generation** or **epoch**) consists of creating an offspring population from the current one.

We try to modify the existing set of candidate solutions to create another set, and later select which of those are good enough to qualify for further computation.

Initialization

Usually random. (Restrictions to generate individuals from a relevant domain subset.)

May be preselected: results of previous evolutionary runs, solutions from other algorithms, … diversity!

Parent Selection

We need to select individuals that will “reproduce”. Speaking normally - their values will be somehow merged to create a new set of candidate solutions.

This choice is usually based on the fitness value, fitness ranking, or via direct comparison between individuals.

May be deterministic or stochastic.

How many to select? As a result we usually want offspring population to be of the same size as the parent one.

Selection operators

Some of the simplest and most popular methods are:

- ➔ Fitness-proportionate selection (a.k.a. Roulette wheel selection)
- ➔ Tournament selection
  (In form of best-of-k with known fitness values, or true tournament with “matches” against pairs of individuals.)
- ➔ Fully random selection

Variation

Mutation

Should introduce diversity to the population and prevent stagnation.

Usually randomly modifies individual, but not too much and only with some small chance. Can be applied to the entire individual (e.g. global flip bit) or for a selected genes:

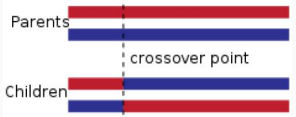
- ➔ Flip a bit
- ➔ Substitute with uniformly-randomized value (for floats/ints)
- ➔ Modify value by adding Gaussian distributed random value (for floats)

Crossover (recombination)

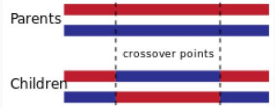
Combine existing individuals to create new ones.

Usually some form of gene-mixing of two (rarely more) genomes aimed for obtained a better quality child(ren). Popular operators (more sophisticated ones are usually problem specific)

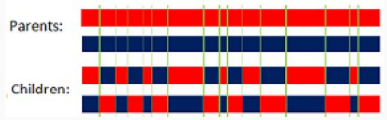
- ➔ One-point crossover:



- ➔ Two-point crossover:



- ➔ Uniform crossover:



Survivor Selection

We have the current population (parents) and newly created offspring population. We need to select individuals that will form a next-generation “current” population.

Who do we want to replace? Always remove the worst, or use some kind of randomness? We need to preserve the population diversity!

Often we use **elitism**, that is we do not forget the best individuals. It is usually a good thing, but too high will reduce the population diversity. We may limit the lifespan of an individual via e.g. applying a concept of **aging**.

Selection operators

Some simple ideas:

- ➔ Take the entire children population
- ➔ Take best *n* (population size) from parents+children
- ➔ Keep best *k* parents (elitism) and refill the population with best children
- ➔ Use roulette selection (without repetitions)
- ➔ Take best *n* from joint population by fitness ordering but skip too similar to already selected

Termination

We are usually not interested in an open-ended evolution, but want to use the results after a finite time.

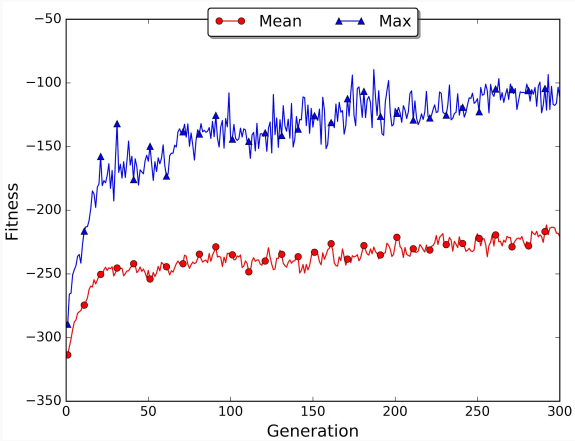
What criteria can we use to stop evolution?

- ➔ Exceeded given time limit / computational limit
- ➔ We reached given generations limit
- ➔ We have found solution that satisfies given criteria
- ➔ We did not improve fitness since some number of generations
- ➔ ...

Visualizing evolutionary run

We need to somehow see what is happening during the course of the evolution and confirm that the algorithm works.

Good way is to plot a fitness of the best individual and population average for each generation. Ideally, gathering data from multiple (as many as possible) runs.



