# Evolutionary Frankenstein:
# From "metaphor algorithms" to "components research"

Claus Aranha
University of Tsukuba

2022-12-15

download this material:

e-mail: caranha@cs.tsukuba.ac.jp

# A little bit about myself

- Born in Brazil, living in Japan for a long time;
- Love programming:
  - Recombine small words to make anything;
  - Calculate, Visualize, Simulate;
  - ... Games
  - ... Art?
  - ... Life???
- Research: Artificial Evolution and Artificial Life;
- And also computer games;

... Do crazy things with the computer!

# Outline of the Talk

**Topic**: How to design algorithms that use Evolution to solve optimization problems?

**Message**: It is possible (**and fun!**) to create new techniques through careful study and recombination of classic algorithms;

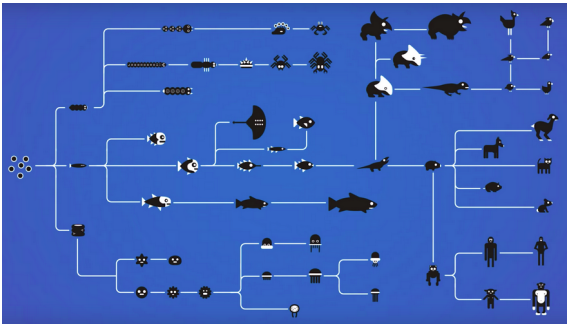
**Topics**:

- What is Evolutionary Computation?
- A few classic Evolutionary Computation algorithms;
- A LOT of weird algorithms; (and the problem with them);
- How can we keep the good and throw away the bad?

# Part I
# A quick guide to Evolutionary Computation

# What is Evolutionary Computation?

- Natural Evolution is amazing:
  - Creates a variety of incredible creatures;
  - These creatures are well adapted to their environment;
  - Domain information is not used **a priori**;



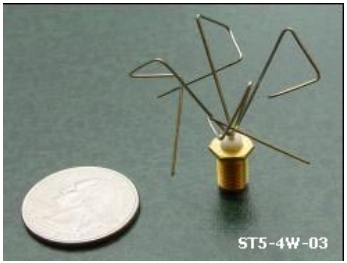
- Can we apply evolution to computational systems?

# Evolutionary Computation is Cool

For over 50 years, researchers have developed an outline for Artificial Evolution:

- Define the problem to be solved as **The Environment**
- Define the solution to be found as **The Individuals**
- Evaluate the individuals and give them a **Fitness Score**;
- Reproduce and Modify the individuals based on their Fitness;

This general algorithm has been used to solve many hard problems:




# Why does Evolutionary Computation Work?

Evolutionary Computation is commonly applied to Optimization Problem:

*Find a vector $x \in \mathbb{R}^d$, that minimizes the value of $f(x)$*

It works by testing several solutions, until it finds a solution that works. This is also called Search Based Optimization (SBO).

- SBO does not depend on gradient, so it works for discontinuous problems;
- SBO does not require domain knowledge, so it is a *general* method;

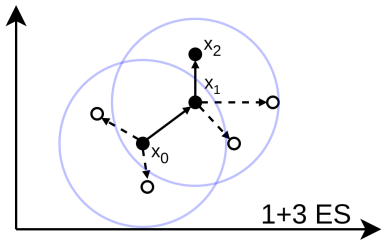
Also, algorithms based on Evolutionary Computation are very easy to implement, and also easy to parallelize.

Let's see some classical Evolutionary Computation algorithms.

# Evolution Strategy (ES)

Rechenberg and Schwefel, 1971

Key idea: Offspring with beneficial mutations replace the parent.




Later versions adapt the mutation step $s$ during the optimization.

Evolution Strategy (1+$\lambda$-ES)

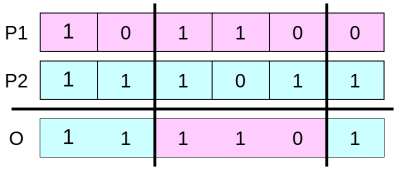
**Require:** $x \in \mathbb{R}^d$ (Initial Solution)
**Require:** $\lambda$: number of offspring
**Require:** $s$: step size
**while not** terminate condition **do**
  **for** $i : 0 \rightarrow \lambda$ **do**
    $o_i \leftarrow x + N(0,1)^d * s$
  **end for**
  $o_b \leftarrow$ best $o_i$
  **If** $o_b$ **better than** $x$: $x \leftarrow o_b$
**end while**
**Output** $x$

# Genetic Algorithm

Holland, 1975

Key ideas:

- Population tracks several solutions in parallel;
- Selection bias search to good solutions;
- Crossover exchanges building blocks;




These three ideas imply that good building blocks spread through the population (schemata theory)

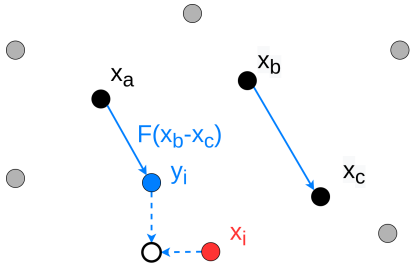
Genetic Algorithm

**Require:** Binary Representation of Solution
**Require:** $P \in \{1,0\}^{d \times p}$ (Initial Population)
**while not** terminate condition **do**
  Evaluate fitness of population, $f(P)$
  **for** $i : 0 \rightarrow p$ **do**
    Select parents $x_a$,$x_b$ by fitness
    $o_i \leftarrow$ crossover$(x_a, x_b)$
    $o_i \leftarrow$ mutation$(o_i)$
    Insert $o_i$ in $P_{\text{new}}$
  **end for**
  $P \leftarrow P_{\text{new}}$
**end while**
**Output** best $x \in P$

# Differential Evolution

Storn and price, 1997

Key idea: Mutation based on the difference of two random solutions means that the mutation step automatically adapts to the size of the population.



