# Swarm Intelligence Algorithms

Rui Mendes

# Introduction

## Characteristics

- Inspired in social systems
- Borrows ideas from the organization of swarms, flocks, social psychology

## Examples

- Particle Swarm Optimization
- Jaya Algorithm
- Grey Wolf Optimization

# Concepts

- Optimization based on social interactions
- Uses population of individuals
- Most algorithms are using for real optimization
- They borrow ideas from the behavior of living systems

# Particle Swarm Optimization

- Created in 1995 by James Kennedy and Russell Eberhart [@blum2008swarm]
- Inspired in social psychology and bird flock simulation
- Uses a population of individuals
- Each individual has a position and a velocity
- Velocity is updated by:
  - Atraction to the best position it found in the past
  - Attraction to the best position found by the group

# PSO General scheme

1. Initialize population
2. Evaluate individuals
3. For each individual
    1. Choose individuals from neighborhood
    2. Imitate these individuals
    3. Update best performance if a better position was found
4. Iterate to 3 until stopping criterion is found

# PSO Algorithm

## Individuals' state

**Position** Current position $\vec{x}_i$

**Velocity** Current velocity $\vec{v}_i$

**Individualism** Previously best found position $\vec{p}_i$

**Conformism** Previously best found position by the group $\vec{p}_g$

## Algorithm

$$\begin{cases} \vec{v}_i = & \chi \left(\vec{v}_i + \vec{\mathrm{U}}[0,\varphi_1]\left(\vec{p}_i - \vec{x}_i\right) + \vec{\mathrm{U}}[0,\varphi_2]\left(\vec{p}_g - \vec{x}_i\right)\right) \\ \vec{x}_i = & \vec{x}_i + \vec{v}_i \end{cases}$$

# Initial version

- Initially, the algorithm was proposed with these equations:

$$\left\{ \begin{array}{ll} \vec{v}_i = & \vec{v}_i + \vec{\mathrm{U}}[0, \varphi_1]\,(\vec{p}_i - \vec{x}_i) + \vec{\mathrm{U}}[0, \varphi_2]\,(\vec{p}_g - \vec{x}_i) \\ \vec{x}_i = & \vec{x}_i + \vec{v}_i \end{array} \right.$$

# Maximum velocity

- The velocity often becomes very large
- To counter this effect, a new parameter was introduced: $V_{max}$
- This parameter prevents the velocity from becoming too large
- This parameter is usually coordinate-wise
- If it is too large, individuals fly past good solutions
- If it is too small, individuals explore too slowly and may become trapped in local optima
- Early experience showed that $\varphi_1$ and $\varphi_2$ could be set to 2 for almost all applications and only $V_{max}$ needed to be adjusted

# Innertial Weight

$$\begin{cases} \vec{v}_i = & \alpha \left(\vec{v}_i + \vec{\mathrm{U}}[0, \varphi_1] \left(\vec{p}_i - \vec{x}_i\right) + \vec{\mathrm{U}}[0, \varphi_2] \left(\vec{p}_g - \vec{x}_i\right)\right) \\ \vec{x}_i = & \vec{x}_i + \vec{v}_i \end{cases}$$

- $\varphi_1$ and $\varphi_2$ were usually set to 2.1
- The *alpha* parameter was uniformly varied between 0.9 and 0.4.

# Constriction Coefficient

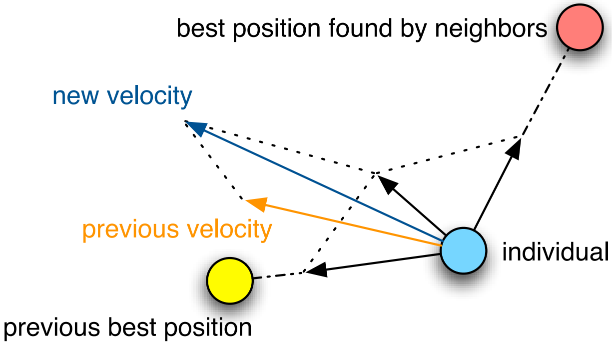
- Clerc proposed a version similar to the innertia weight
- The coefficient has a fixed value instead of a varying one
- The theoretical study seemed to indicate that a setting of all the parameters was enough to guarantee convergence without explosion or oscillation behaviors

