Resumen capitulo 3 del Sean Luke - Essentials of Metaheuristics

Algorithm 18 *The* (μ, λ) *Evolution Strategy*

```
μ ← number of parents selected
λ ← number of children generated by the parents

P ← {}
for λ times do                                        ▷ Build Initial Population
    P ← P ∪ {new random individual}
Best ← □
repeat
    for each individual P_i ∈ P do
        AssessFitness(P_i)
        if Best = □ or Fitness(P_i) > Fitness(Best) then
            Best ← P_i
    Q ← the μ individuals in P whose Fitness( ) are greatest      ▷ Truncation Selection
    P ← {}                       ▷ Join is done by just replacing P with the children
    for each individual Q_j ∈ Q do
        for λ/μ times do
            P ← P ∪ {Mutate(Copy(Q_j))}
until Best is the ideal solution or we have run out of time
return Best
```

como cambia al variar parametros

- The size of λ. This essentially controls the sample size for each population, and is basically the same thing as the n variable in Steepest-Ascent Hill Climbing With Replacement. At the extreme, as λ approaches ∞, the algorithm approaches exploration (random search).
- The size of μ. This controls how *selective* the algorithm is; low values of μ with respect to λ push the algorithm more towards exploitative search as only the best individuals survive.
- The degree to which Mutation is performed. If Mutate has a lot of noise, then new children fall far from the tree and are fairly random regardless of the selectivity of μ.

Algorithm 19 *The $(\mu + \lambda)$ Evolution Strategy*

$\mu \leftarrow$ number of parents selected
$\lambda \leftarrow$ number of children generated by the parents

$P \leftarrow \{\}$
for λ times **do**
    $P \leftarrow P \cup \{$new random individual$\}$
$Best \leftarrow \square$
repeat
    for each individual $P_i \in P$ **do**
        AssessFitness(P_i)
        if $Best = \square$ or Fitness(P_i) $>$ Fitness($Best$) **then**
            $Best \leftarrow P_i$
    $Q \leftarrow$ the μ individuals in P whose Fitness( ) are greatest
    $P \leftarrow Q$ ▷ The Join operation is the only difference with (μ, λ)
    for each individual $Q_j \in Q$ **do**
        for λ / μ times **do**
            $P \leftarrow P \cup \{$Mutate(Copy(Q_j))$\}$
until $Best$ is the ideal solution or we have run out of time
return $Best$

tiene el riesgo de explotar mucho a los padres y quedarse en un minimo local

3.1.1 Mutation and Evolutionary Programming

Evolution Strategies historically employ a representation in the form of a fixed-length vector of real-valued numbers. Typically such vectors are initialized using something along the lines of Algorithm 7. Mutation is typically performed using Gassian Convolution (Algorithm 11).

Gaussian Convolution is controlled largely by the distribution variance σ^2. The value of σ^2 is known as the **mutation rate** of an ES, and determines the noise in the Mutate operation. How do you pick a value for σ^2? You might pre-select its value; or perhaps you might slowly decrease the value; or you could try to adaptively change σ^2 based on the current statistics of the system. If the system seems to be too exploitative, you could increase σ^2 to force some more exploration (or likewise decrease it to produce more exploitation). This notion of changing σ^2 is known as an **adaptive mutation rate**. In general, such **adaptive** breeding operators adjust themselves over time, in response to statistics gleaned from the optimization run.[16]

distintas formas de ir variando la varianza a través del metodo llevan a distintos resultados
El algoritmo 7 mencionado es como armar un vector random y el algoritmo 11 seria este

Algorithm 11 *Gaussian Convolution*

$\vec{v} \leftarrow$ vector $\langle v_1, v_2, \ldots v_l \rangle$ to be convolved
$p \leftarrow$ probability of adding noise to an element in the vector ▷ Often $p = 1$
$\sigma^2 \leftarrow$ variance of Normal distribution to convolve with ▷ Normal = Gaussian
$min \leftarrow$ minimum desired vector element value
$max \leftarrow$ maximum desired vector element value

for i from 1 to l **do**
  if $p \geq$ random number chosen uniformly from 0.0 to 1.0 **then**
    repeat
      $n \leftarrow$ random number chosen from the Normal distribution $N(0, \sigma^2)$
    until $min \leq v_i + n \leq max$
    $v_i \leftarrow v_i + n$
return $\vec{v}$

One old rule for changing σ^2 adaptively is known as the **One-Fifth Rule**, by Ingo Rechenberg,[17] and it goes like this:

- If more than $\frac{1}{5}$ children are fitter than their parents, then we're exploiting local optima too much, and we should increase σ^2.
- If less than $\frac{1}{5}$ children are fitter than their parents, then we're exploring too much, and we should decrease σ^2.
- If exactly $\frac{1}{5}$ children are fitter than their parents, don't change anything.

reglas tipicas

Algorithm 20 *The Genetic Algorithm (GA)*

$popsize \leftarrow$ desired population size ▷ This is basically λ. Make it even.