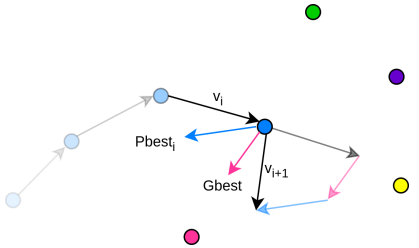

Differential Evolution

**Require:** $P \in \mathbb{R}^{d \times p}$ (Initial Population)
**Require:** Parameters $F, Cr$
**while not** terminate condition **do**
  **for** $i : 0 \rightarrow p$ **do**
    Choose random $x_a, x_b, x_c \in P$
    $y_i = x_a + F(x_b - x_c)$
    **for** $j : 0 \rightarrow d$ **do**
      $y_{i,j} \leftarrow x_{i,j}$ with prob $Cr$
    **end for**
    **If** $y_i$ better than $x_i$: $x_i \leftarrow y_i$
  **end for**
**end while**
**Output** best $x \in P$

# Particle Swarm Optimization

Dorigo 1992

Key idea: Solutions move through the search space like particles, influenced by the current local and global optima. Particles exchange information and track multiple optima.




Particle Swarm Optimization

**Require:** $P \in \mathbb{R}^{d \times p}$ (Initial Population)
**Require:** $V \in \mathbb{R}^{d \times p}$ (Speed Vector)
**while not** terminate condition **do**
    Pbest $\leftarrow P$, Gbest $\leftarrow$ best($P$)
    **for** $i : 0 \rightarrow p$ **do**
        $V_i = V_i + w_p * \text{Pbest}_i, w_g * \text{Gbest}$
        $P_i = P_i + V_i$
        Update $\text{Pbest}_i$
    **end for**
    Update Gbest
**end while**
**Output** best $x \in P$

# Different Algorithms, Different Ideas

- Exploration vs Exploitation;
- Parallel Search vs Collaborative Search;
- Sequential Search vs Sampling;
- Escaping Local Optima;
- Niching;
- Exploiting Domain Knowledge

There are many ideas to explore, and they can guide the creation of new algorithms;

# Part II
# The Rise of the Metaphors

# The Birds and the Bees

## Artificial Bee Colony (Karaboga and Dervis, 2005)

- Population is divided into Worker Bees, Onlooker Bees, Scout Bees
- Worker bees find food, evaluate the closest nectar, and dances
- Onlooker bees observe the dance and go to the closest source
- Scout bees replace abandoned food sources with new sources

## Cuckoo Search (Yang and Deb, 2009)

- A cuckoo replaces its solution by using Levy Flight;
- Choose a random nest and replace it with a new solution;
- Abandon a fraction of the worst nests;

# There is something strange about the birds and the bees

The description of "Artificial Bee Colony" and "Cuckoo Search" feels different from the classical algorithms:

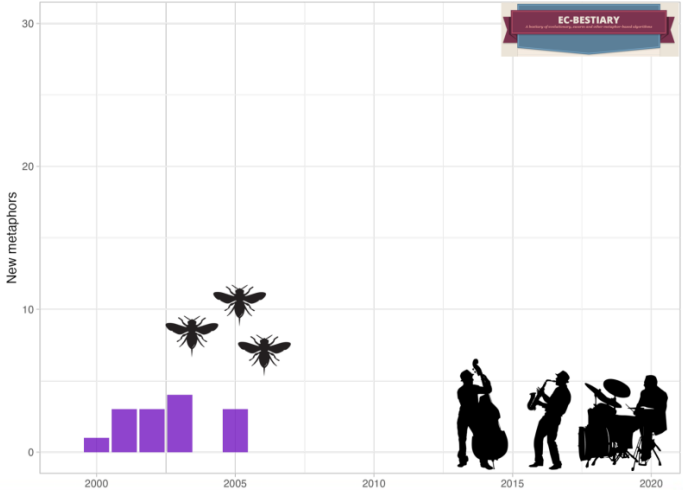
- What does the dance of the bees mean?
- The ABC description does not mention that only one dimention is changed at a time;
- Are the solutions in CS the birds or the nests?
- "Scout bees" and "Abandon nest" do the same thing; Why different names?
- The details of the implementations are not explained fully in the papers.

**Metaphor Heuristics**: The metaphor (birds, bees) is more important than the algorithm

# A Cambrian Explosion of Metaphor Heuristics

2000-2005

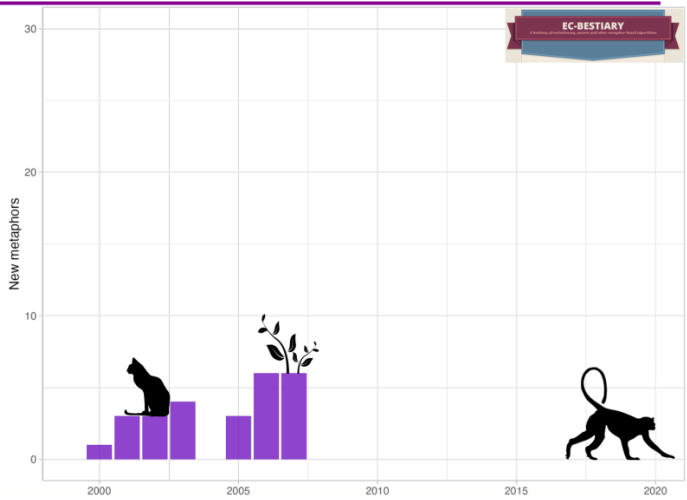
Antibodies (2000)
Honey Bee Marriages (2001)
Musicians (2001)
Sheep Flocks (2001)
Bacterial Chemotaxis (2002)
Bacterial Foraging (2002)
Gene Expression (2002)
Queen Bees (2003)
Fish Swarms (2003)
Leaping Frogs (2003)
Social Behavior (2003)
Dendritic Cells (2005)
Nuclear Collision (2005)
Wasps (2005)



