Automatically Discovering Euler's Identity via Genetic Programming

Konstantine Arkoudas
Rensselaer Polytechnic Institute
Department of Cognitive Science
arkouk@rpi.edu

Abstract

We show that by using machine learning techniques (genetic programming, in particular), Euler's famous identity ($V - E + F = 2$) can be automatically discovered from a limited amount of data indicating the values of V, E, and F for a small number of polyhedra—the five platonic solids. This result suggests that mechanized inductive techniques have an important role to play in the process of doing creative mathematics, and that large amounts of data are not necessary for the extraction of important regularities. Genetic programming was implemented from scratch in SML-NJ.

Introduction

In this paper we show that, using machine learning techniques, Euler's identity ($V - E + F = 2$) can be automatically discovered in a few seconds from a limited amount of data indicating the values of V, E, and F for certain types of polyhedra. It should be noted that Euler himself arrived at the identity inductively, by looking at the numbers of vertices, edges, and faces of various polyhedra, formulating hypotheses on the basis of those numbers, testing the hypotheses on new polyhedra, etc.; he did not have a deductive proof when he first announced the identity. The data table that we use is shown in figure , taken from Pólya (Pólya 1990, p. 36). It depicts the values of V, E, and F for nine polyhedra. Pólya, stressing the experimental and inductive aspects of mathematical discovery, writes that the table "is somewhat similar to the notebook in which the physicist enters the results of his experiments," and that a mathematician would examine such a data table in an "attempt at establishing a thoroughgoing regularity." And he quotes Laplace's pronouncement that "even in the mathematical sciences, our principal instruments to discover the truth are induction and analogy."

Exactly how common it is for mathematicians to develop conjectures by way of induction appears to be an empirical question of fact, although the issue is not as simple as it might seem at first glance.[1] But the broader question of whether inductive techniques can be profitably used in the aid of mathematical discovery has an eminently positive answer. Indeed, our results here indicate that *automatic* learning techniques based on symbolic representations (e.g., genetic programming, inductive logic programming, etc.) could have an important role to play in the process of doing creative mathematics. Such techniques are very efficient in extracting patterns from data and could thus be helpful in suggesting the type of "thorougoing regularities" that interest mathematicians.[2]

Remarkably, our program does not even need the nine entries of figure to discover Euler's identity. It suffices to provide the values of V, E, and F for the five regular polyhedra—the so-called *Platonic solids*. With either set of data our program converges to Euler's identity in a few generations.

Briefly, the main technical characteristics of our algorithm are as follows: a maximum of 50 generations, a population of 4000 abstract syntax trees, a generation size of 3600 programs (90% of the population size), a maximum tree depth of 13, ramped half-and-half initialisation, 3 non-terminals and 12 terminals (three of the latter being the variables V, E, and F, the other nine being random ephemeral integer constants in the $1, \ldots, 9$ range), and a tournament size of 5. Crossover was the only genetic operation used; mutation was not necessary. Section provides a brief overview of genetic programming.

The algorithm was implemented in SML-NJ (Standard ML of New Jersey). The source code can be downloaded from `www.rpi.edu/~arkouk/euler/`.

Genetic Programming

In genetic programming (GP) the goal is to automatically construct a computer program that solves a specific problem. The basic idea is to randomly generate an initial population of programs (typically a few thousands) and then iteratively apply selection pressure by mating "fit" programs to produce

[1]In particular, there are difficult philosophical issues concerning the extent of the influence of theory on observation and data collection. For instance, the data shown in the table of figure 1 could not even be assembled until the theoretical concept of an edge was available (a conceptual innovation of Euler).

[2]After completing the present paper, we became aware of an article by Colton and Muggleton (2006) on applications of inductive logic programming (ILP) in mathematics. Like our paper, that work encourages the use of inductive machine-learning techniques in the process of doing creative mathematics, but focusing on ILP in particular. We are not aware of any other applications of genetic programming in this domain.

