Evolution Programs

We use a common term, **Evolution Programs (EP)** [1], for all evolution based systems, which maintain a population of potential solutions, have some selection process based on fitness of individuals, and some "genetic" operators. One type of such system is a class of Evolution Strategies i.e., algorithms that imitate the principles of natural evolution for parameter optimization problems. Evolutionary Programming is a technique for searching through a space of small finite-state machines. Glover's Scatter Search techniques maintain a population of reference points and generate offspring by weighted linear combinations. Another type of evolution based system is Holland's Genetic Algorithms (GAs). In 1990, Koza proposed an evolution-based system, Genetic Programming, to search for the fittest computer program to solve a particular problem. The structure of an evolution program is shown in Figure 1.

```
procedure evolution program
begin
   $t \leftarrow 0$
   initialize $P(t)$
   evaluate $P(t)$
   while (not termination-condition) do
   begin
      $t \leftarrow t + 1$
      select $P(t)$ from $P(t-1)$
      alter $P(t)$
      evaluate $P(t)$
   end
end
```

Fig. 1: The structure of an evolution program

The evolution program is a probabilistic algorithm that maintains a population of individuals, $P(t) = \{x_1^t, \cdots, x_n^t\}$ for iteration *t*. Each individual represents a potential solution to the problem at hand, and, in any evolution program, is implemented as some (possibly complex) data structure *S*. Each solution x_i^t is evaluated to give some measure of its "fitness". Then, a new population (iteration *t* + 1) is formed by selecting the more fit individuals (select step). Some members of the new population undergo transformations (alter step) using "genetic" operators to form new solutions. There are unary transformations m_i (mutation type), which create new individuals by a small change in a single individual ($m_i : S \rightarrow S$), and higher-order transformations c_j (crossover type), which create new individuals by combining parts from several (two or more) individuals ($c_j : S \times \ldots \times S \rightarrow S$). After some number of generations the program converges - it is hoped that the best individual represents a near-optimum (reasonable) solution.

There are a lot of crucial practical optimization problems for which such algorithms of high quality have become available. For instance, we can apply ***simulated annealing*** for wire routing and component placement problems in VLSI design or for the traveling salesman problem.

Simulated annealing is an optimization method coming from the study of physics [2]. Annealing is the process of heating an alloy and then cooling it down slowly. As the atoms are first made to jump around a lot and then gradually settle into a low energy state, the atoms can find a low energy conformation and this idea is applied in searching, for better solutions.

Moreover, many other large scale combinatorial optimization problems (many of which have been proved NP-hard) can be solved approximately on present-day computers by this kind of Monte Carlo technique.

In general, any abstract task to be accomplished can be thought of as solving a problem, which, in turn, can be perceived as a search through a space of potential solutions. Since we are after "the best" solution, we can view this task as an optimization process. For small spaces, classical ***exhaustive methods*** usually suffice; for larger spaces special artificial intelligence techniques must be employed. ***Genetic Algorithms*** (GAs) are among such techniques; they are stochastic algorithms whose search methods model some natural phenomena.

The idea behind genetic algorithms is to do what nature does. Let us take rabbits as an example: at any given time there is a population of rabbits. Some of them are faster and smarter than other rabbits. These faster, smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits. Of course, some of the slower, dumber rabbits will survive just because they are lucky. This surviving population of rabbits starts breeding. The breeding results in a good mixture of rabbit genetic material: some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits, and so on. And on the top of that, nature throws in a 'wild hare' every once in a while by mutating some of the rabbit genetic material. The resulting baby rabbits will (on average) be faster and smarter than these in the original population because relatively faster, smarter parents survived the foxes. (It is a good thing that the foxes are undergoing similar process - otherwise, the rabbits might become too fast and smart for the foxes to catch any of them). A genetic algorithm follows a step-by-step procedure that closely matches the story of the rabbits.

GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, transportation problems, traveling salesman problems, optimal control problems, database query optimization, etc. GAs belong to the class of probabilistic algorithms, yet they are very different from random algorithms as they combine elements of directed and stochastic search. Because of this, GAs are also more robust than existing directed search methods. Another important property of such genetic-based search methods is that they maintain a population of potential solutions - all other methods process a single point of the search space.