# A Cambrian Explosion of Metaphor Heuristics

2006-2007

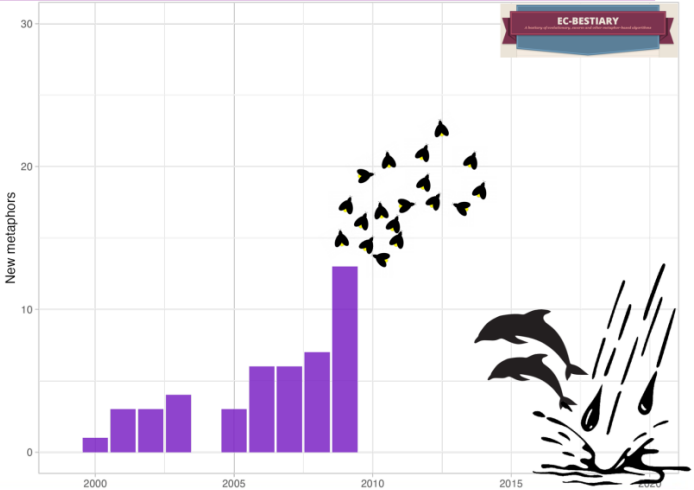
Bee Colonies (2006)
Big Bang (2006)
Cats (2006)
Invasive Weeds (2006)
Plant Saplings Growth (2006)
Small World (2006)
Bee Colonies (another one) (2007)
Central Force (2007)
Monkey Foraging (2007)
Parliamentarism Elections (2007)
Imperialism (2007)
River Formation (2007)



# A Cambrian Explosion of Metaphor Heuristics

2008-2009

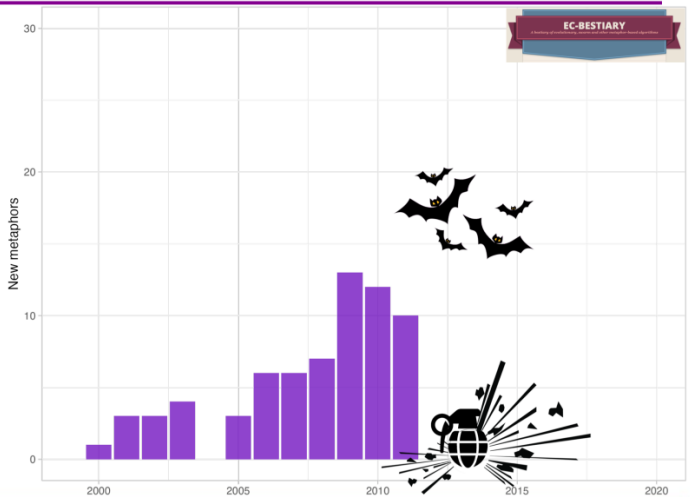
Bacterial Swarming (2008)
Biogeography (2008)
Fish Schools (2008)
Glow Worms (2008)
Roach Infestations (2008)
Slime Mold (2008)
Virus Replication (2008)
Animal Behavior: Searching (2009)
Bumblebees (2009)
Cuckoos (2009)
Dolphin Partners (2009)
Fireflies (2009)
Gravitation (2009)
Group Counselling (2009)
Locusts (2009)
Mountain Climbers (2009)
Paddy Fields (2009)
Sports Championships (2009)
Troops of Soldiers (2009)
Intelligent Water Drops (2009)



# A Cambrian Explosion of Metaphor Heuristics

2010-2011

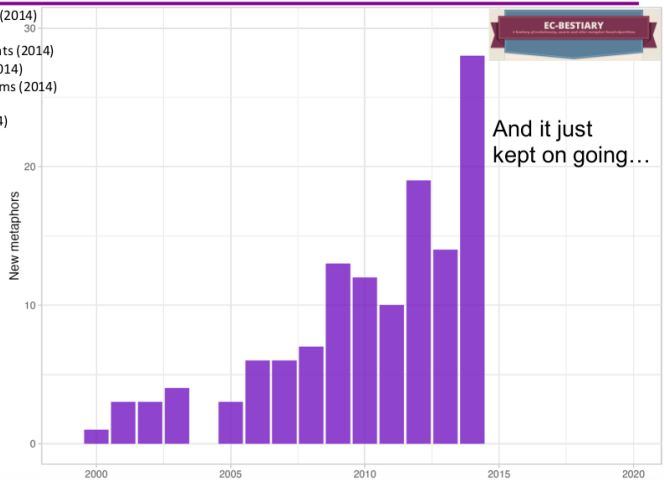
Bats (2010)
Charged Systems (2010)
Consultants (2010)
Eagles (2010)
Emotions (2010)
Fireworks (2010)
Grenades (2010)
Reincarnation (2010)
Social Behavior: Seeking (2010)
Termites (2010)
Water Flow (2010)
Wind (2010)
Brainstorming (2011)
Chemical Reactions (2011)
Ecology (2011)
Electrons Flowing (2011)
Catfish (2011)
Galaxies (2011)
Gravitational Interactions (2011)
Mosquitos: Egg-laying Behavior (2011)
Spirals (2011)
Teachers (2011)



# A Cambrian Explosion of Metaphor Heuristics

2012-2014

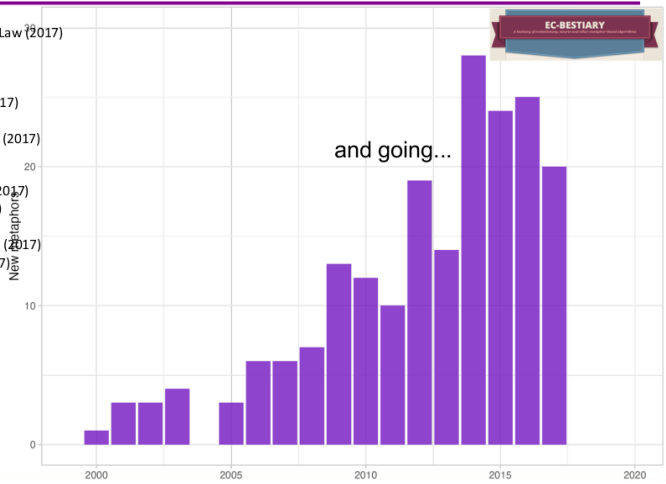
Anarchic Society (2012)
Bird Migrations (2012)
Community of scientists (2012)
Electromagnetism (2012)
Flower Pollination (2012)
Japanese Tree Frogs (2012)
Fruit Fly (2012)
Hoopoe (2012)
Krill (2012)
Lions (2012)
Mine Explosions (2012)
Pearl Hunting (2012)
Plants: Plant Growth (2012)
Rays of Light (2012)
Salmon Migrations (2012)
Swallows (2012)
Water Cycle (2012)
Wolves (2012)
Zombies (2012)
Magnetotactic Bacteria (2013)
Black Holes (2013)
Blind Naked Mole Rats (2013)
Clouds (2013)
Dogs (2013)
Dolphin Echolocation (2013)
Cuttlefish (2013)
Brownian Motion (2013)
Ladybirds (2013)
Penguins (2013)
Soccer Games (2013)
Social Spiders (2013)
Swine Flu (2013)
Vultures (2013)
Amoeba (2014)
Animals Hunting (2014)
Birds Mating (2014)
Chicken Swarms (2014)
Cockroaches (2014)
Colliding Bodies (2014)
Coral Reefs (2014)
Crystal Energy (2014)
Experts (2014)
Ecogeography (2014)
Tree Survival (2014)
Kinetic Energy (2014)
Heart (2014)
Interior Design (2014)
Keshtel Duck (2014)
Markets (2014)
Spider Monkeys (2014)
Pigeons (2014)
Plant Propagation (2014)
Political Strategies (2014)
Scientific Method (2014)
Sharks (2014)
Soccer Tournaments (2014)
Soccer Leagues (2014)
Symbiotic Organisms (2014)
Rain Drops (2014)
Grey Wolves (2014)
Worms (2014)