	Polyhedron	F	V	E
1.	Cube	6	8	12
2.	Triangular prism	5	6	9
3.	Pentagonal prism	7	10	15
4.	Square pyramid	5	5	8
5.	Triangular pyramid	4	4	6
6.	Pentagonal pyramid	6	6	10
7.	Octahedron	8	6	12
8.	Tower	9	9	16
9.	Truncated cube	7	10	15

Figure 1: The numbers of faces, vertices, and edges for nine polyhedra.

offspring—new programs that become members of the new generation. This assumes that we have a mechanical way of measuring a program's fitness. Typically this is done by executing the program on a number of inputs and comparing the obtained results to the correct values. The smaller the deviation, the better the program. Mating is performed purely syntactically: Viewing two parent programs as abstract syntax trees, a random node is selected in each tree and the subtrees rooted there are swapped. This operation is called *crossover*. Mutation might also be used in order to maintain genetic diversity. In the rest of this section we will discuss these ideas in more detail.

Genetic programming can be seen as a particular application of a more general scheme known by the collective name of *genetic algorithms* (GA). The idea behind GA is to search a space $\mathcal{S}$ of solutions to a given problem, using the evolutionary principle of natural selection as a guide. For example, suppose that we are interested in the satisfiability problem: Given a CNF formula of propositional logic[3] over n atoms (Boolean variables) $A_1, \ldots, A_n$, find an assignment of truth values to the atoms that makes the formula true. In this case, a solution can be represented by a string of n bits, where the i^{th} bit indicates the value assigned to atom A_i (with 0 standing for false and 1 for true). Accordingly, the search space $\mathcal{S}$ is the set of all bit strings of length n.

We assume we have a fitness function that can assign a numerical score to any element of the search space. If we view the genetic algorithm as an optimizer, then this is the function we are trying to optimize. For the satisfiability problem, a fitness measure for a given assignment $\sigma \in \mathcal{S}$ might be the number of clauses of the formula that come out true under σ. A score of zero signifies a very unfit solution, while a score equal to the number of clauses indicates a perfect solution, i.e., a satisfying assignment.

We also assume that a collection of reproduction operators are available which can take one or more individual elements of the search space, combine or modify their genetic material, and produce one or more offspring objects. The two standard genetic operators are crossover and mutation. Crossover takes two individuals (the parents) and produces one or two new individuals (the children) whose genetic material is a recombination of the genetic material of the parents. For instance, if individuals are represented by bit strings of length n and the two parents are $a_1 \cdots a_n$ and $b_1 \cdots b_n$, then we might randomly choose a positive integer $i < n$ as the splitting point and produce the bit string $a_1 \cdots a_i \, b_{i+1} \cdots b_n$ as the offspring. Alternatively, we might produce two children by swapping bit segments, resulting, say, in $a_1 \cdots a_i \; b_{i+1} \cdots b_n$ and $b_1 \cdots b_i \; a_{i+1} \cdots a_n$. Mutation takes a single parent and produces a child by randomly altering the genetic material of the parent at a small number of points, e.g., by randomly flipping a few bits (typically only one).

Specifically, the standard genetic algorithm attempts to evolve a solution to the problem as described below. The algorithm is parameterized over three integer quantities $P > 0$, $1 \leq G \leq P$, and $N > 0$, respectively denoting the population size, the generation size, and the maximum number of iterations.

1. Set $i \leftarrow 1$.
2. Construct an initial random population of P possible solutions, called *individuals*. This can be regarded as a small subset of the overall search space $\mathcal{S}$, and a starting point from which to launch the evolution process. Randomness is crucial in ensuring that disparate regions of the search space are represented in the initial population. The symbolic representation of individuals varies depending on the problem. Oftentimes they are encoded as bit strings of a fixed length.
3. If $i > N$ then halt.
4. Determine and record the fitness of each individual in the population. (Other statistics might also be collected at this stage.) If the fittest individual solves the problem, then halt and announce the solution.
5. Create a new population as follows:
   - Pick $P - G$ individuals and copy them over directly into the new population.
   - Create G new individuals by repeatedly applying crossover or mutation to randomly selected parents from the current population.

   Note that every time individuals need to be randomly selected from the population, the probability of an individual being chosen should be based on its fitness: fitter individuals should be more likely to be selected. At the end of these two steps we have a new population of $(P - G) + G = P$ individuals.
6. Set $i \leftarrow i + 1$ and go to step 3.