$P \leftarrow \{\}$
for $popsize$ times **do**
  $P \leftarrow P \cup \{$new random individual$\}$
$Best \leftarrow \square$
repeat
  for each individual $P_i \in P$ **do**
    AssessFitness(P_i)
    if $Best = \square$ or Fitness(P_i) $>$ Fitness($Best$) **then**
      $Best \leftarrow P_i$
  $Q \leftarrow \{\}$ ▷ Here's where we begin to deviate from (μ, λ)
  for $popsize/2$ times **do**
    Parent $P_a \leftarrow$ SelectWithReplacement(P)
    Parent $P_b \leftarrow$ SelectWithReplacement(P)
    Children $C_a, C_b \leftarrow$ Crossover(Copy(P_a), Copy(P_b))
    $Q \leftarrow Q \cup \{$Mutate(C_a), Mutate(C_b)$\}$
  $P \leftarrow Q$ ▷ End of deviation
until $Best$ is the ideal solution or we have run out of time
return $Best$

como funciona la mutacion

Algorithm 22 *Bit-Flip Mutation*

$p \leftarrow$ probability of flipping a bit ▷ Often p is set to $1/l$
$\vec{v} \leftarrow$ boolean vector $\langle v_1, v_2, ... v_l \rangle$ to be mutated

for i from 1 to l **do**
  if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
    $v_i \leftarrow \neg(v_i)$
return $\vec{v}$

3 opciones de crossover

Algorithm 23 *One-Point Crossover*

$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ... v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ... w_l \rangle$ to be crossed over

$c \leftarrow$ random integer chosen uniformly from 1 to l inclusive
for i from c to l **do**
  Swap the values of v_i and w_i
return $\vec{v}$ and $\vec{w}$

Algorithm 24 *Two-Point Crossover*

$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ... v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ... w_l \rangle$ to be crossed over

$c \leftarrow$ random integer chosen uniformly from 1 to l inclusive
$d \leftarrow$ random integer chosen uniformly from 1 to l inclusive
if $c > d$ **then**
  Swap c and d
for i from c to $d - 1$ **do**
  Swap the values of v_i and w_i
return $\vec{v}$ and $\vec{w}$

Algorithm 25 *Uniform Crossover*

$p \leftarrow$ probability of swapping an index ▷ Often p is set to $1/l$. At any rate, $p \leq 0.5$
$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ... v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ... w_l \rangle$ to be crossed over

for i from 1 to l **do**
  if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
    Swap the values of v_i and w_i
return $\vec{v}$ and $\vec{w}$

Hacer crossover en la poblacion P no te deja salir de un hipercubo por lo que es necesario realizar la mutacion para explorar otras partes del espacio. El crossover hace que se esparzan los buildingblocks por la poblacion.
Por lo general las genes no son independientes entre si, tenerlo en cuanta a la hora de trabajar.

ahora vamos a mezclar no solo dos sino varios vectores a la vez

Algorithm 26 *Randomly Shuffle a Vector*

$\vec{p} \leftarrow$ elements to shuffle $\langle p_1, ..., p_l \rangle$

for i from l down to 2 **do** ▷ Note we don't do 1
  $j \leftarrow$ integer chosen at random from 1 to i inclusive
  Swap p_i and p_j

Algorithm 27 *Uniform Crossover among K Vectors*

$p \leftarrow$ probability of swapping an index ▷ Ought to be very small
$W \leftarrow \{W_1, ..., W_k\}$ vectors to cross over, each of length l

$\vec{v} \leftarrow$ vector $\langle v_1, ..., v_k \rangle$
for i from 1 to l **do**
  if $p \geq$ random number chosen uniformly from 0.0 to 1.0 inclusive **then**
    for j from 1 to k **do** ▷ Load $\vec{v}$ with the ith elements from each vector in W
      $\vec{w} \leftarrow W_j$
      $v_j \leftarrow w_i$
    Randomly Shuffle $\vec{v}$
    for j from 1 to k **do** ▷ Put back the elements, all mixed up
      $\vec{w} \leftarrow W_j$
      $w_i \leftarrow v_j$
      $W_j \leftarrow \vec{w}$
return W

si estoy trabajando con floats

Algorithm 28 *Line Recombination*

$p \leftarrow$ positive value which determines how far long the line a child can be located ▷ Try 0.25
$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

