Intelligent Systems Engineering

EC2 Evolutionary Optimization

Numerical and Combinatorial Optimization

Petr Musilek

University of Alberta

Outline

These notes are based on [Keller et al. 2017] chapter 11.

Evolutionary Optimization

Simulated evolution has been used as a search mechanism to find optimal solutions to problems

- design computer programs that would control a robot [Friedman,1956]
- crafting of simple computer programs [Friedberg et al., 1958]
- design of physical devices [Rechenberg, 1965]
- development of predictive models of time series data [Fogel et al., 1966]
- generating strategies in games [Reed et al., 1967; Fogel and Burgin, 1969]

There is a rich and interesting history to the field of evolutionary computation, as described in Evolutionary Computation: The Fossil Record [Fogel,1998]; in virtually all of these cases

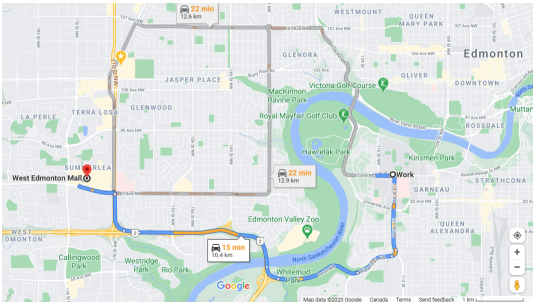
- evolution was used as a foundation for searching a space of possible solutions to a problem using methods of random variation and selection

The problem to be solved must be well defined; this especially means

- any possible solution to the problem must be comparable to another possible solution
- most often, this based on quantitative measures of how well a proposed solution meets the needs of the problem

Examples

- finding a minimum path between 2 points
  - any proposed path that avoids the obstacles and gets from the first point to the other point has a certain distance
  - that distance provides a quantitative measure of the quality of the path (the shorter the better)
- designing a schedule for a factory that maximizes profitability
- finding a new potential drug that binds to a specific protein with min free energy



In some cases other than purely quantitative measures of a solution's "fitness" can be used (due to flexibility of evolutionary algorithms)

Example: find a blend of coffee that you find most enjoyable from a mixture of five different types of coffee beans. Try random combinations of blends and seek to evolve the overall blend that you like best; you could

- rate each blend on a numeric scale, say from 0 to 10 with 10 being the best coffee you've ever tasted to make a quantitative comparison between blends,
  (there is a risk that the best coffee that you've ever tasted might be surpassed by a new blend that is the new best coffee that you've ever tasted, and by our scoring system, each would receive a score of 10, unless you retaste and rescore coffees at each generation)
- simply rank the blends in order of preference, or
- simply assert that you like one blend more than another (in the most primitive case)
  (see Herdy [1997] for an example of evolving coffee blends)

Qualitative (or even fuzzy) fitness measures are most often found in interactive evolutionary computation, where a human provides a judgment about the quality of proposed solutions.

There are essentially two forms of optimization problem

Numeric e.g., find the point (x, y) such that $f(x, y) = x^2 + y^2$ is minimized: the solution space is $\mathbb{R}^2$ and $f(x, y)$ can be used as a measure of solution quality (lower is better because it's a minimization problem).

Combinatoric e.g., given a collection of tools, each with a certain weight, find the combination that has the greatest combined weight that will fit in a bag that can hold only 25% of the weight of all the tools combined (knapsack problem).

Here, we are not searching for a point in $\mathbb{R}^n$, but rather for a combination of items that can be listed (here, the order of the listing doesn't matter; in other problems, the order of presentation of the items makes a big difference - e.g. optimizing the arrival of supplies at a construction site).

A Canonical Example in One Dimension

Consider the simplest example: searching for a point $x \in \mathbb{R}$ such that $f(x) = x^2$ is minimized

- since the function $f(x)$ is quadratic, finding the minimum using calculus methods is straight forward
- suppose we did not know that the function $f(x)$ was actually $f(x) = x^2$, but
- a "black box" that responded with a number any time we put a number in the box
- we need to find a number that minimizes what the box generates
  (this is called black box optimization)

An evolutionary approach to finding the minimum number could be:

1. We start by forming a population of candidate solutions, $x_1, \ldots, x_\mu$, where there are μ parents selected at random from a portion of real numbers (say uniformly between a lower limit of -100 and an upper limit of +100).