The selection mechanism used in step 5 is left unspecified, and may be regarded as another parameter of the algorithm. It is nevertheless vitally important, as it is regulates the application of selection pressure. One of the most popular and efficient ways to choose an individual from a population is the method of *tournament selection*: $k > 0$ individuals are

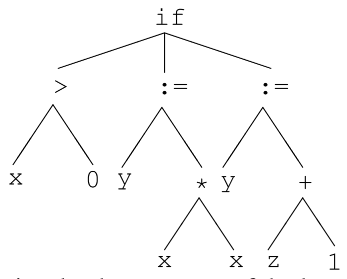
[3]CNF stands for conjunctive normal form.

first randomly picked, and then the fittest of the k is selected. The number k is called the *tournament size*. The smaller its value, the less the selection pressure. Indeed, if $k = 1$ then there is no selection pressure whatsoever: all individuals have an equally good chance of reproducing, or being regenerated or mutated. If $k = P$ then the selection pressure is excessive: Genetic operations will only be applied to the fittest individual in the whole population. This will immediately result in a complete loss of genetic diversity, which will almost always cause the algorithm to get stuck in a suboptimal solution. The relationship between the parameters G and P is also noteworthy: If $G = P$ then we have full population replacement at each step; if $G = 1$ then we have a steady-state algorithm. A parameterized implementation of the above algorithm based on tournament selection and using SML's functors will be discussed in the next section.

Genetic programming can now be understood as an instance of the general GA scheme in which the search space consists of all programs—up to a certain size—generated according to some abstract grammar. Individual programs are represented as abstract syntax trees (ASTs).[4] For instance, the program

```
if x > 0 then
 y := x * x
else
 y := z + 1
```

can be represented by the following AST:



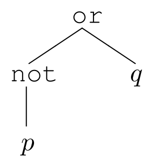
As noted previously, the grammar of the language is usually very simple. Issues such as nested levels of lexical scope and variable capture rarely ever arise.

Internal nodes of ASTs are called *non-terminals*; leaves are *terminals*. Oftentimes the programs are functional expressions over a number of input variables and certain function symbols. In that case every non-terminal is a function symbol of a certain arity (number of arguments), and every leaf is either a variable or a constant symbol (which can be regarded as a function symbol of zero arguments). Such expressions can be regarded as programs that produce a unique result once concrete values for the variables are given. Consider, for instance, the "program"

$$\texttt{or}(\texttt{not}(p), q)$$

which can be represented by the following AST:

[4]Also known as parse trees, since they are typically the output of parsing algorithms.



Executing this program for inputs $p \mapsto$ `false` and $q \mapsto$ `false` produces the result `true`.

The first step in formulating a genetic programming problem is to fix the set of non-terminals and the set of terminals that can appear in ASTs, which is tantamount to specifying the abstract syntax of the programs. An important requirement is *sufficiency*: A solution to the problem must be expressible as a program built from the chosen terminals and non-terminals. Suppose, for instance, that we are trying to learn a function of four floating-point arguments x_1, x_2, y_1, y_2 that represents the two-dimensional Euclidean distance between points (x_1, y_1) and (x_2, y_2):

$$\sqrt{|x_2 - x_1|^2 + |y_2 - y_1|^2}.$$

Then our terminal set must include the four variables x_1, x_2, y_1, y_2, and our non-terminal set must include symbols for the square-root function, the absolute-value function, binary addition and subtraction, and exponentiation. The semantics of the programs are given operationally, by providing a procedure that can execute an arbitrary program on arbitrary inputs. Typically this procedure will be used to evaluate the fitness of a program by executing it on a number of input values and determining the error of the output values.

In genetic programming crossover on two parent ASTs is performed by swapping two randomly chosen subtrees, one from each parent.[5] Figure illustrates this operation on two ASTs representing arithmetical expressions of the sort that were used in discovering Euler's identity; T_1—T_6 stand for arbitrary subtress. Mutation could also be used to replace a randomly chosen subtree of a parent with a new randomly generated tree. However, mutation was not used in this application[6].

The initial population is built by randomly creating the required number of ASTs. There are two ways of building a tree randomly, the "full" method and the "grow" method. In the former, the produced AST is full, that is, any two branches of it are of equal length.[7] This means that all terminals in the tree are at the same depth. ASTs produced by the grow method need not obey this constraint. Trees produced by this method exhibit a greater variety in their shape and structure. Usually the initial population is generated with a combination method called *ramped half-and-half initialisation*: Half of the trees are built using one method, and the other half using the other method.

[5]This always results in two children. Other variations of crossover are possible, e.g., producing only one child.

