Evolutionary Computation

2023/24

Master Artificial Intelligence

Arno Formella

Departamento de Informática
Escola Superior de Enxeñaría Informática
Universidade de Vigo

23/24

GA: selection

- In the principal loop, there are two selection processes:
  - How to select the parents to generate the off-springs? and
  - How to rearrange the final population or next generation?
- We use the following notation:
  - μ stands for the number of individuals in the population
  - λ stands for the number of children being generated
- We distinguish two main strategies:
  - $(\mu + \lambda)$-strategy
    - from the μ individuals of the current generation select the parents and generate λ children
    - from the $\mu + \lambda$ individuals choose the μ best ones as new generation
  - (μ, λ)-strategy
    - from the μ individuals of the current generation select the parents and generate $\lambda \geq \mu$ children
    - from the λ children choose the μ best ones as new generation

The second question from above is answered.

GA: selection options

To select the parents being allowed to have off-springs,
there exists a bunch of suggestions:

roulette wheel: assign to each individual a fraction of the wheel according to its relative fitness and spin the wheel (variation: smooth, weight, or normalize the fitness somehow, e.g., use $\log$ of objective function, or use z-score)

rank based: order the individuals according to fitness and select with a probability weighted by the rank
(variation: compute selection probability with linear function of rank, so the least ranked still gets certain probability to get selected)

GA: selection options (continued)

tournament based: draw a certain number of random individuals, select the best one as parent (variation: select directly the best two as parents)

truncation selection: only the individuals with highest fitness values will be parents

what-ever-you like: remember, do something, be happy...

The first question (from two slides earlier) is answered.

GA: initialization

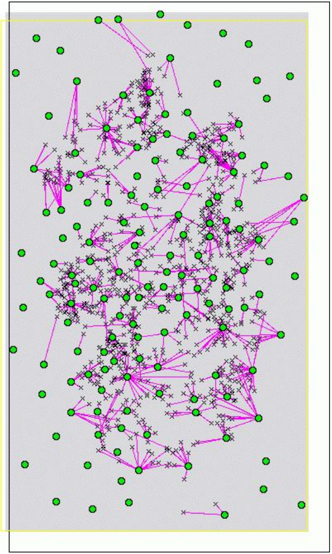
- generate the initial population with random genomes
  - take into account that a distribution in genotype not necessarily is similar to the same distribution in phenotype
  - there are maybe many individuals with very low fitness
  - the initial convergence rate might be slow
- generate the initial population with individuals from another heuristic algorithm or various such algorithms
  - the population might be biased into a certain region of the search space
  - the diversity of the population might be low
  - the convergence rate might be trapped early in a local optimum
- recommendation: use a mixture of both

GA: termination criteria

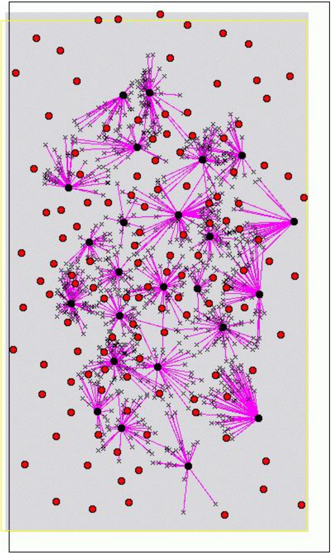
There are many possibilities when to stop the iteration of a genetic algorithm:

- once the first solution has been found
- once a sufficiently good solution has been found
- once the optimum has been found
- once a certain number of iterations has been executed
- once the diversity of the population is below a certain threshold
- once the convergence rate of the improvement is below a certain threshold
- once a certain amount of runtime has been spent
- recommendation: use an or-mixture of all

GA: results for the example (2007)



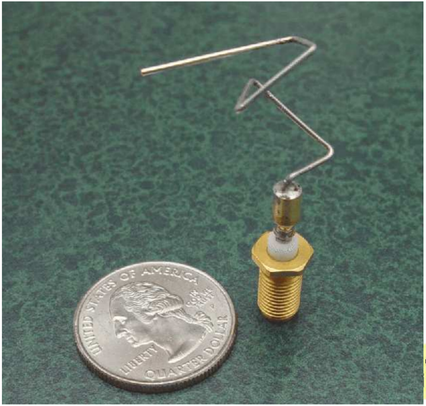
119 of 149 nodes used

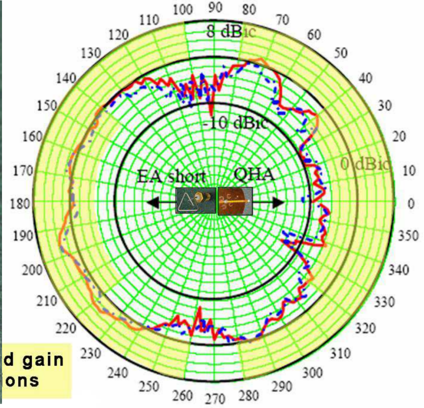


24 of 149 nodes used

GA: use for antenna design (small satellites)