$i = 0;$
repeat
$\quad i = i + 1$
$\quad x(i) = U(-100, 100);$
until $(i == \mu)$

where $U(-100, 100)$ is a uniformly distributed random variable over $[-100, 100]$.

2. Each of these parents is varied randomly to create more solutions, $x_{\mu+1}, \ldots, x_{\mu+\lambda}$, where there are λ offspring. For example, one typical method is to add a random number from a standard Gaussian distribution $N(0,1)$ to a parent to create an offspring.

   For the sake of simplifying notation, assume that in this case $\lambda = \mu$ and thus each parent creates one offspring (although there is no limitation in evolutionary algorithms about the number of offspring that can be created from a parent).

   $i = 0;$
   repeat
   $\quad i = i + 1$
   $\quad x[\mu + i] = x[i] + N(0,1);$
   until $(i == \mu)$

   where $N(0,1)$ denotes a standard Gaussian random variable (a.k.a. "standard normal")

3. At this point, we have 2μ random ideas about what to put in the black box. We now have to test each idea and see what the box says. In pseudocode, this would be as follows:

```
i = 0;
repeat
    i = i + 1
    score_x[i] = f(x[i])
until (i == 2μ)
```

4. We then rank order the 2μ solutions in terms of their scores from lowest to highest. The μ best-ranking solutions then become the new parents for the next generation.

```
InitializePopulation;
repeat
    CreateOffsrping;
    ScoreEveryone;
    SelectNewParents;
until (done)
```

The question of when to stop this procedure is often a matter of how much time is available to compute a solution or what level of quality is required, i.e.

- continue the process of variation and selection for some number of generations g, or
- until the value of the best solution is lower than a threshold, say, 10^{-6}.

This threshold would work in our problem if we had a hunch that zero was the minimum (which it is for $f(x) = x^2$).

If we compared different algorithms on this specific problem, we would observe that

- simple evolutionary algorithm quickly locates solutions that have a score of less than 10^{-6},
- but not as quickly as some other search methods (e.g. bisection or gradient search),
- and certainly not as quickly as calculus if we knew the function inside the black box.

A Canonical example in Two or More dimensions

The 1D example can be easily extended it to the canonical case of two (or more dimensions)

- parents are chosen from the space $\mathbb{R}^n$, where n is the number of dimensions
- offspring can be created by randomly varying each dimension of the parent,
- and also by combining or averaging across parents; then
- all the solutions are assessed and the best are retained to be parents of the next generation

With regard to creating offspring from parents

- methods that use a single parent to create a single offspring are called mutation, whereas
- methods that combine multiple parents to create offspring are called recombination. There are various forms of recombination that have their origins in inspiration from nature.

One method of **recombination** is called crossover; it operates on the following two solutions:

$$x_{11}, x_{12}, \ldots, x_{1n}$$
$$x_{21}, x_{22}, \ldots, x_{2n}$$

where x_{12} denotes the second parameter of the first solution and n is the number of parameters (dimensions).

A crossover point is selected, usually at random, and two new solutions are created by splicing the first part of the first solution with the second part of the second solution, and vice versa.

For example, suppose the crossover point was 3, then the two offspring would be

$$x_{11}, x_{12}, x_{23} \ldots, x_{2n}$$
$$x_{21}, x_{22}, x_{13} \ldots, x_{1n}$$

This is called "one-point" crossover.

- The one-point crossover operator has the sometimes undesirable property of forcing segments that are near each end of the solution vector to remain together.
- Thus, a multipoint (or n-point) crossover operator can be employed, which treats the solution vectors more like rings in which sections can be exchanged, rather than strings in which a transition is made from one to the other.
- The limiting form of this, called uniform crossover, selects one component from either parent at random without regard to maintaining continuous segments and exchanges them.
- Another form of recombination is blending. This averages parameters of parent solutions when creating offspring. For example, the two parents

$$x_{11}, x_{12}, x_{13} \ldots, x_{1n} \text{ and } x_{21}, x_{22}, x_{23} \ldots, x_{2n}$$

could create

$$(x_{11} + x_{21})/2,\ (x_{12} + x_{22})/2,\ (x_{13} + x_{23})/2, \ldots, (x_{1n} + x_{2n})/2$$

In general, there is no need to use a simple arithmetic mean; a weighted arithmetic mean or even a geometric mean may be useful in certain circumstances.