[6]Koza (1992) has argued that crossover usually maintains sufficient levels of diversity and that therefore mutation is not necessary in most cases, and can indeed decrease performance by introducing untested genetic material into the population. Although this is somewhat controversial (Angeline 1997; Chellapilla 1998), in practice most implementations of GP use little or no mutation.

[7]A tree branch is any path from the root to a leaf.

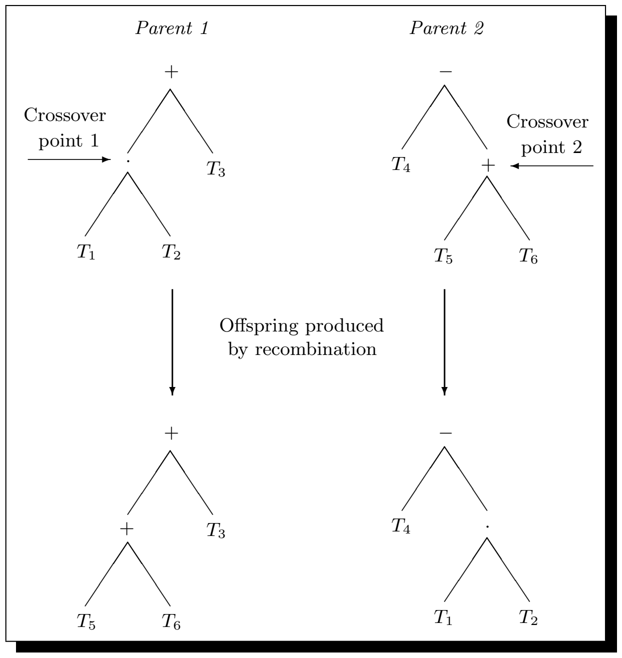


Figure 2: Illustration of the crossover operation on ASTs. The subtrees rooted at the crossover points are swapped. This is intended to simulate the exchange of genetic material.

Our formulation of the problem

The problem can be cast as a standard symbolic regression problem, where we view F as a function of two variables, E and V, and search the space of all rational (non-transcedental) functions for a function with maximal fitness. The generated "programs" would be abstract syntax trees built from four binary operations (addition, subtraction, multipication, and protected division), along with numeric constants and the two variables V and E. The fitness function would be implemented in the standard way, by tabulating, for each data point, the difference between the actual number of edges and the result produced by the program for given values of V and E. We have built such an implementation that quickly converges to the solution (for a population of 7000, generation size of 6300, and tournament size of 6).

In what follows we take an alternative approach, by attempting to discover a mathematical relationship between V, E, and F that agrees with ("covers") all the examples of figure . The relationship also needs to exclude certain possibilities; this is a rather important point that will be discussed shortly. We consider relationships of the form $T_1 = T_2$, where the terms T_1 and T_2 can be arbitrary polynomials over the three variables V, E, and F. More precisely, our search space will contain all identities I generated by the following abstract grammar and having depth no more than 13:

$$\begin{array}{lcl} I & \rightarrow & T_1 = T_2 \\ T & \rightarrow & L \mid T_1 + T_2 \mid T_1 - T_2 \mid T_1 \cdot T_2 \\ L & \rightarrow & V \mid E \mid F \mid n \end{array}$$

Here L ranges over leaves, where a leaf is either one of the variables V, E, F, or a positive integer n. Of course our set of terminals needs to be finite, so we cannot have infinitely many numerals, one for each positive integer. Instead, we used the nine numerals $1, \ldots, 9$. Any other integer, positive or negative, can be built up from these nine numerals via the available numeric operators (in fact addition and subtraction would suffice); zero can be expressed as, say, $(V - V)$. So our set of terminals is $\{V, E, F, 1, \ldots, 9\}$. The set of non-terminals is $\{+, -, \cdot\}$. The identity sign is not, strictly speaking, one of our non-terminals, since it can only appear at the root of a parse tree; there are no expressions such as $(2 = V) + 3$.[8] The relevant SML data types are as follows:

```
datatype variable = V | E | F;
datatype bin_op = plus | minus | times;
datatype term =
  var of variable | const of int |
  app of bin_op * term * term;
type ident = {left:term,right:term};
```