Hillclimbing methods use the iterative improvement technique; the technique is applied to a single point (the current point) in the search space. During a single iteration, a new point is selected from the neighborhood of the current point (this is why this technique is known also as neighborhood search or local search). If the new point provides a better value of the objective function, the new point becomes the current point. Otherwise, some other neighbor is selected and tested against the current point. The method terminates if no further improvement is possible.

It is clear that the hillclimbing methods provide local optimum values only and these values depend on the selection of the starting point. Moreover, there is no information available on the relative error (with respect to the global optimum) of the solution found.

To increase the chances to succeed, hillclimbing methods usually are executed for a (large) number of different starting points (these points need not be selected randomly - a selection of a starting point for a single execution may depend on the result of the previous runs).

The simulated annealing technique eliminates most disadvantages of the hillclimbing methods: solutions do not depend on the starting point any longer and are (usually) close to the optimum point. This is achieved by introducing a probability ***p*** of acceptance (i.e., replacement of the current point by a new point): $p = 1$, if the new point provides a better value of the objective function; however, $p > 0$, otherwise. In the latter case, the probability of acceptance p is a function of the values of objective function for the current point and the new point, and an additional control parameter, "temperature", T. In general, the lower the temperature T is, the smaller the chances for the acceptance of a new point are. During the execution of the algorithm, the temperature of the system, T, is lowered in steps. The algorithm terminates for some small value of T, for which virtually no changes are accepted anymore.

As mentioned earlier, a GA performs a multi-directional search by maintaining a population of potential solutions and encourages information formation and exchange between these directions. The population undergoes a simulated evolution: at each generation, the relatively "good" solutions reproduce, while the relatively "bad" solutions die. To distinguish between different solutions we use an objective (evaluation) function which plays the role of an environment. Examples of hillclimbing, simulated annealing, and genetic algorithm techniques are given next:

Assume, the search space is a set of binary strings ***v*** of the length 30. The objective function ***f*** to be maximized is given as:

$$f(v) = |11 \cdot \text{one}(v) - 150|,$$

where the function one(v) returns the number of 1s in the string v.

For example, the following three strings

$$v_1 = (110110101110101111111011011011),$$

$$v_2 = (111000100100110111001010100011),$$

$$v_3 = (000010000011001000000010001000),$$

would evaluate to

$$f(v_1) = |11 \cdot 22 - 150| = 92,$$

$$f(v_2) = |11 \cdot 15 - 150| = 15,$$

$$f(v_3) = |11 \cdot 6 - 150| = 84,$$

Since, one $(v_1) = 22$, one $(v_2) = 15$, and one $(v_3) = 6$.

The function *f* is linear and does not provide any challenge as an optimization task. We use it only to illustrate the ideas behind these three algorithms. However, the interesting characteristic of the function *f* is that it has one global maximum for

v_g = (111111111111111111111111111111),

$f(v_g)$ =|11· 30 - 150| = 180, and one local maximum for

v_l = (000000000000000000000000000000),

$f(v_l)$ =|11·0 - 150| = 150.

There are a few versions of hillclimbing algorithms. They differ in the way a new string is selected for comparison with the current string. One version of a simple (iterated) hillclimbing algorithm (MAX iterations) is given in Figure 2 (steepest ascent hillclimbing). Initially, all 30 neighbors are considered, and the one v_n which returns the largest value $f(v_n)$ is selected to compete with the current string v_c. If $f(v_c) < f(v_n)$, then the new string becomes the current string. Otherwise, no local improvement is possible: the algorithm has reached (local or global) optimum (*local* = TRUE). In such a case, the next iteration ($t \leftarrow t+1$) of the algorithm is executed with a new current string selected at random.

```
procedure iterated hillclimber
begin
  $t \leftarrow 0$
  repeat
    $local \leftarrow$ FALSE
    select a current string $v_c$ at random
    evaluate $v_c$
    repeat
      select 30 new strings in the neighborhood of $v_c$
        by flipping single bits of $v_c$
      select the string $v_n$ from the set of new strings
        with the largest value of objective function $f$
      if $f(v_c) < f(v_n)$
        then $v_c \leftarrow v_n$
        else $local \leftarrow$ TRUE
    until $local$
    $t \leftarrow t + 1$
  until $t = MAX$
end
```

Fig. 2: A simple (iterated) hillclimber.