(Slightly) more in-depth look at EAs

- ★ Naming conventions
- ★ Evolution Strategies
- ★ Genetic Programming
- ★ Implementation
- ★ Advanced approaches

Naming Conventions

- **Evolutionary Computations**
  Lot of weird things including everything below and much, much more (swarm-stuff, etc.)
  - **Evolutionary Algorithms (EA)**
    Everything below ;-)
    - **Genetic Algorithms (GA)**
      Characterized by binary string representation
    - **Evolution Strategies (ES)**
      Real-values vector representation, mutation-based
    - **Genetic Programming (GP)**
      Evolves tree-like structures
    - **Neuroevolution**
      Evolves artificial neural networks
    - ...

Evolution Strategies (ES)

Let us describe a how **(μ+λ) ES** works.

1. Create a population of μ+λ individuals.
2. Each generation:
   a. Evaluate all individuals in the population
   b. Sort by fitness
   c. Remove the worst λ individuals
   d. Replace with mutated copies of the μ best

ESs are usually mutation-based (no crossover), and mutation is Gaussian.

Algorithm 2: Evolution Strategy(μ,λ,n)

```
1  INITIALIZE (Population, μ + λ individuals)
2  for i=1 to n do
3      for j=1 to (μ + λ) do
4          EVALUATE (Population[j])
5      end
6      PERMUTE (Population)
7      SORTONFITNESS (Population)
8      for j=μ to (μ + λ) do
9          Population[j] ← COPY (Population[j-λ])
10         WEIGHTMUTATE (Population[j])
11     end
12 end
```

(1+1) ES is a simple hill climber: keeps one individual and replaces its mutated version (a single child) is better.

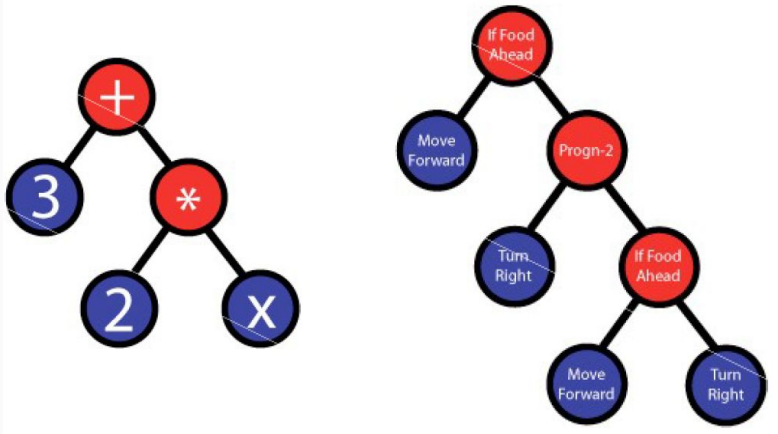
In **(μ, λ) ES** variant selection takes place only among λ offsprings, parents are always forgotten.

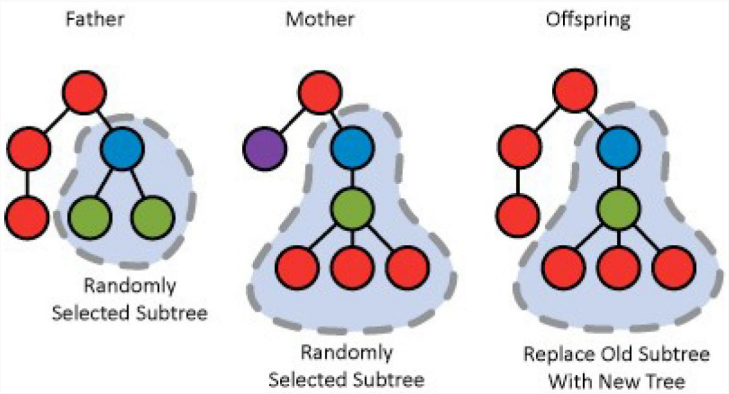
Genetic Programming (GP)

In GP, the genotype has a tree structure.

In particular, its goal is to evolve complete computer programs using evolutionary algorithms. (Note that this tree-like representation is natural for programming languages like Lisp)

In general, this “computer program” can represent anything, from a simple arithmetic expression, encoding of an AI behaviour, to a codification of a game rules.