Horny, AlGlobus, Linden, Lohn: Automated Antenna Design with Evolutionary Algorithms





GA: diversity

- The diversity measures, in some sense, the non-similarity between the individuals of a population.
- E.g., Hamming-distance over the bitstring (using exor): $010010 \otimes 101000 = 111010 \longrightarrow 4$
- e.g., delta-distance over the integer (or real) sequence: $\sum_i |x_i - x'_i|$ (being x and x' two individuals)
- There are much more similarity measures.
- similar individuals in a population reduce the diversity and the genetic algorithm maybe gets stuck in some region of the search space (maybe, but not necessarily, a local minimum).
- To augment the diversity, we have only the mutation operation, provided the mutation becomes visible in the next generation. (Observe: whenever an allele disappears in a population, most of the crossover operations cannot regenerate it!)
- Another possibility is just to regenerate a completely or partially new population.

GA: elitism

We have to draw the decision whether the best individual(s) is (are) forced to belong unmodified to the next generation.

- Elitism might help to converge faster.
- Elitism might reduce diversity faster.
- The consequences of this trade-off are problem dependent.

GA: final remarks

- The difficulties of understanding and analizing genetic algorithms lie in the fact that they implement a combination of random search (by mutation) and biased search (by recombination).
- Genetic algorithms need unique and problem-specific mutation and recombination operators, which makes it more challenging to implement a generic version that can be easily applied to different optimization problems.
- Nature still has its somewhat better approach: DNA, RNA, gene expression, proteins, and mitochondria (mtDNA)...

Once we have seen genetic algorithms, evolutionary programming is somewhat simpler: it just uses mutation.

- there exist only the phenotypes, let's say x_i (for $i = 1, \ldots, n$), i.e., n individuals in the population
- modification (mutation) is realized over the phenotypes as:

$$x_i' = x_i + r_i \sqrt{\beta f(x_i) + \gamma}$$

  being $\beta > 0$ and $\gamma \geq 0$ tuning parameters (for instance $\beta = 1$ and $\gamma = 0$) and r_i is a random value taken from a normal distribution with mean 0 and variance 1 (i.e., $r_i \in N[0,1]^n$).
- Note that the fitness (objective function f) must be shifted, so the minimum is positive.
- Usually a $(\mu + \mu)$-selection strategy is used: all individuals are mutated and the best μ individuals are kept.

EP: principal loop

A evolutionary programming algorithm can be summarized in the following principal loop:

```
InitializePopulation()
EvaluateIndividuals()
while not Stopping():
  GenerateChildrenByMutation()
  EvaluateIndividuals()
  ReestablishPopulation()
```

differential evolution (DE)

Once we have seen genetic algorithms, differential evolution is somewhat simpler: it just uses a special type of recombination.

- there exist only the phenotypes, let's say x_i (for $i = 1, \ldots, n$), i.e., n individuals in the population
- For each individual we select three other individuals, say x_j, x_k, x_l, to compute a mutant vector v_i

$$v_i = x_j + F \cdot (x_k - x_l)$$

  being $F \in [0.4, 0.9]$ (usually) a tuning parameter.
- Then we generate an off-spring with a uniform crossover between individual x_i and mutant v_i using a certain threshold c
- Usually a $(\mu + \mu)$-like selection strategy is used: all individuals are used to generate off-springs, and the best are kept.

DE: principal loop

A differential evolution algorithm can be summarized in the following principal loop:

```
InitializePopulation()
EvaluateIndividuals()
while not Stopping():
  GenerateChildrenByDiffusion()
  EvaluateIndividuals()
  ReestablishPopulation()
```

DE: some variations

- One might consider to use always the best individual found so far as individual x_j.
- The tuning parameter F might vary, i.e., taking the value from a uniform or a normal distribution.
- One might use DE on discrete sets as well by just rounding the mutants appropriately (or search in the close integer neighborhood according to the dimension of the underlying problem).
- (My opinion) Differential evolution is not just a genetic algorithm, as there is no genotype, rather the other way round: a genetic algorithm using the phenotype as genotype, no mutation, and a random recombination, becomes a differential evolution algorithm.