# A Cambrian Explosion of Metaphor Heuristics

2015-2017

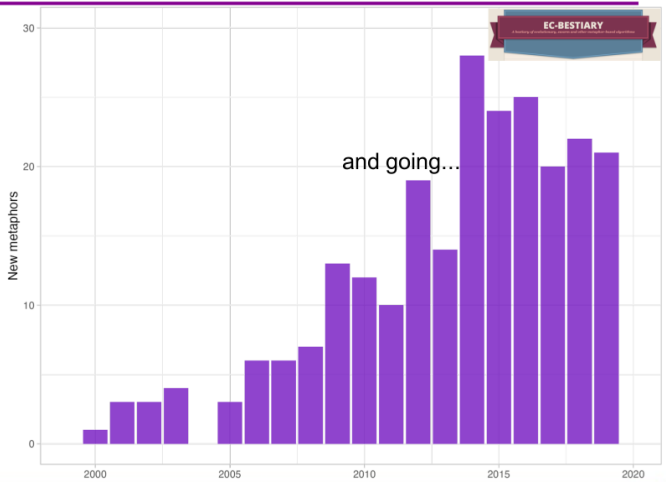
Algae (2015)
Animal Predation (2015)
Ant Lion (2015)
Monarch Butterflies (2015)
Cancers (2015)
Dragonflies (2015)
Earthworms (2015)
Elephants (2015)
Elephant Herds (2015)
Fractals (2015)
General Relativity (2015)
Ions (2015)
Jaguars (2015)
Lightning (2015)
Moths (2015)
Multiverse (2015)
Optics (2015)
Presidential Elections (2015)
Quantum Superposition (2015)
Roots (2015)
See-See Partridges (2015)
Viruses Attacking (2015)
Vortices (2015)
Water Waves (2015)
African Buffalo (2016)
Camels (2016)
Crows (2016)
Scottish Red Deer (2016)
Duelists (2016)
Epidemics (2016)
FIFA World Cup (2016)
Forest Regeneration (2016)
Galaxy Motion (2016)
Harris's Hawk (2016)
Hormones (2016)
Kestrels (2016)
Mosquitos Flying (2016)
Plant Intelligence (2016)
Rhinoceros (2016)
Hybrid Rice (2016)
Tug of War (2016)
Vibrating Particles (2016)
Virus Colonies (2016)
Virulence (2016)
Vehicles (2016)
Water Evaporation (2016)
Whales (2016)
Sperm Whales (2016)
Yin-Yang Pairs (2016)
Chicken Laying Eggs (2017)
Grasshoppers (2017)
Group Decisions (2017)
Selfish Herds (2017)
Hyenas (2017)
Kidneys (2017)
Newton's Cooling Law (2017)
Neurons (2017)
Polar Bears (2017)
Ravens (2017)
Salp Planktons (2017)
Sine Waves (2017)
Social Engineering (2017)
Sperm (2017)
Sonar (2017)
States of Matter (2017)
Vaccination (2017)
Rain (2017)
Hydrological Cycle (2017)
Killer Whales (2017)



# A Cambrian Explosion of Metaphor Heuristics

2018-2020

Plasmodium (2018)
Andean Condors (2018)
Anglerfish (2018)
Beetles (2018)
Bison (2018)
BrunsVigia Flower (2018)
Buses (2018)
Butterflies (2018)
Cheetah (2018)
Coyotes (2018)
Chaotic Crows (2018)
Farmland Fertility (2018)
Fish Mouth Brooding (2018)
Meerkats (2018)
Mushroom Reproduction (2018)
Emperor Penguins (2018)
Pigeons Feeding (2018)
Social Behavior: Queuing (2018)
Flying Squirrels (2018)
Volleyball Leagues (2018)
Binary Whales (2018)
Yellow Saddle Goldfish (2018)
Artillery (2019)
Barnacles Mating (2019)
Hitchcock Birds (2019)
Buzzards (2019)
Dice Games (2019)
Donkeys (2019)
Radar (2019)
Falcons (2019)
Human Life Choices (2019)
Humans Hunting (2019)
Search and Rescue (2019)
Wild Mice (2019)
Molecular Dynamics (2019)
Owls (2019)
Sailfish (2019)
Sandpiper (2019)
Seagulls (2019)
Hammerhead Sharks(2019)
Thieves and Police (2019)
Poor and Rich (2019)
Sooty Tern (2019)
Battle Royale Game (2020)
Black Widow (2020)
Bonobos (2020)
COVID19 (2020)
Border Collies (2020)
Horses (2020)
Students (2020)
Mayflies (2020)
Marine Predators (2020)
Soccer Style (2020)
Urbanization (2020)
Tunicates (2020)
Orcas (2020)
Whirlpools (2020)



# What is the problem with Metaphor Heuristics?

- Are there really over 200 ways to create Search Based Optimization algorithms?
- And are metaphors the correct way to organize this research?

# Problems with Metaphor Heuristics

## Endlessly Reinventing the Wheel

Because Metaphor Metaheuristics hide their details behind complex metaphors, it is difficult to tell what is similar, and what is different between these algorithms.