Implementation

Using existing frameworks

They exist. Many of them. E.g., DEAP (Python), ECJ (Java), Jenetics (Java), GeneticSharp (C#), ECF (C++), Evolving Objects (C++), …

Useful when you need a lot off-the-shelf operator implementations to test, and generally optimized code. Also your problem can not be too weird.

You are willing to sacrifice some time and effort to learn the framework usage.

Writing your own code

When you value customization and/or you are too lazy to properly learn an existing library.

When a task requires lot of problem-specific customization.

Usually it is fairly easy to write your own EA implementation, that will be minimalistic, task-suited, fast to modify, and easy to improve.

Advanced approaches

There is many non-trivial things when one goes deeper into the evolution, and want his algorithm to really solve a given problem.

For some specific kinds of problems, a specific solution schemes have been developed. It is often worth to search through the literature to find them.

But also, for many not-so-formal problems (games usually fit into that category) ad-hoc solutions and progressively fixing issues may work well enough.

Just to name a few things worth to consider:

- ➔ Novelty search, stagnancy detection
- ➔ Adaptive mutation/other operators
- ➔ Clever initialization
- ➔ Noisy/nondeterministic evaluation
- ➔ Extremely computationally expensive evaluation
- ➔ No fitness value, only comparing individuals
- ➔ Multi-objective optimization
- ➔ Problems with constraints: not all the solutions are *valid* ones.

Evolutionary Algorithms for Games

- ★ Parameter tuning
- ★ Search engine
- ★ Neuroevolution
- ★ Procedural Content Generation

Parameter tuning

Nearly any AI algorithm we are making with hope of playing the game well depends on the number of parameters.

- ➔ weights of evaluation functions, action-type priorities, C constant for MCTS, various parameters for MCTS enhancements (***ε*** for ε-greedy, ***γ*** for decaying, PPA learning rate, RAVE ***K***), ...

Also, we know the value ranges for these parameters, but do not know the overall best working combination.

Usually, the exhaustive search is too expensive. In such case, we can use evolution to approximately find what we are looking for.

- ➔ Our search space is a subset of AI algorithms.
- ➔ Genotype is an array of parameters
- ➔ Phenotype is the algorithm
- ➔ We need to be able to simulate the game
- ➔ The fitness value is not explicitly known - it has to reflect the strength of the AI with the current values of the parameters.

Parameter tuning: fitness function

When our program is e.g. a multiplayer bot, the only thing we can do is to compare two (or more) agents by playing against each other.

However, such approach creates a bunch of problems:

- ➔ Evaluations are costly
- ➔ Often unreliable (depend on the map parameters, nondeterminism in agents)
- ➔ It is possible to end up with a rock-paper-scissor situation

Selection is usually based on some sort of tournament.

If it is within a population, comparison across different generations may become problematic, as there is no predefined reference point.

We may remember leaders from previous populations, and compare against them.

This type of EA application is very domain dependent, and all operators have to be chosen taking into account all the limitations of the problem.

Search Engine

We may want to use an evolutionary approach in a similar way as we use MCTS: to perform semi-random search through the game tree in hope of finding good move choices.

Thus, our search space is the sequences of game moves, with a single move treated as a gene, and phenotype being the game state after applying this sequence.

This is another simulation-based algorithm, as the evaluation function is usually based on a number of game playouts.

It's a relatively novel approach.

As we are working with "evolution", we hope that the game move space is somewhat "monotonic", and search operators (crossover/mutation) will finally produce something that works well.

We rather optimize the whole sequence instead of search for a good sequence gradually.

Such approaches exist in the literature under the names *Rolling Horizon Evolutionary Algorithm* (RHEA), and *Online Evolutionary Planning* (OEP).

These are the topic of our next lectures.