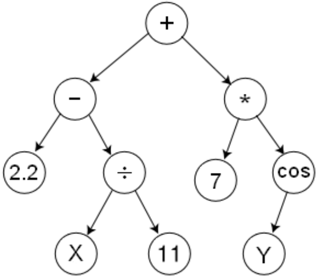
genetic programming (GP)

Once we have seen genetic algorithms, genetic programming is a genetic algorithm with some special phenotypes and genotypes.

- the genotype is a (simple) program described as a syntax tree that can be written as well with Polish notation (prefix notation), see next slide...
- the parenthesis can be eliminated, interpretation of the corresponding expression is easy to perform with a stack automaton.
- some properties of the execution of the resulting program (as phenotype) are used as fitness (see example, later)

GP: syntax tree

syntax tree and Polish notation



$$\left(2.2 - \left(\frac{X}{11}\right)\right) + \left(7 * \cos(Y)\right)$$

- `(2.2-(x/11))+(7*cos(y))`
- `(+ (- (2.2 (/ X 11))) (* (7 cos(Y))))`
- `+ - 2.2 / X 11 * 7 cos Y`

image taken from wikipedia

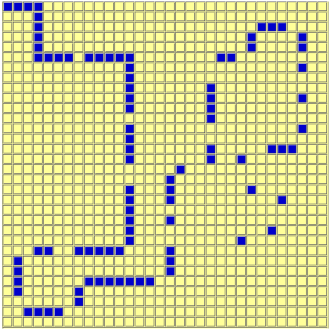
GP: mutation and crossover operations

- the programs are modified with adecuate mutation and crossover operations
- mutation:
  - change a node, but take care to keep a valid syntax tree (maybe subtrees must be removed or added)
  - rotate nodes
  - interchange nodes
- crossover: interchange a subtree of one parent with a subtree of the other parent

GP: example

Program a robot (ant) that starts at some cell (usually a corner) and tries to find as many objects (food) with as few steps as possible.

Santa Fe Trail



nodes: turn-left, turn-right, move, if-food-ahead

- The inspiration comes from social behavior of individuals within an environment including other individuals.
- We work with n individuals that move in a continuous d-dimensional search space.
- The individuals move (in steps) through the search space and adjust their velocities according to information gathered from others (and their own *histories*).
- The individuals are grouped into neighborhoods.

PSO: velocity actualization

- x_i vector of current positions
- v_i vector of current directional velocities
- b_i best local position vector
- h_i best neighbor position vector
- $\varphi_1 = 2.05, \varphi_2 = 2.05$ influence values (just some *magic*)
- $\xi \in [0.4, 1]$, e.g. $\xi = 0.729$ inertia reduction value
- velocity actualization

$$
\begin{aligned}
v_i &= \xi\, v_i + U[0, \varphi_1] \circ (b_i - x_i) + U[0, \varphi_2] \circ (h_i - x_i) \\
x_i &= x_i + v_i
\end{aligned}
$$

- The $\circ$ operator is either a Hadamard-operation (i.e., component-wise), or a linear operation (i.e., scalar multiplication)

A particle swarm optimization can be summarized in the following principal loop:

```
InitializePopulation() # i.e. x_i, v_i
EvaluateIndividuals()  # i.e. b_i
DefineNeighborhoodSize()
while not Stopping():
  DetermineNeighborhoodValues()  # h_i
  UpdateIndividuals()            # i.e., x_i, v_i, b_i
```

PSO: some more details

- The velocity can be confined not to pass a certain maximum velocity, which helps to avoid explosion, i.e., that the area of the search space being explored becomes exponentially larger.
- Initial velocities can be zero or some random values.
- Small neighborhoods tend to provide a better global search, while large neighborhoods tend to produce a faster convergence (but maybe premature).
- Neighborhoods can be defined as nearest neighbors, as fixed and overlapping, or entail the entire population, or what-ever-you-like.
- The inertia reduction can be increased with the simulation time.
- The best global individual g can be included in the equation: add $+U[0,\varphi_3] \circ (g - x_i)$
- The worst (local and global) positions can be *avoided*: add $-U[0,\varphi_4] \circ (\overline{b}_i - x_i)$ and/or $-U[0,\varphi_5] \circ (\overline{h}_i - x_i)$ and/or $-U[0,\varphi_6] \circ (\overline{g} - x_i)$