Parameters $\varphi_1 = \varphi_2 = 2.05$, $\chi = 0.729$

## Algorithm

$$\begin{cases} \vec{v}_i = & \chi\left(\vec{v}_i + \vec{\mathrm{U}}[0, \varphi_1]\left(\vec{p}_i - \vec{x}_i\right) + \vec{\mathrm{U}}[0, \varphi_2]\left(\vec{p}_g - \vec{x}_i\right)\right) \\ \vec{x}_i = & \vec{x}_i + \vec{v}_i \end{cases}$$

# Solution Generation in PSO



# Premature Convergence in PSO

- The global best individual in the population often degrades PSO performance
- Convergence is fast
- Diversity is lost fast
- This leads to premature convergence
- It is better to use strategies to decrease the information flow

# What are the causes of premature convergence?

Optimization is a balance between two factors:

exploration
: The ability to explore the search space to find promising areas

exploitation
: The ability to concentrate on the promising areas of the search space

- An algorithm with too much exploration isn't *efficient*
- An algorithm with too much exploitation loses *diversity* fast
- If an algorithm doesn't have enough diversity, it will quickly stagnate

# Neighborhood concept

- Individuals imitate their (most successful) neighbors
- His neighbors will only influence their neighbors once they become sufficiently successful
- This favours clustering: different social neighborhoods may explore different areas of the search space
- Immediacy may be based on:

**Proximity** Proximity in Cartesian space

**Social** To share social bonds

# Example of Good Topologies

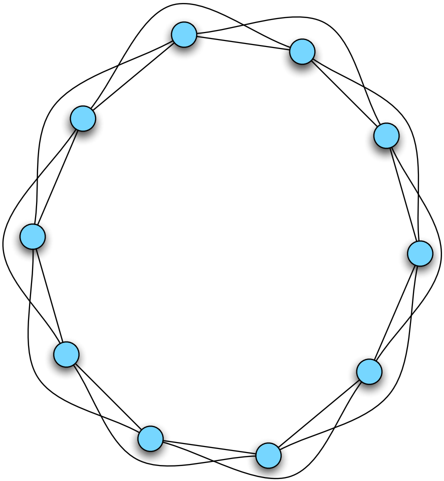


Figure 1: LBest 2

# Example of Good Topologies

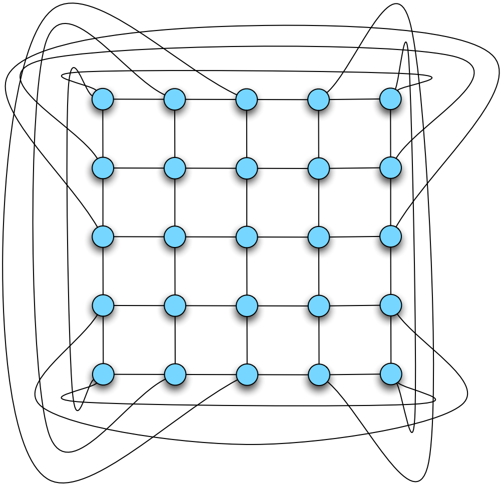


Figure 2: von Neumann or Square

# Discussion

- PSO is easy to implement
- The canonical version has a setting for both $\varphi_1$ and $\varphi_2$ and *chi*
- Thus, the only parameters that need to be changed are the population size and the number of function evaluations
- Any of the population topologies given above works well

# Fully Informed Particle Swarm

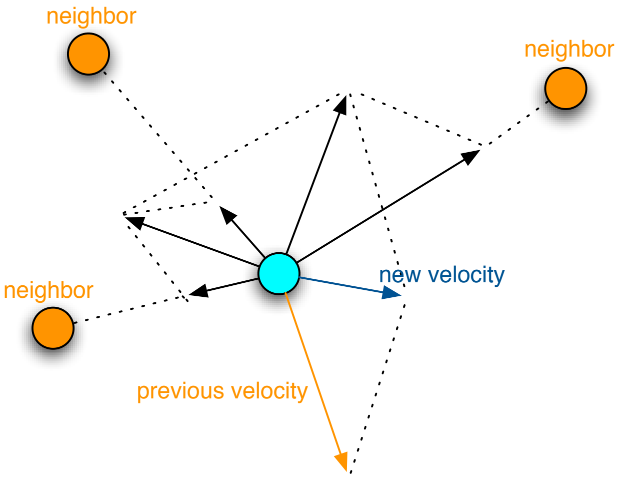
## Characteristics

- All contributions of the neighborhood are used
- Individual imitates the social norm
- The social norm is the center of gravity
- $\vec{p}_k$ is the best position of neighbor $k$
- $\mathcal{N}$ is the set of neighbors