Both crossover and blending recombination can be extended to more than two parents. There are no restrictions that evolutionary operators follow the operations as found in nature.

It is almost always the case that evolutionary algorithms are employed when an optimization problem has multiple dimensions rather than just one dimension. In pseudocode, the process can be described as

```
InitializePopulation;
repeat
    CreateOffsrping; // mutate and/or recombine
    ScoreEveryone;
    SelectNewParents;
until (done)
```

Evolution versus Gradient Methods

If the problem presents a smooth, convex, continuous landscape (e.g., $f(x, y) = x^2 + y^2$)

- then gradient or related methods of optimization will be faster in locating the single optimum point

If the problem landscape has multiple local optima (e.g., $f(x) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)]$)

- then gradient methods will likely fail to find the global optimum solution because each gradient is associated only with the local optima created by the effects of the cosine function ($A = 20$)

This is an opportunity for "generate and test" methods evolutionary optimization to be used effectively: in cases where the landscape is discontinuous and/or not smooth and, thus, gradient-based approaches may be inapplicable as it may not be possible to compute gradients.

Combinatorial Optimization

Consider the canonical case of combinatorial optimization: traveling salesman problem (TSP)

- there are n cities
- the salesman starts at one of these cities and must visit each other city once and only once and then return home
- the salesman wants to do this in the shortest distance
- the problem then is to determine the best ordering of the cities to visit

This is a difficult problem because the total number of possible solutions increases as a factorial function of the number of cities n

$$(n-1)!/2$$

- for $n = 10$ (small problem), there are 181,440 different paths to choose from
- for $n = 100$, the number of different paths is on the order of 10^{150}; enumerating them all and choosing the best option is infeasable

Evolutionary approach to TSP

1. Create a data structure to represent a solution; one possibility is a list of cities to be visited in order, e.g., given seven cities, a potential solution is

$$[1, 3, 2, 5, 7, 6, 4]$$

in which the salesman starts at city 1 and return to city 1 after visiting city 4.
Other data structures are possible; e.g. in the form of a series of links (graph edges)

$$[(1, 3), (3, 2), (2, 5), (5, 7), (7, 6), (6, 4), (4, 1)]$$

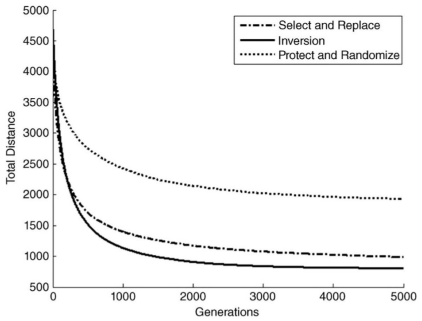
2. Score any potential solution. Here, the score associated with any solution, f, is the length of the total path through the cities (lower scores are better)

$$f = \sum_{i=1}^{n-1} d(c_i, c_{i+1}) + d(c_n, c_1)$$

where $d(a, b)$ is the distance between city a and city b, and c_i is the i-th city.

3. Then we must determine a method for creating offspring solutions from parent solutions. Given the representation of an ordered list, the following are some of the potential methods:

- *Select and replace:* Choose a city at random along the list and replace it at a random place along the list.
- *Invert:* Choose two cities along the list at random and invert the segment between those cities.
- *Protect and randomize:* Choose a segment of the list to be passed from the parent to the offspring intact, and then randomize the remaining cities in the list.



The results suggest that the two-point inversion operator generates better solutions faster than the other two mutation operators on this problem.

4. Consider a recombination operator. One possibility is partially mapped crossover (PMX). PMX works on two parents
   - it chooses a segment of the first parent to move directly to the offspring,
   - moves the feasible parts from the second parent to the offspring and, finally,
   - it assigns the remaining values to the offspring based on the indexing in the first parent.