$\alpha \leftarrow$ random value from $-p$ to $1 + p$ inclusive
$\beta \leftarrow$ random value from $-p$ to $1 + p$ inclusive
for i from 1 to l **do**
  $t \leftarrow \alpha v_i + (1 - \alpha) w_i$
  $s \leftarrow \beta w_i + (1 - \beta) v_i$
  if t and s are within bounds **then**
    $v_i \leftarrow t$
    $w_i \leftarrow s$
return $\vec{v}$ and $\vec{w}$

Algorithm 29 *Intermediate Recombination*

$p \leftarrow$ positive value which determines how far long the line a child can be located ▷ Try 0.25
$\vec{v} \leftarrow$ first vector $\langle v_1, v_2, ...v_l \rangle$ to be crossed over
$\vec{w} \leftarrow$ second vector $\langle w_1, w_2, ...w_l \rangle$ to be crossed over

for i from 1 to l **do**
  repeat
    $\alpha \leftarrow$ random value from $-p$ to $1+p$ inclusive ▷ We just moved these two lines!
    $\beta \leftarrow$ random value from $-p$ to $1+p$ inclusive
    $t \leftarrow \alpha v_i + (1-\alpha) w_i$
    $s \leftarrow \beta w_i + (1-\beta) v_i$
  until t and s are within bounds
  $v_i \leftarrow t$
  $w_i \leftarrow s$
return $\vec{v}$ and $\vec{w}$

ahora vamos a ver como se selecciona

Algorithm 30 *Fitness-Proportionate Selection*

perform once per generation
  global $\vec{p} \leftarrow$ population copied into a vector of individuals $\langle p_1, p_2, ..., p_l \rangle$
  global $\vec{f} \leftarrow \langle f_1, f_2, ..., f_l \rangle$ fitnesses of individuals in $\vec{p}$ in the same order as $\vec{p}$ ▷ Must all be ≥ 0
  if $\vec{f}$ is all 0.0s **then** ▷ Deal with all 0 fitnesses gracefully
    Convert $\vec{f}$ to all 1.0s
  for i from 2 to l **do** ▷ Convert $\vec{f}$ to a CDF. This will also cause $f_l = s$, the sum of fitnesses.
    $f_i \leftarrow f_i + f_{i-1}$
perform each time
  $n \leftarrow$ random number from 0 to f_l inclusive
  for i from 2 to l **do** ▷ This could be done more efficiently with binary search
    if $f_{i-1} < n \leq f_i$ **then**
      return p_i
  return p_1

Algorithm 31 *Stochastic Universal Sampling*

perform once per n individuals produced ▷ Usually $n = l$, that is, once per generation
  global $\vec{p} \leftarrow$ copy of vector of individuals (our population) $\langle p_1, p_2, ..., p_l \rangle$, shuffled randomly
  ▷ To shuffle a vector randomly, see Algorithm 26
  global $\vec{f} \leftarrow \langle f_1, f_2, ..., f_l \rangle$ fitnesses of individuals in $\vec{p}$ in the same order as $\vec{p}$ ▷ Must all be ≥ 0
  global $index \leftarrow 0$
  if $\vec{f}$ is all 0.0s **then**
    Convert $\vec{f}$ to all 1.0s
  for i from 2 to l **do** ▷ Convert $\vec{f}$ to a CDF. This will also cause $f_l = s$, the sum of fitnesses.
    $f_i \leftarrow f_i + f_{i-1}$
  global $value \leftarrow$ random number from 0 to f_l / n inclusive
perform each time
  while $f_{index} < value$ **do**
    $index \leftarrow index + 1$
  $value \leftarrow value + f_l / n$
  return p_{index}

Algorithm 32 *Tournament Selection*

$P \leftarrow$ population
$t \leftarrow$ tournament size, $t \geq 1$

$Best \leftarrow$ individual picked at random from P with replacement
for i from 2 to t **do**
    $Next \leftarrow$ individual picked at random from P with replacement
    if Fitness($Next$) $>$ Fitness($Best$) **then**
        $Best \leftarrow Next$
return $Best$

Algorithm 33 *The Genetic Algorithm with Elitism*

$popsize \leftarrow$ desired population size
$n \leftarrow$ desired number of elite individuals ▷ $popsize - n$ should be even