It is interesting to note that the success or failure of the single iteration of the above hillclimber algorithm (i.e., the return of the global or local optimum) is determined by the starting string (randomly selected). It is clear that if the starting string has thirteen ls or less, the algorithm will always terminate in the local optimum (failure). The reason is that a string with thirteen ls returns a value 7 of the objective function, and any single-step improvement towards the global optimum, i.e., increase the number of ls to fourteen, decreases the value of the objective function to 4. On the other hand, any decrease in the number of ls would increase the value of the function: a string with twelve ls yields a value of 18, a string with eleven 1s yields a value of 29, etc. This would push the search in the “wrong” direction, towards the local maximum.

For problems with many local optima, the chances of hitting the global optimum (in a single iteration) are slim.

The structure of the simulated annealing procedure is given in Figure 3. The function random [0, 1)[1] returns a random number from the range [0, 1). The (termination-condition) checks whether ‘thermal equilibrium’ is reached, i.e., whether the probability distribution of the selected new strings approaches the Boltzmann distribution. However, in some implementations, this repeat loop is executed just k times (k is an additional parameter of the method). The temperature T is lowered in steps ($g(T, t) < T$ for all t). The algorithm terminates for some small value of T: the (stop-criterion) checks whether the system is ‘frozen’, i.e., virtually no changes are accepted anymore.

As mentioned earlier, the simulated annealing algorithm can escape local optima. Let us consider a string

v_4 = (111000000100110111001010100000),

with twelve 1s, which evaluates to $f(v_4) = |11 \cdot 12 - 150| = 18$. For v_4 as the starting string, the hillclimbing algorithm (as discussed earlier) would approach the local maximum

v_l = (000000000000000000000000000000),

```
procedure simulated annealing
begin
   $t \leftarrow 0$
   initialize temperature $T$
   select a current string $v_c$ at random
   evaluate $v_c$
   repeat
      repeat
         select a new string $v_n$
            in the neighborhood of $v_c$
            by flipping a single bit of $v_c$
         if $f(v_c) < f(v_n)$
            then $v_c \leftarrow v_n$
            else if $random[0,1) < \exp\{(f(v_n) - f(v_c))/T\}$
                then $v_c \leftarrow v_n$
      until (termination-condition)
      $T \leftarrow g(T,t)$
      $t \leftarrow t+1$
   until (stop-criterion)
end
```

Fig. 3: Simulated Annealing.

[1] If the variable x is said to be within the range of [0, 1), it implies $0 \le x < 1$.

since any string with thirteen 1s (i.e., a step ‘towards’ the global optimum) evaluates to 7 (less than 18).

On the other hand, the simulated annealing algorithm would accept a string with thirteen 1s as a new current string with probability

$$p = \exp\{(f(v_n) - f(v_c))/T\} = \exp\{(7 - 18)/T\}$$

which, for some temperature, say, T = 20, gives

$$p = e^{-\frac{11}{20}} = 0.57695,$$

i.e., the chances for acceptance are better than 50%.

On the other hand, Genetic algorithms (see Figure 4), can maintain a population of strings. For example, two relatively poor strings:

v_5 = (111110000000110111001110100000) and
v_6 = (000000000000110111001010111111)

each of which evaluates to 4, can produce much better offspring (if the crossover point falls anywhere between the 5th and the 12th position):

v_7 = (111110000000110111001010111111).