## Algorithm

$$\begin{cases} \vec{v}_{t+1} = & \chi \left( \vec{v}_t + \frac{\sum_{k \in \mathcal{N}} \vec{\mathrm{U}}[0, \varphi_{max}](\vec{p}_k - \vec{x}_t)}{|\mathcal{N}|} \right) \\ \vec{x}_{t+1} = & \vec{x}_t + \vec{v}_{t+1} \end{cases}$$

# Solution Generation in FIPS



# Differences between Canonical PSO and FIPS

- There is no self contribution
- All individuals in the neighborhood contribute to the influence
- The number of individuals used is very important: A few contributions, typically between 2 and 4 are best
- Given a well chosen population topology, it outperforms the canonical model

# Discussion

- FIPS is easy to implement
- It often has better performance than the canonical PSO
- Parameter settings are the same as for PSO, and thus are fixed
- Thus, the only parameters that need to be changed are the population size and the number of function evaluations
- Any of the population topologies given above works well

# Bare Bones PSO

## Characteristics

- Use a probability distribution instead of velocity update
- Each particle explores the position between its personal best and the global best
- The step is adaptive and uses the standard deviation

## Algorithm

$$x_i = \mathcal{N}(\frac{p_i + p_g}{2}, |p_i - p_g|)$$

# Jaya Optimization

## Characteristics

- Similar to PSO
- Fewer control parameters
- Has a component towards the best point found
- Has a component away from the worst point found
- Update is elitist (i.e., new position is only used if it is better than the current one)

## Algorithm

$$x_i = x_i + \mathrm{U}[0,1]\,(x_{best} - |x_i|) - \mathrm{U}[0,1]\,(x_{worst} - |x_i|)$$

# Discussion

- Jaya is similar to PSO
- The best and worst performers influence the search
- The only parameters that need to be changed are the population size and the number of function evaluations
- It needs further study to ascertain its performance and if the formulas can be simplified

# Grey Wolf Optimization

## Characteristics

- Idea is quite similar to FIPS
- Best three solutions ($\alpha$, $\beta$, $\delta$) found guide the search
- New positions are generated by a random combination of the three positions
- The parameter $a$ is linearly decreased from 2 to 0

## Algorithm

$$
\begin{aligned}
\vec{A}_k &= \vec{\mathrm{U}}[-a, a] && \text{where } k \in \{\alpha, \beta, \delta\} \\
\vec{C}_k &= \vec{\mathrm{U}}[0, 2] && \text{where } k \in \{\alpha, \beta, \delta\} \\
\vec{D}_k &= |\vec{C}_k \otimes \vec{X}_k - \vec{X}| && \text{where } k \in \{\alpha, \beta, \delta\} \\
\vec{P}_k &= \vec{X}_k - \vec{A}_k \otimes \vec{D}_k && \text{where } k \in \{\alpha, \beta, \delta\} \\
\vec{X}_{t+1} &= \frac{\vec{P}_\alpha + \vec{P}_\beta + \vec{P}_\delta}{3}
\end{aligned}
$$

# Solution Generation in GWO

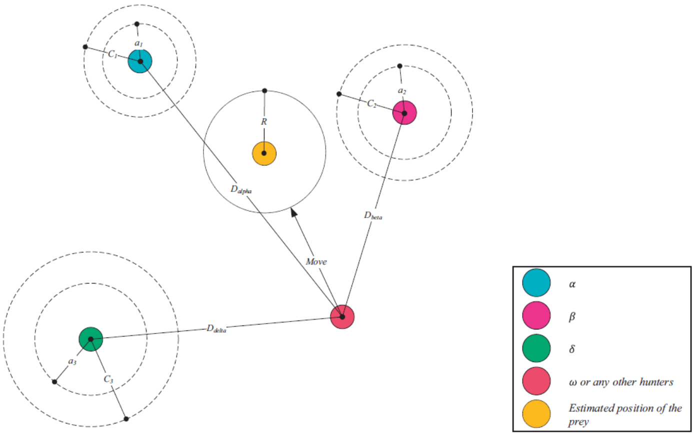



Figure taken from https://doi.org/10.1016/j.advengsoft.2013.12.007

# Discussion

- GWO is similar to FIPS
- The 3 best individuals guide the search
- Like FIPS, new solutions are generated by a stochastic barycenter
- The only parameters that need to be changed are the population size and the number of function evaluations
- It needs further study to ascertain its performance and if the formulas can be simplified
- It seems to only work well when the best solution is zero
- There is ongoing research for solving this tendency