- "The amount of soil on the edges of the iteration-best solution is reduced based on the goodness of the solution"
- "This population is generated by a first shot explosion producing a number of individuals (shrapnel pieces)"
- "In this case the bait helps the Green Heron bird to catch a prey and thus the solution set elements remain constant"
- "The selection of two barnades is based on the length of their penises, pl. The selection process mimics the behaviour of barnacles. [It] is done randomly, but it will be restricted to the penis length of the barnacle"
- "The objective function was regarded as invariant to the reference frame, something like a transcendental entity in the space time"

# Problems with Metaphor Heuristics

## Spamming the literature, and salami science

- The literature is flooded with algorithms that are identical or very similar;
- These papers lack in reproducibility and analysis standards;
- People outside of the field can't see the entire literature, and end up picking the flashier names;
- Some suspicious activity, such as salami science and citation rings;

A paper using purely metaphor language with poor scientific standards will often be non-falsifiable;

# Pushback on Metaphor Algorithms

Fortunately, in recent years there has been a push back against this kind of practice.

- Kenneth Sorsen, 2015: "Metaheuristics, the Metaphor Exposed";
- Villalon, Weyland: Analysis of individual popular metaphors;
- Many people: "Metaphor-based Metaheuristics, a call for action"

- Some Journals have updated their policies to reject "Metaphor Heuristics" (Operations Research, Swarm Intelligence, TELO, Journal of Heuristics, and others)

Still, we ask ourselves: Then, how do we properly create new metaheuristics?

# Part III
# Component-Oriented Evolutionary Computation

# We still want new meta-heuristic optimizers

We don't want to create a new algorithm for everything,
but different problems require different strategies.

## Properties of the Fitness Landscape

- Number of Local Optima;
- Multi-modal problems;
- Separability of the parameters;

## Properties of the Problem Domain

- Dynamic Problems;
- Multi-Objective problems;
- Constraints;

So, we still need to create "new algorithms" from time to time. How to do that responsibly?

# Mixing and Matching: The Composition Approach

Instead of creating a **NEW ALGORITHM!**...

...how about re-using the knowledge that already exists?

Search-Based Optimizers have a common structure:

- Generate solutions;
- Test solutions;
- Modify solutions;

Can we populate this structure with “tricks” that were discovered in previous algorithms?

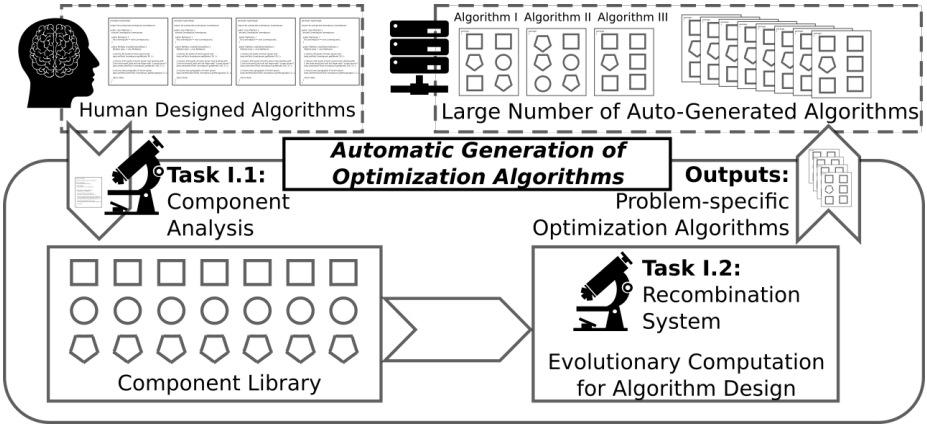
- The mutation from Evolutionary Strategies;
- The crossover from Genetic Algorithm;
- The velocity from PSO;
- etc...

This is the Compositional Approach of algorithm design!

# Imagine the Magic!

## Using the Composition Approach for Algorithm Design

# Well... actually it looks more like this




There is a lot to talk about! Today, we will focus on (1) How to compose the algorithm and (2) How to pick the set of components.

# How to Compose the Algorithm?

## The Parametric Approach

One way to automatically compose an optimization algorithm is to Treat the components as **Parameters** of the algorithm.

Then we can use standard algorithm configuration tools (such as iRace or SMAC) to choose the best components, given a set of test functions.

One early example of this approach is *"Automatic Component-Wise Design of Multiobjective Evolutionary Algorithms"* (Bezerra et al, 2015), where they set some of the components of an MOEA as parameters, and find that they can improve the performance of the algorithm by optimizing those.

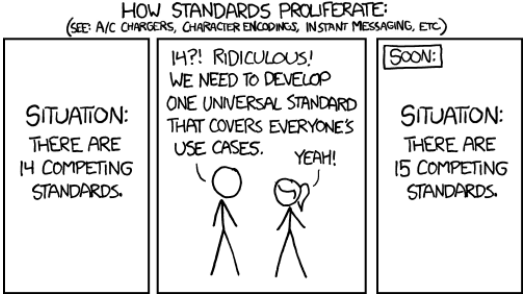
# One big problem in the Composition Approach: the coding!

It is fairly difficult to generate a flexible package that can compose different algorithms together at the same time.

**Why?** Many researchers use non-standard coding patterns (when the code is available) or descriptions (when it isn't).

It is actually good to do this work, though, as it develop healthy programming habits that us researchers don't usually have.

Me and Felipe campelo gave our contribution to this effort by creating **MOEADr**, a package that implements the MOEA/D algorithm in R, using a component-oriented design.



# The MOEADr Package

Campelo and Aranha, 2017–2021

The MOEADr is an R package (You can find it in CRAN!) that implements the MOEA/D algorithm, as well as many of its popular components.

The goal of the package is to allow specific components of the MOEA/D framework to be easily added or removed, to facilitate automated design.

Let's give a quick look at the main ideas.

# The MOEA/D Framework

The pseudocode below summarizes the MOEA/D algorithm.
The blue comments to the left indicate parts where we can replace components.