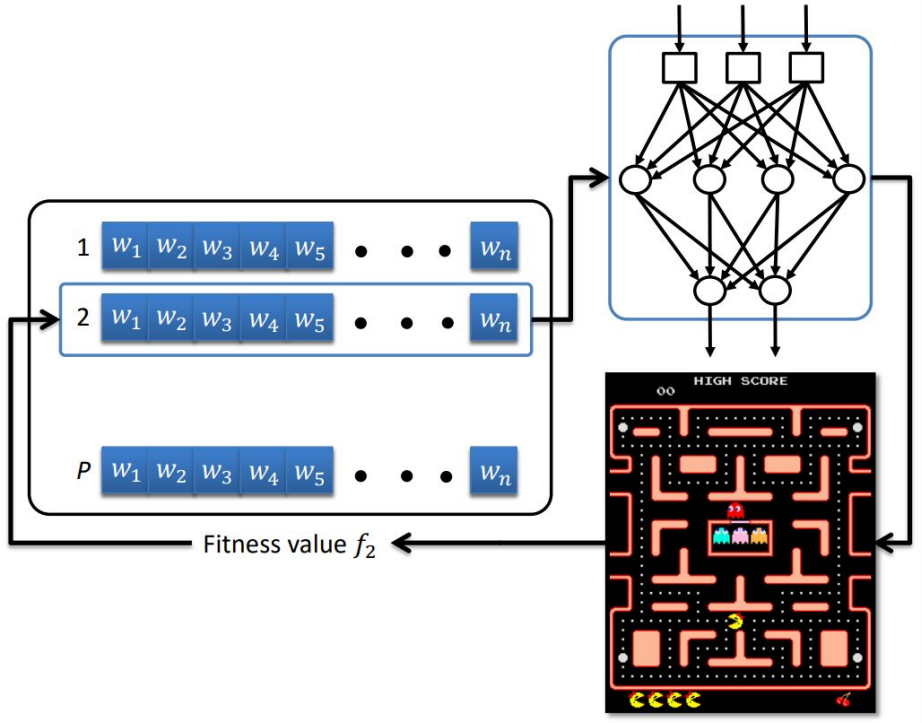
Neuroevolution

The idea is to designing neural networks (their weights, topology) using evolutionary approach.

Such networks could be responsible for handling the in-game character AI.

Neuroevolution was used in multiple games including e.g., classic board games, MS. Pac-Man, Simulated Car Racing, Mario, Atari games.

★ S. Risi, J. Togelius: Neuroevolution in games: State of the art and open challenges. IEEE TCIAIG, 9(1), 25-41, 2015.

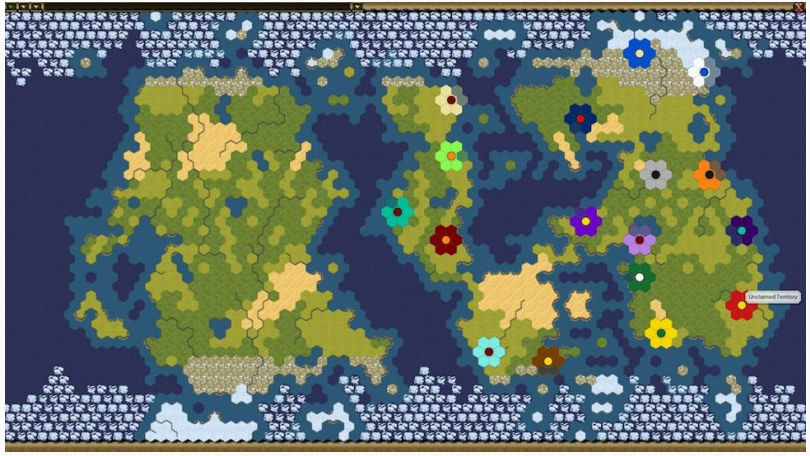


G. N. Yannakakis, J. Togelius: Artificial Intelligence and Games. Springer, 2018.

Procedural Content Generation (PCG)

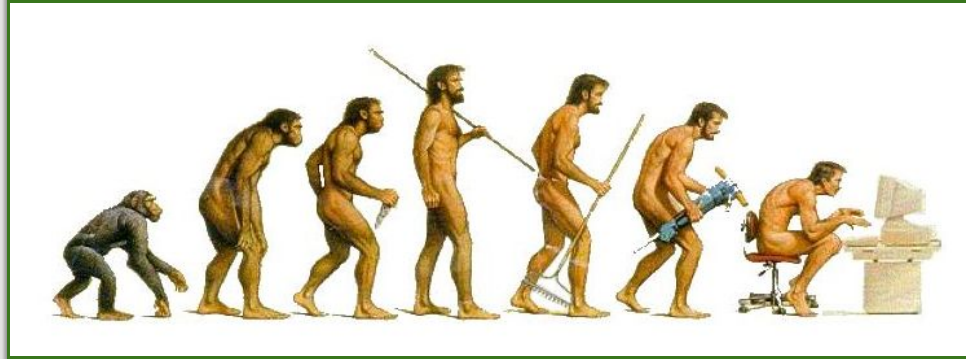
Evolutionary approach is one of the main methods for search-based procedural content generation, which ranges from creating textures, gear statistics, levels and quests, up to game rules.