# The No Free Lunch Theorem for Search

- All search algorithms perform equally over all possible problems

# The No Free Lunch Theorem for Search

- Configuration parameters must be fine tuned for different problems
- Some algorithms simply aren't that good at finding good solutions over all possible problem classes
- When attacking a new problem, we need to choose:
  - the best algorithm and
  - the most suitable parameters

# Selecting the best algorithm

- This is a meta optimization task
- Select the best algorithm
- Select the best parameters
- Evaluate each solution by running the algorithm on the optimization problem

# Grammatical Evolution

- Genomes encode a solution that is decoded using a grammar that defines the program
- Can evolve programs in any language or complexity
- Any structure that can be specified by a grammar can be evolved
- Uses BNF Grammars consisting of the tuple $< T, N, P, S >$ where

  T : is the set of Terminals

  N : is the set of Non-Terminal

  P : is the set of Production Rules

  S : is the start symbol ($S \in N$)

# BNF Example

```
<expr> ::= <expr> <op> <expr>
    |   (<expr> <op> <expr>)
    |   <fun> (<expr>)
    |   <var>
<op>   ::= + | - | * | /
<fun>  ::= Sin | Cos | Tan
<var>  ::= X
```

# Advantages

- Genotype to phenotype mapping
- Genotype is usually a sequence of integers
- Phenotype is a valid phrase in a given grammar
- This may be a program
- Genome indicates how the program is built using the BNF grammar
- Anything that can be specified with a gramar can be evolved, like:
  - programs
  - neural networks
  - graphs
- Can handle different types (e.g., Boolean, Integer, String)

# Genotype to phenotype encoding

- Several genes map to the same phenotype
- Genome is usually a sequence of integers
- Each integer indicates which production to choose

# Example

- Given the individual:

  280 45 127 29 59 163

- The NT <expr> has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
        |  (<expr> <op> <expr>)
        |  <fun> (<expr>)
        |  <var>
```

- Using the first codon 280, we get $280 \mod 4 = 0$
- Thus, we will use <expr> <op> <expr>

# Example

- Now we have `<expr> <op> <expr>`
- The first NT is `<expr>`
- The remaining chromosome is:

  280 45 127 29 59 163

- The NT `<expr>` has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
        |  (<expr> <op> <expr>)
        |  <fun> (<expr>)
        |  <var>
```

- Using the next codon 45, we get $45 \mod 4 = 1$
- Thus, we will use `(<expr> <op> <expr>)`

# Example

- Now we have `(<expr> <op> <expr>) <op> <expr>`
- The next NT is `<expr>`
- The remaining chromosome is:

  280 45 127 29 59 163

- The NT `<expr>` has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
     |   (<expr> <op> <expr>)
     |   <fun> (<expr>)
     |   <var>
```

- Using the next codon 127, we get $127 \mod 4 = 3$
- Thus, we will use `<var>`

# Example

- Now we have (<var> <op> <expr>) <op> <expr>
- The next NT is <var>
- The remaining chromosome is:

  280 45 127 29 59 163

- The NT <var> has 1 production rule:

  `<var> ::= X`

- Using the next codon 29, we get $29 \mod 1 = 0$
- Thus, we will use X

# Example

- Now we have `(X <op> <expr>) <op> <expr>`
- The next NT is `<op>`
- We continue using the chromosome:

  280 45 127 29 59 163

- The NT has 4 production rules:

  `<op>    ::= + | - | * | /`

- Using the next codon 59, we get $59 \mod 4 = 3$
- Thus, we will use `*`

# Example

- Now we have `(X * <expr>) <op> <expr>`
- The next NT is `<expr>`
- We continue using the chromosome:

  280 45 127 29 59 163

- The NT has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
       |  (<expr> <op> <expr>)
       |  <fun> (<expr>)
       |  <var>
```

- Using the next codon 163, we get $163 \mod 4 = 3$
- Thus, we will use `<var>`

# Example

- Now we have `(X * <var>) <op> <expr>`
- The next NT is `<var>`
- We reuse the chromossome:

  280 45 127 29 59 163

- The NT `<var>` has 1 production rule:

  `<var> ::= X`

- Using the next codon 280, we get $280 \mod 1 = 0$
- Thus, we will use `X`

# Example

- Now we have `(X * X) <op> <expr>`
- The next NT is `<op>`
- We continue using the chromosome:

  280 45 127 29 59 163

- The NT has 4 production rules:

  `<op>    ::= + | - | * | /`

- Using the next codon 45, we get $45 \mod 4 = 1$
- Thus, we will use `-`

# Example

- Now we have `(X * X) - <expr>`
- The next NT is `<expr>`
- We continue using the chromosome:

  280 45 127 29 59 163

- The NT has 4 production rules:

```
<expr> ::= <expr> <op> <expr>
    |   (<expr> <op> <expr>)
    |   <fun> (<expr>)
    |   <var>
```

- Using the next codon 127, we get $127 \mod 4 = 3$
- Thus, we will use `<var>`

# Example

- Now we have `(X * X) - <var>`
- The next NT is `<var>`
- We continue using the chromosome:

  280 45 127 29 59 163
- The NT has 1 production rule:

  `<var> ::= X`
- Using the next codon 29, we get $29 \mod 1 = 0$
- Thus, we will use `<X>`

# Example

- Thus, the phenotype corresponding to the chromosome

  280 45 127 29 59 163

- is `(X * X) - X`

# Notes

- Each of the chromosomes depends on context
- The value 127 would mean selecting:
  - the second value for `<fun>` (it has 3 values)
  - the last value for `<expr>` (it has 4 values)
- The value 115 has the same properties
- Therefore, if 127 is mutated ito 115, it is a *silent mutation*

# Initialization

- Use **Ramped half and half**
- Individuals are generated in such a way that their derivation trees follow *Ramped half and half*
- For each node of the derivation tree
  - Record which choice was made for each production
  - Generate a number that, following the mod rule, gives the correct production

# Genetic operators

- The operators used in the genetic algorithms can be used:
  - One point crossover
  - Point mutation

# Parameters

- Same as GA
  - Population size
  - Number of iterations
  - Genome size
  - Genetic operator probabilities
  - Fitness function
- Ways of limiting individuals' complexity:
  - Maximum depth of the derivation tree
  - Maximum number of wrappings

# Part of the grammar for our problem

```
<expr> ::= barebonesPso(<BB>) | canonicalPso(<PSO>) |
        differentialEvolution(<DE>)

<DE> ::= {"populationSize": <populationSize>,
        "crossoverProbability": <crossoverProbability>,
        "differentialWeight": <differentialWeight>}

<crossoverProbability> ::= 0.<int><int> | 1
<differentialWeight> ::= 0.<int><int> | 1.<int><int> | 2
<populationSize> ::= 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80
<int> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

# Experiments

## Algorithms

- Canonical PSO
- Barebones PSO
- Differential Evolution
- Cuckoo Search
- Artificial Bee Colony

## Problems

- 10 optimization problems
- Both unimodal and multimodal

## Fitness function

- Run each algorithm along with the configuration parameters for a fixed number of evaluations
- Record the fitness of the best solution found for the optimization problem
- Use that fitness as the fitness for the GE individual

## Experiments

1. Choose best parameters for each algorithm and problem
2. Choose both the algorithm and the best parameters for each problem

# Results

## Experiment 1

- Except in two problems, all algorithms were improved when tweeking their configuration parametes for the specific problem
- In those two problems, results were equally good (it is probably not possible to improve them)

## Experiment 2

- No algorithm was always chosen for all problems
- Some algorithms were particularly better suited in some problems and were always chosen
- Letting our approach choose both the algorithm and its parameters was always a good approach

# Results

Table 1: Number of times GE chose an algorithm to solve a test problem.

| Problem | PSO | BB | ABC | CS | DE |
|---|---|---|---|---|---|
| Sphere ($f_1$) | 0 | 9 | 0 | 21 | 0 |
| Ackley ($f_2$) | 0 | 21 | 0 | 6 | 3 |
| Griewank ($f_3$) | 1 | 18 | 6 | 1 | 4 |
| Rastrigin ($f_4$) | 0 | 0 | 27 | 0 | 3 |
| Schwefel ($f_5$) | 0 | 0 | 5 | 0 | 25 |
| Rosenbrock ($f_6$) | 4 | 17 | 2 | 6 | 1 |
| Michalewicz ($f_7$) | 0 | 0 | 8 | 0 | 22 |
| Easom ($f_8$) | 7 | 8 | 5 | 2 | 8 |
| DeJong3 ($f_9$) | 4 | 8 | 9 | 3 | 6 |
| DeJong5 ($f_{10}$) | 0 | 18 | 2 | 1 | 9 |

# Conclusions

- NFL indicates that not all algorithms are equally suited for all optimization problems
- We aim to develop a tool that can help us choose/create the most suitable metaheuristic for a specific problem
- Results are encouraging