**Require:** Objective functions $\mathbf{f}(\cdot)$; Constraint functions $\mathbf{g}(\cdot)$;
**Require:** Component-specific input parameters;
$t \leftarrow 0$; *run* $\leftarrow$ TRUE
Generate initial population $\mathbf{X}^{(t)}$ by random sampling. ▷ Initialization
Generate weights $\Lambda$ ▷ Decomposition strategies
**while** *run* **do**
  Define or update neighborhoods $B$ ▷ Neighborhood assignment strategies
  Copy incumbent solution set $\mathbf{X}^{(t)}$ into $\mathbf{X}'^{(t)}$
  **for** each variation operator $\mathbf{v} \in \mathcal{V}$ **do**
    $\mathbf{X}'^{(t)} \leftarrow v(\mathbf{X}'^{(t)})$ ▷ Variation Stack
  **end for**
  Evaluate solutions in $\mathbf{X}^{(t)}$ and $\mathbf{X}'^{(t)}$ ▷ Aggregation functions and Constraint handling
  Define next population $\mathbf{X}^{(t+1)}$ ▷ Update strategies
  Update *run* flag; $t \leftarrow t + 1$ ▷ Stop criteria
**end while**
**return** $\mathbf{X^{(t)}}$; $\mathbf{f}\left(\mathbf{X}^{(t)}\right)$

# MOEA/D Component Classes

To facilitate the component design, we separate the MOEA/D component in three classes, and each class had a common structure.

- Decomposition Strategy
- Aggregation Function
- Objective Scaling Strategy
- Neighborhood Assignment Strategy
- Variation Operator Set
- Update Strategy
- Constraint Handling
- Termination Criteria

The components in black deal with the decomposition of a multi-objective problem into several single-objective problems. This is the main characteristic of MOEA/D.

The components in red perform the modification and update of solutions, based on their fitness values. There is a large variety of designs here, including operators from other algorithms. To deal with this variety, we designed a Variation Stack.

The components in Blue handle methods to modify the result of the fitness evaluation. Although this is usually problem specific, there are some common ways to handle constraints.

# Components Implemented by the MOEADr Package (v1.0.1)

| Component Class | Name | User Parameters |
|---|---|---|
| 3*Decomposition Method | SLD | $h \in \mathbb{Z}_{>0}$ |
| | MSLD | $\mathbf{h} \in \mathbb{Z}_{>0}^{K}; \tau \in (0,1]^{K}$ |
| | Uniform | $N \in \mathbb{Z}_{>0}$ |
| 5*Scalar Aggregation Function | WS | – |
| | WT | – |
| | AWT | – |
| | PBI | $\theta^{pbi} \in \mathbb{R}_{>0}$ |
| | iPBI | $\theta^{ipbi} \in \mathbb{R}_{>0}$ |
| Objective Scaling | – | $type \in \{none;\ simple\}$ |
| 2*Neighborhood Assignment | 2*– | $type \in \left\{by\ \lambda_i;\ by\ \mathbf{x}_i^{(t)}\right\}$ |
| | | $\delta_p \in [0,1]$ |

# Components Implemented by the MOEADr Package (v1.0.1)

| Component Class | Name | User Parameters |
|---|---|---|
| 9*Variation Operators | SBX recombination | $\eta_X \in \mathbb{R}_{>0}$; $p_X \in [0, 1]$ |
| | Polynomial mutation | $\eta_M \in \mathbb{R}_{>0}$; $p_M \in [0, 1]$ |
| | 2*Differential mutation | $\phi \in \mathbb{R}_{>0}$<br>$basis \in \{rand;\ mean;\ wgi\}$ |
| | Binomial recombination | $\rho \in [0, 1]$ |
| | Truncation | – |
| | 3*Local search | $type \in \{tpqa;\ dvls\}$<br>$\tau_{ls} \in \mathbb{Z}_{>0}$; $\gamma_{ls} \in [0, 1]$<br>$\epsilon \in \mathbb{R}_{>0}$ (if $type = tpqa$) |
| 3*Update Strategy | Standard | – |
| | Restricted | $nr \in \mathbb{Z}_{>0}$ |
| | Best | $nr \in \mathbb{Z}_{>0}$; $T_r \in \mathbb{Z}_{>0}$ |

# Components Implemented by the MOEADr Package (v1.0.1)

| Component Class | Name | User Parameters |
|---|---|---|
| 3*Constraint Handling | Penalty functions | $\beta_v \in \mathbb{R}_{>0}$ |
| | 2*VBR | $type \in \{ts;\ sr;\ vt\}$<br>$p_f \in [0, 1]$ (if $type = sr$) |
| 3*Termination Criteria | Evaluations | $max_{eval} \in \mathbb{Z}_{>0}$ |
| | Iterations | $max_{iter} \in \mathbb{Z}_{>0}$ |
| | Time | $max_{time} \in \mathbb{R}_{>0}$ |

We hope to include new components soon!

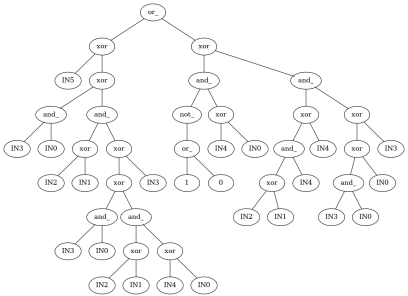
# Genetic Programming Approach

Another way to to approach the automatic design of algorithms is through Genetic Programming.

Genetic programming is an amazing technique that uses evolution for the design and improvements of programs. So the idea is:
Can we use Genetic Programming for the design of meta heuristics?

There is a big problem though:

Genetic programming still has difficulty with programs involving loops and complex data structures... what can we do?



# Using Grammatical Evolution to generate meta-heuristics

Bogdanova, Pereira, Aranha, 2019

We used Grammatical Evolution (GE) to automatically generate Swarm-like (DE, PSO, etc) heuristics.