The new offspring v_7 evaluates to $f(v_7) = |11 \cdot 19 - 150| = 59$.

```
1. Form the initial population (usually random)
2. Compute the fitness to evaluate each chromosomes (member of them population)
3. Select pairs to mate from best-ranked individuals and replenish population
      a. - Apply crossover operator
      b. - Apply mutation operator
4. Check for termination criteria, else go to step #2
```

Fig. 4: Pseudocode for Genetic Algorithm

Conclusions:
Here we have discussed the basic versions of hillclimbing, simulated annealing and genetic algorithm to keep it simple for developing the intuitions about a non-deterministic search algorithm in general – however, it should be noted that updated and improved versions are available.

We have several other algorithms with similar idea, such as: *Particle Swarm Optimization* (PSO) [3], *Artificial Colony Optimization* (ACO) [4], *Differential Evolution* (DE) [5], *Artificial Bee Colony* (ABC) [6], *Glowworm Swarm Optimization* (GSO) [7], *Cuckoo Search Algorithm* (CSA) [8], *Firefly Algorithm* (FA) [9], *Bat Algorithm* (BA) [10], *Monkey Algorithm* (MA) [11], *Krill Herd Algorithm* (KHA) [12], *Wind Driven Optimization* (WDO) [13], *Social Spider Algorithm* (SSA) [14], *Artificial Immune Systems* [15], *Conformational Space Annealing* [16], and so on.

Next, we discuss, how do the GAs work (separate study material is provided).

References:

1. Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution*. Programs New York: Springer-Verlag. 1992.
2. Segaran, T., *Programming Collective Intelligence*. 2007: O'REILLY.
3. Kennedy, J. and R.C. Eberhart, *Particle swarm optimization.* Proceedings of IEEE International Conference on Neural Networks, 1995: p. 1942 - 1948.
4. Dorigo, M., V. Maniezzo, and A. Colorni, *Ant system: optimization by a colony of cooperating agents.* IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics, 1996. **26**(1): p. 29-41.
5. Storn, R. and K. Price, *Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces.* Journal of Global Optimization, 1997. **11**(4): p. 341 - 359.
6. Karaboga, D., *An idea based on honeybee swarm for numerica optimization.* Technical Report TR06, Erciyes University, 2005.
7. Krishnanand, K. and D. Ghose. *Detection of multiple source locations using a glowworm metaphor with applications to collective robotics*. in *Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*. 2005. IEEE.
8. Yang, X.-S. and S. Deb, *Cuckoo Search via Lévy flights.* World Congress on Nature & Biologically Inspired Computing (NaBIC), 2009: p. 210 - 214
9. Fateen, S.-E.K. and A. Bonilla-Petriciolet, *Intelligent firefly algorithm for global optimization*, in *Cuckoo Search and Firefly Algorithm*. 2014, Springer. p. 315-330.
10. Yang, X.-S. and A. Hossein Gandomi, *Bat algorithm: a novel approach for global engineering optimization.* Engineering Computations, 2012. **29**(5): p. 464-483.
11. Zhao, R. and W. Tang, *Monkey algorithm for global numerical optimization.* J. Uncertain Syst., 2008. **2**: p. 165 - 176.
12. Gandomi, A.H. and A.H. Alavi, *Krill herd: A new bio-inspired optimization algorithm.* Nonlinear Sci. Numer. Simul., 2012. **17**: p. 4831 - 4845.
13. Bayraktar, Z., M. Komurcu, and D.H. Werner. *Wind Driven Optimization (WDO): A novel nature-inspired optimization algorithm and its application to electromagnetics*. in *Antennas and Propagation Society International Symposium (APSURSI), 2010 IEEE*. 2010. IEEE.
14. Yu, J.J.Q. and V.O.K. Li, *A Social Spider Algorithm for Global Optimization.* Neural and Evolutionary Computing, 2015. **30**(C): p. 614 - 627.
15. Cutello, V., et al., *An Immune Algorithm for Protein Structure Prediction on Lattice Models.* IEEE Transactions on Evolutionary Computation, 2007. **11**(1).
16. Lee, J., *Conformational space annealing and a lattice model Protein.* Journal of the Korean Physical Society, 2004. **45** (6): p. 1450-1454.

--- x ---