$P \leftarrow \{\}$
for $popsize$ times **do**
    $P \leftarrow P \cup \{$new random individual$\}$
$Best \leftarrow \square$
repeat
    for each individual $P_i \in P$ **do**
        AssessFitness(P_i)
        if $Best = \square$ or Fitness(P_i) $>$ Fitness($Best$) **then**
            $Best \leftarrow P_i$
    $Q \leftarrow \{$the n fittest individuals in P, breaking ties at random$\}$
    for $(popsize - n)/2$ times **do**
        Parent $P_a \leftarrow$ SelectWithReplacement(P)
        Parent $P_b \leftarrow$ SelectWithReplacement(P)
        Children $C_a, C_b \leftarrow$ Crossover(Copy(P_a), Copy(P_b))
        $Q \leftarrow Q \cup \{$Mutate(C_a), Mutate(C_b)$\}$
    $P \leftarrow Q$
until $Best$ is the ideal solution or we have run out of time
return $Best$

Algorithm 34 *The Steady-State Genetic Algorithm*

```
popsize ← desired population size

P ← {}
for popsize times do
    P ← P ∪ {new random individual}
Best ← □
for each individual P_i ∈ P do
    AssessFitness(P_i)
    if Best = □ or Fitness(P_i) > Fitness(Best) then
        Best ← P_i
repeat
    Parent P_a ← SelectWithReplacement(P)          ▷ We first breed two children C_a and C_b
    Parent P_b ← SelectWithReplacement(P)
    Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
    C_a ← Mutate(C_a)
    C_b ← Mutate(C_b)
    AssessFitness(C_a)                             ▷ We next assess the fitness of C_a and C_b
    if Fitness(C_a) > Fitness(Best) then
        Best ← C_a
    AssessFitness(C_b)
    if Fitness(C_b) > Fitness(Best) then
        Best ← C_b
    Individual P_d ← SelectForDeath(P)
    Individual P_e ← SelectForDeath(P)             ▷ P_d must be ≠ P_e
    P ← P − {P_d, P_e}                             ▷ We then delete P_d and P_e from the population
    P ← P ∪ {C_a, C_b}                             ▷ Finally we add C_a and C_b to the population
until Best is the ideal solution or we have run out of time
return Best
```

Algorithm 35 *The Genetic Algorithm (Tree-Style Genetic Programming Pipeline)*

$popsize \leftarrow$ desired population size
$r \leftarrow$ probability of performing direct reproduction ▷ Usually $r = 0.1$

$P \leftarrow \{\}$
for $popsize$ times **do**
    $P \leftarrow P \cup \{$new random individual$\}$
$Best \leftarrow \square$
repeat
    for each individual $P_i \in P$ **do**
        AssessFitness(P_i)
        if $Best = \square$ or Fitness(P_i) $>$ Fitness($Best$) **then**
            $Best \leftarrow P_i$
    $Q \leftarrow \{\}$
    repeat ▷ Here's where we begin to deviate from The Genetic Algorithm
        if $r \geq$ a random number chosen uniformly from 0.0 to 1.0 inclusive **then**
            Parent $P_i \leftarrow$ SelectWithReplacement(P)
            $Q \leftarrow Q \cup \{$Copy(P_i)$\}$
        else
            Parent $P_a \leftarrow$ SelectWithReplacement(P)
            Parent $P_b \leftarrow$ SelectWithReplacement(P)
            Children $C_a, C_b \leftarrow$ Crossover(Copy(P_a), Copy(P_b))
            $Q \leftarrow Q \cup \{C_a\}$
            if $||Q|| < popsize$ **then**
                $Q \leftarrow Q \cup \{C_b\}$
    until $||Q|| = popsize$ ▷ End Deviation
    $P \leftarrow Q$
until $Best$ is the ideal solution or we have run out of time
return $Best$

Algorithm 38 *Differential Evolution (DE)*

$\alpha \leftarrow$ mutation rate ▷ Commonly between 0.5 and 1.0, higher is more explorative
popsize $\leftarrow$ desired population size

$P \leftarrow \langle \rangle$ ▷ Empty population (it's convenient here to treat it as a vector), of length *popsize*
$Q \leftarrow \square$ ▷ The parents. Each parent Q_i was responsible for creating the child P_i
for i from 1 to *popsize* **do**
  $P_i \leftarrow$ new random individual
$Best \leftarrow \square$
repeat
  for each individual $P_i \in P$ **do**
    AssessFitness(P_i)
    if $Q \neq \square$ and Fitness(Q_i) $>$ Fitness(P_i) **then**
      $P_i \leftarrow Q_i$ ▷ Retain the parent, throw away the kid
    if $Best = \square$ or Fitness(P_i) $>$ Fitness($Best$) **then**
      $Best \leftarrow P_i$
  $Q \leftarrow P$
  for each individual $Q_i \in Q$ **do** ▷ We treat individuals as vectors below
    $\vec{a} \leftarrow$ a copy of an individual other than Q_i, chosen at random with replacement from Q
    $\vec{b} \leftarrow$ a copy of an individual other than Q_i or $\vec{a}$, chosen at random with replacement from Q
    $\vec{c} \leftarrow$ a copy of an individual other than Q_i, $\vec{a}$, or $\vec{b}$, chosen at random with replacement from Q
    $\vec{d} \leftarrow \vec{a} + \alpha(\vec{b} - \vec{c})$ ▷ Mutation is just vector arithmetic
    $P_i \leftarrow$ one child from Crossover($\vec{d}$, Copy(Q_i))
until $Best$ is the ideal solution or we ran out of time
return $Best$