PCG will be our topic near the end of the course.

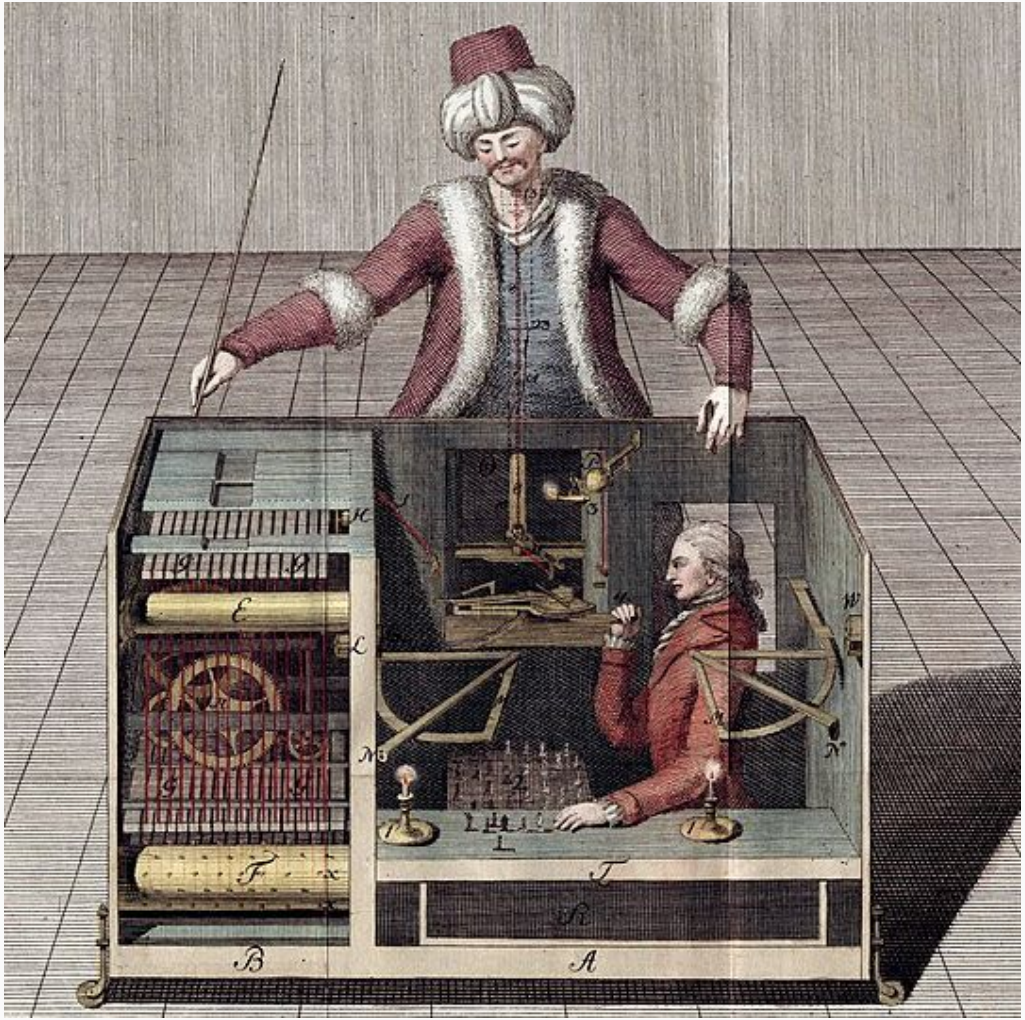


Summary

- ★ EAs optimize candidate solutions relying on semi-random modifications of promising so-far obtained results.
- ★ Generic scheme: selection, crossover+mutation, replacement
- ★ Applicable to a wide range of problems



Thanks!



Bonus reference quiz