Example: Suppose we have two parents

$$[3, 5, 1, 2, 7, 6, 8, 4] \text{ and } [1, 8, 5, 4, 3, 6, 2, 7]$$

and that the segment $[1, 2, 7]$ is selected to be saved from parent 1. At this step the offspring could be written as

$$[+, +, 1, 2, 7, +, +, +]$$

where the + symbol in a placeholder for an undetermined component of the solution.
Next, the feasible elements of parent 2 are copied to the offspring. These are the values that do not already appear in the offspring. So, the offspring becomes

$$[+, 8, 1, 2, 7, 6, +, +]$$

At this point, we have

$$\begin{array}{rcl} \text{parent 1} & = & [\ 3,\ 5,\ 1,\ 2,\ 7,\ 6,\ 8,\ 4\ ] \\ \text{parent 2} & = & [\ 1,\ 8,\ 5,\ 4,\ 3,\ 6,\ 2,\ 7\ ] \\ \text{(partial) offspring} & = & [\ +,\ 8,\ 1,\ 2,\ 7,\ 6,\ +,\ +\ ] \end{array}$$

When we look at the first position

- it was a 1 in parent 2, but we cannot copy that because 1 already appears in the offspring
- so, we look at where 1 appears in parent 1 and see that the corresponding value for parent 2 in that position is 5
- so, 5 goes in the first place in the offspring

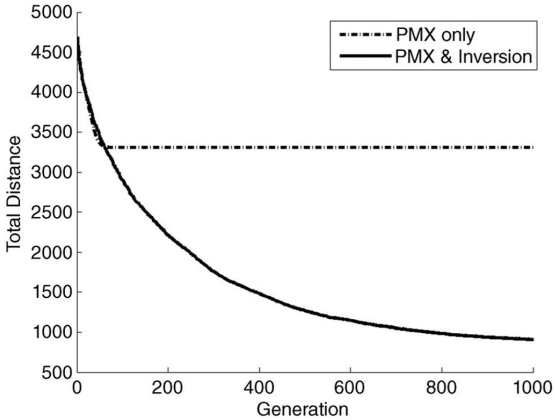
Similarly

- 4 goes in the seventh place and
- 3 goes in the eighth place

and the final offspring is

$$[5, 8, 1, 2, 7, 6, 4, 3]$$

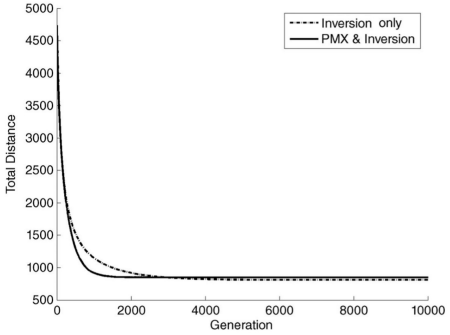
When the PMX operator is applied alone (on the same 100-city TSP) instead of the other mutation operators, the convergence stalls at a mean tour length - just less than 3000 units (the average results on 50 trials shown). In combination with inversion, convergence recovers.



The stalled convergence is caused by selection that eliminates the variation in the population and, subsequently, the PMX operator unable to generate anything new (it can only recombine existing solutions - once they look the same, any standard crossover operator becomes incapable of searching further. The results also show that adding in the possibility of mutating offspring via inversion allows the evolutionary search to proceed toward improved solutions.

Comparing PMX + inversion with inversion alone (to see if PMX is offering any benefit to the search) shows that up to generation 3000, the combination of PMX and inversion generated faster improvement than inversion alone, on average. After the 3000th generation, inversion alone did better on average at fine-tuning toward better solutions.

Perhaps it would make sense to use different variation operators at different times, a.k.a. self-adaptation.



There is a wide variety of combinatorial optimization problems that can be addressed using EA

- extensions of canonical problems (TSP), e.g. optimizing the routing of delivery trucks for a major logistics company
  - each truck must be loaded optimally in order to provide each driver the opportunity to deliver the products on time
  - not all customers are equally important; some deserve higher priority
  - not all drivers are capable of handling the same equipment; some are rated to drive larger vehicles.

  Determining the best way to allocate materials to trucks, assigning drivers to vehicles, and routing the vehicles to their destination present a complicated real-world problem.
- Other forms of combinatorial problems that are related to optimal subset selection problems
  - mathematical regression: selection of subset of independent variables the provide the most explanatory power for observed data
  - ML applications (determining the appropriate number of neurons and topology for a NN, or the number and shape of membership functions in a FCS).
  - data structures of variable size such as finite-state automata and symbolic expression trees also pose opportunities for combinatorial optimization