- **The size of the population** A very large population approaches random search. A very small population approaches hill-climbing.

- **How likely a fit parent is chosen over an unfit parent (*Selection Pressure*)** High selection pressure approaches hill-climbing. Low selection pressure approaches a *random walk* (note: not random search).

- **How many children are generated from a parent** Many children samples a lot near parents, similar to steepest-ascent hill-climbing. Few children is similar to plain hill-climbing.

- **How different children are from their parents (*Mutation Rate*)** A high mutation rate fuzzes out the samples more, approaching random search. Very small mutation rates finesse local optima more precisely.

- **Whether parents can stick around (*Elitism or Survival Selection*)** If parents cannot stick around, the algorithm is more like a random walk. Else it tends to exploit the parents' optima. In cases like DE, a child must defeat its parent to even be included in the population.

Algorithm 39 *Particle Swarm Optimization (PSO)*

$swarmsize \leftarrow$ desired swarm size
$\alpha \leftarrow$ proportion of velocity to be retained
$\beta \leftarrow$ proportion of personal best to be retained
$\gamma \leftarrow$ proportion of the informants' best to be retained
$\delta \leftarrow$ proportion of global best to be retained
$\epsilon \leftarrow$ jump size of a particle

$P \leftarrow \{\}$
for $swarmsize$ times **do**
    $P \leftarrow P \cup \{$new random particle $\vec{x}$ with a random initial velocity $\vec{v}\}$
$\overrightarrow{Best} \leftarrow \square$
repeat
    for each particle $\vec{x} \in P$ with velocity $\vec{v}$ **do**
        AssessFitness($\vec{x}$)
        if $\overrightarrow{Best} = \square$ or Fitness($\vec{x}$) $>$ Fitness($\overrightarrow{Best}$) **then**
            $\overrightarrow{Best} \leftarrow \vec{x}$
    for each particle $\vec{x} \in P$ with velocity $\vec{v}$ **do** ▷ Determine how to Mutate
        $\vec{x}^* \leftarrow$ previous fittest location of $\vec{x}$
        $\vec{x}^+ \leftarrow$ previous fittest location of informants of $\vec{x}$ ▷ (including $\vec{x}$ itself)
        $\vec{x}^! \leftarrow$ previous fittest location any particle
        for each dimension i **do**
            $b \leftarrow$ random number from 0.0 to β inclusive
            $c \leftarrow$ random number from 0.0 to γ inclusive
            $d \leftarrow$ random number from 0.0 to δ inclusive
            $v_i \leftarrow \alpha v_i + b(x_i^* - x_i) + c(x_i^+ - x_i) + d(x_i^! - x_i)$
    for each particle $\vec{x} \in P$ with velocity $\vec{v}$ **do** ▷ Mutate
        $\vec{x} \leftarrow \vec{x} + \epsilon\vec{v}$
until $\overrightarrow{Best}$ is the ideal solution or we have run out of time
return $\overrightarrow{Best}$

This implementation of the algorithm relies on five parameters:

- α: how much of the original velocity is retained.
- β: how much of the personal best is mixed in. If β is large, particles tend to move more towards their own personal bests rather than towards global bests. This breaks the swarm into a lot of separate hill-climbers rather than a joint searcher.
- γ: how much of the informants' best is mixed in. The effect here may be a mid-ground between β and δ. The *number* of informants is also a factor (assuming they're picked at random): more informants is more like the global best and less like the particle's local best.
- δ: how much of the global best is mixed in. If δ is large, particles tend to move more towards the best known region. This converts the algorithm into one large hill-climber rather than separate hill-climbers. Perhaps because this threatens to make the system highly exploitative, δ is often set to 0 in modern implementations.
- ϵ: how fast the particle moves. If ϵ is large, the particles make big jumps towards the better areas — and can jump over them by accident. Thus a big ϵ allows the system to move quickly to best-known regions, but makes it hard to do fine-grained optimization. Just like in hill-climbing. Most commonly, ϵ is set to 1.