Genetic Evolution uses a *formal grammar* to defines the general structure of a SWARM heuristic. This formal grammar can express:

- Commands;
- Loops and Recursion;
- Conditions / Stop conditions;

So it is more powerful than just treating components as parameters.

```
# Initialization, no branching
<s> ::= <init>{::}<call>

<init> ::= \nport numpy as np{::}from src \nport *{::}def ge(n, my_func, bounds,
dimension, max_nfe):{:Solution.setProblem(my_func, bounds, dimension, maximize=False)
{::}Solution.repair = <repair>{::}X = Solution.initialize(n){::}for Xl in X:
Xl.setX(op.init_random(*Solution.bounds, Solution.dimension)){::}Xl.getFitness() for
Xl in X{::}Solution.updateHistory(X){::}while Solution.nfe < max_nfe:{::}U = X{::}
<main>{::}[Xl.getFitness() for Xl in X]:}{::}return X:}{::}

# BRANCHING STARTS HERE
# main
<main> ::= <round>|<round>{::}'#Round Drop'{::}<drop>

<round> ::= '#Round 1'{::}<last_step>|
                    '#Round 1'{::}<step>{::}'#Round 2'{::}<last_step>|
                    '#Round 1'{::}<step>{::}'#Round 2'{::}<step>{::}'#Round 3'{::}
<last_step>

<last_step> ::= S1 = <select>{::}U  = <operator1>{::}X  = <output_last>{::}|
                          S1 = <select>{::}S2 = <select>{::}U  = <operator2>{::}
X  = <output_last>{::}|
                          S1 = <select>{::}S2 = <select>{::}S3 = <select>{::}U
= <operator3>{::}X  = <output_last>{::}

<step> ::=      S1 = <select>{::}U = <operator1>{::}X  = <output>{::}|
                          S1 = <select>{::}S2 = <select>{::}U  = <operator2>{::}X  =
<output>{::}|
                          S1 = <select>{::}S2 = <select>{::}S3 = <select>{::}U  =
<operator3>{::}X  = <output>{::}

<select>    ::= <select_random>|<select_tournament>|<select_current>#|
<select_roulette>
<operator1> ::= <op_pso>|<op_cs>|<op_mut_uni>
<operator2> ::= <op_crx_blend>|<op_crx_exp>|<op_crx_uni>
<operator3> ::= <op_mut_de>
<output>    ::= <update_all>|<update_best>|<update_better_than_random>|<update_later>
<output_last> ::= <update_all>|<update_best>|<update_better_than_random>
<drop>      ::= <drop_random>|<drop_worst>
<repair>    ::= <repair_random>|<repair_truncate>

#selection
<select_random>     ::= op.select_random(<X_or_U>, 1)
<select_tournament> ::= op.select_tournament(<X_or_U>, n=1, k=<k>)
<select_current>    ::= op.select_current(<X_or_U>)
<select_roulette>   ::= op.select_roulette(<X_or_U>)
<X_or_U> ::= X | U

#operators 1-1
<op_pso>     ::= op.w_pso(S1, w=<w>, c1=<c1>, c2=<c2>)
<op_cs>      ::= op.w_levy_flight(S1)
<op_mut_uni> ::= op.w_mut_uni(S1, pr=<pr>)
```

# Example of a Formal Grammar

Grammar:

```
S := A| A * A
A := A + A | A - A | B
B := X | Y | B * B
```

Examples of instances of this Grammar:

- Y
- X + X - Y
- (Y - XY) + (X - (Y + Y))

This example is very simple, but I hope it shows that grammars can generate complex combinations, and complex programs.

```
# Initialization, no branching
<S> ::= <init>{::}<call>

<init> ::= import numpy as np{::}from src import *{::}def ge(n, ny_func, bounds,
dimension, max_nfe):{:Solution.setProblem(my_func, bounds, dimension, maximize=False)
{::}Solution.repair = <repair>{::}X = Solution.initialize(n){::}for Xi in X:
Xi.setX(op.init_random(*Solution.bounds, Solution.dimension)){::}[Xi.getFitness() for
Xi in X]{::}Solution.updateHistory(X){::}while Solution.nfe < max_nfe:{:{::}U = X{::}
<main>{::}[Xi.getFitness() for Xi in X]:}{::}return X:}{::}

# BRANCHING STARTS HERE
# main
<main> ::= <round>|<round>{::}'#Round Drop'{::}<drop>

<round> ::= '#Round 1'{::}<last_step>|
                          '#Round 1'{::}<step>{::}'#Round 2'{::}<last_step>|
                          '#Round 1'{::}<step>{::}'#Round 2'{::}<step>{::}'#Round 3'{::}
<last_step>

<last_step> ::= S1 = <select>{::}U  = <operator1>{::}X  = <output_last>{::}|
                                    S1 = <select>{::}S2 = <select>{::}U  = <operator2>{::}
X  = <output_last>{::}|
                                    S1 = <select>{::}S2 = <select>{::}S3 = <select>{::}U
= <operator3>{::}X  = <output_last>{::}

<step> ::=      S1 = <select>{::}U = <operator1>{::}X  = <output>{::}|
                                    S1 = <select>{::}S2 = <select>{::}U  = <operator2>{::}X  =
<output>{::}|
                                    S1 = <select>{::}S2 = <select>{::}S3 = <select>{::}U  =
<operator3>{::}X  = <output>{::}

<select>    ::= <select_random>|<select_tournament>|<select_current>#|
<select_roulette>
<operator1> ::= <op_pso>|<op_cs>|<op_mut_un1>
<operator2> ::= <op_crx_blend>|<op_crx_exp>|<op_crx_un1>
<operator3> ::= <op_mut_de>
<output>    ::= <update_all>|<update_best>|<update_better_than_random>|<update_later>
<output_last> ::= <update_all>|<update_best>|<update_better_than_random>
<drop>      ::= <drop_random>|<drop_worst>
<repair>    ::= <repair_random>|<repair_truncate>

#selection
<select_random>     ::= op.select_random(<X_or_U>, 1)
<select_tournament> ::= op.select_tournament(<X_or_U>, n=1, k=<k>)
<select_current>    ::= op.select_current(<X_or_U>)
<select_roulette>   ::= op.select_roulette(<X_or_U>)
<X_or_U> ::= X | U

#operators 1-1
<op_pso>     ::= op.w_pso(S1, w=<w>, c1=<c1>, c2=<c2>)
<op_cs>      ::= op.w_levy_flight(S1)
<op_mut_un1> ::= op.w_mut_un1(S1, pr=<pr>)
```

# What did we achieve with Grammatical Evolution?

Things that worked:

- We found a good grammar and components that could generate popular swarm meta-heuristics;
- Using this grammar, GE could also generate “crazy” algorithms;
- The algorithms performed well on simple benchmarks;

Things that did not work:

- Grammatical evolution is **very** slow...
- It is difficult to fine-tune fitness: too hard, or too easy, and evolution stalls;
- How to choose components for more general algorithms?

# How to select components?

The two approaches that we described below rely on using a **common framework**, and selecting components for building this framework.

... But how do we select which components to add to these approaches?

# Selecting components from algorithm analysis

The obvious way to select components is to study many algorihtms and extract promising techniques from them.

But this is not a very good answer, there are many open questions!

- There are too many algorithms to study, and some of them are too similar;
- Which algorithms are more or less efficient?
- If the algorithms use different languages, how do we compare them?

# Algorithm Similarity Measures

This leads us to the question of...

How do we measure the similarity of two Search-Based Optimization Algorithms?

Why do we want to measure the similarity of algorithms?

- Identify groups of algorithms with useful components;
- Create an ensemble of algorithms with different characteristics;
- Select new algorithms when old ones are not performing well against a problem;

# Literature: Manual Similarity Analysis

Earlier work analyzed algorithms manually to find patterns, and classify them following those patterns:

- **Lones, 2014**: Define strategies for SBO's, and classify algorithms by these strategies. For example, Directional Search.
- **Lones, 2020**: Continues the above work by designing a standardised description of strategies and operators based on PSO.
- **Armas, 2021**: Defines a pool template (list of components) for algorithms, and measure similarity by the number of shared components in the template.

# Automated Similarity Analysis

Pereira and Aranha, 2022 (BIOMA)

Recently, we explored the automation of some of the work done by Lones and Armas.

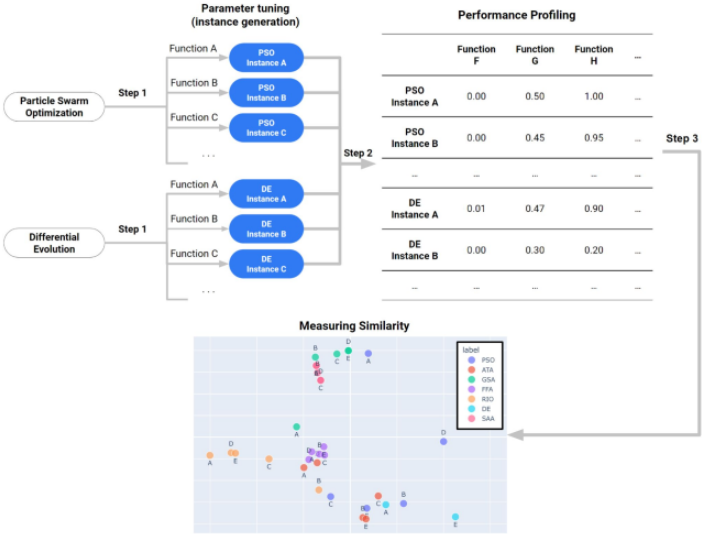
- There are just too many algorithms in the literature to compare manually (and new ones are proposed all the time!)
- If we are automating the composition of algorithms, it makes sense to automate the similarity measure too!

But what criteria do we use to compare algorithms?

- It is hard to automatically measure “algorithmic patterns”;
- On the other hand, pure performance could hide differences;
- Also, how do we measure the effect of changes in parameters?

# Automated Similarity Analysis

## Our current approach: Clustering Algorithm Instances (Pereira and Aranha, 2022)




**Key Idea:** Cluster-based method

1. Generate several instances of each algorithm by varying their parameters;
2. Obtain “features” by profiling these instances on carefully selected benchmark problems;
3. Cluster the algorithm instances based on the features, and analyze the result;

# Automated Similarity Analysis

## New Idea: Performance Profile

Performance Profile: Used to characterize an algorithm (or algorithmic instance). Composed of an array with the performance value on a set of benchmark problems.

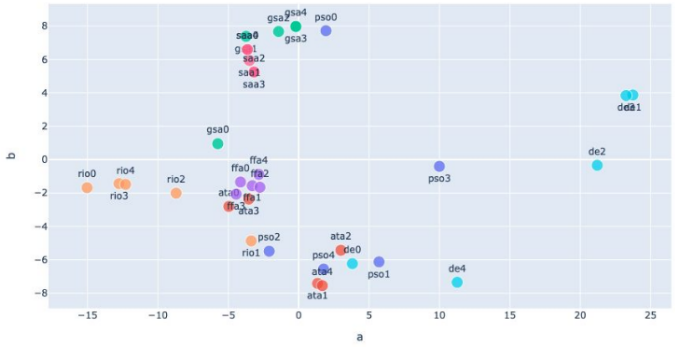
Performance Similarity: Measure to compare two algorithms based on their Performance Profile.

$$
\begin{aligned}
PS(A,B) &= \sqrt{\sum_{i=1}^{n}(e_i^A - e_i^B)^2} \\
e_i^X &= log_{10}(f^X\text{value}_i - f\text{opt}_i)
\end{aligned}
$$

- $f^X\text{value}_i$: the value obtained by algorithm $X$ on problem $i$.
- $f\text{opt}_i$: the optimum of problem $i$.

# Some Results

- PSO instances clustered poorly, FFA instances clustered well:
  - This indicates parameter dependence?
- Clustering limited by positive scores (low difficulty benchmarks?)
- Silhouette score used to evaluate cluter quality: Not a good metric here.



# This research generates several interesting “Open Questions”

- **Which metrics to use as attributes?**
  What are good metrics to clearly show the difference between algorithms, for a set of benchmarks?
- **How to select the functions for the Performance Profile?**
  Alternatively, what are good sets of benchmarks that show the difference between algorithms clearly?
- **Can we use this to evaluate the quality of benchmarks?**
  We create benchmark based on our theoretical knowledge about how blackbox problems should operate. But is this correct? Can we confirm that knowledge through this analysis?
- **Maybe automatically generate good benchmarks using this and GP?**
  Can we generate benchmark functions that maximize the distance between algorithms? Or maybe adversarially generate benchmarks that cause good algorithms to perform poorly?

# Part IV
# The End

# Final Thoughts

- Evolutionary Computation has been used, with success, to solve a variety of Optimization Problems;
- Because there are many different problems, we want to design specific algorithms for each of them;
- However, careless development of new algorithms can fragment our knowledge base;
- A more organized approach is component oriented design, which focuses on variations of algorithms using well known base frameworks;
- There are many challenges in the research and development of component-oriented EA, but it is a very fun challenge for people interested in programming.
- There is still a lot of work to do!

## Thank you for listening!

download this material:
https://github.com/caranha/Lecture_AlgorithmComposition