Genetic programming is heavily stochastic and makes extensive use of randomization. SML-NJ provides a built-in library module for generating pseudo-random numbers (structure `Rand` of the basis library), but it is based on a rather obsolete class of algorithms, namely, linear congruence generators. We instead used the relatively recent Mersenni Twister pseudorandom generator of Matsumoto and Nishimura (1998), which was implemented in SML by Michael Neophytos.[9] The Mersenni Twister produces extremely high-quality pseudorandom numbers and is very efficient. Using the Mersenni Twister, we implemented a function `getRandomInt:int -> int` such that `getRandomInt(k)` returns a random integer in the range `1,...,k`, for any positive integer `k`. This is the only randomization function that was necessary in the project. A particularly useful function that we defined in terms of it is `flipCoin:unit -> bool`, which simulates Bernoulli trials:

```
fun flipCoin() = getRandomInt(2) = 1;
```

Random leaves and internal nodes can now be generated as follows:

```
fun chooseRandomLeaf() =
  case (getRandomInt 12) of
    10 => var(V)
  | 11 => var(E)
  | 12 => var(F)
  | i => const(i));

fun chooseRandomFunSym() =
  case (getRandomInt 3) of
      1 => plus
  | 2 => minus
  | _ => times
```

[8] In GP parlance, this means that the non-terminal set $\{+, -, \cdot, =\}$ does not satisfy the *closure* requirement, which dictates that *any* term built from the non-terminals and terminals should be well-typed (i.e., it should denote a valid result in accordance with the evaluation semantics of terms) as long as the arities are respected. There are various ways of relaxing this constraint, but in our case the issue did not present a problem, since the three numeric operators do satisfy closure and the identity sign always appears at the top.

[9] The name of the algorithm derives from the fact that its period is the gargantuan Mersenne prime $2^{19937} - 1$.

The following function was used to randomly generate a term of a given depth d using the full method:

```
fun makeRandomFullTerm(d) =
     if d < 2 then
        chooseRandomLeaf()
     else app(chooseRandomFunSym(),
              makeRandomFullTerm(d-1),
              makeRandomFullTerm(d-1))
```

A similar function `makeRandomTermGrow:int -> term` implemented the growing method. The higher-order function `makeRandomTerm` takes one of the two functions above and uses it to generate a term of random depth:

```
fun makeRandomTerm(method) =
  method(chooseRandomDepth())
```

where `chooseRandomDepth` returns a random integer in the `1,...,max_depth` range. A random identity can now be generated as follows:

```
fun makeRandomIdent() =
  let fun makeTerm() =
    if flipCoin() then
      makeRandomTerm(makeRandomTermGrow)
    else
      makeRandomTerm(makeRandomFullTerm)
  in
     {left=makeTerm(),right=makeTerm()}
  end;
```

Term mating (crossover) is implemented straightforwardly, with two parent terms always giving rise to two children, either of which might be deeper than the maximum depth. A positional system based on lists of positive integers is used for subterm navigation. Specifically, given any term T and position p (where p is a list of positive integers), we write T/p for the subterm of T located at position p. This partial function can be defined by structural recursion as follows:

$$\begin{array}{rcl} T/[] & = & T \\ f(T_1,\ldots,T_n)/i :: p & = & T_i/p \end{array}$$

For instance, if T is parent 1 in figure , then $T/[1,2] = T_2$. See Baader and Nipkow (section 3.1, 1999) or Courcelle (1983) for more details on terms, term positions, etc.

To mate two identities $S_1 = S_2$ and $T_1 = T_2$, we first choose a random term S_i from the first identity and a random term T_j from the second identity, $i, j \in \{1, 2\}$. We then mate S_i and T_j to produce two offspring S_i' and T_j'. If either of these is deeper than allowed, it is discarded and the original parent (S_i or T_j) is used in its place. Two new identities are produced by replacing S_i and T_j in the original identities by S_i' and T_j', respectively.

The fitness of an identity is measured by evaluating it on the nine data points of the table in figure . (By a data point we mean a triple of integers (i_1, i_2, i_3) representing the values of V, E, and F, respectively, for a given polyhedron.) The more points it covers, the better. Take, for example, the identity $V = F$. This identity holds for the square pyramid, the triangular pyramid, the pentagonal pyramid and the tower; it fails for the remaining five polyhedra. It thus scores 4 out of 9 points. To determine whether an identity covers a data point, we need to be able to evaluate an arbitrary term once we are given values for the variables V, E, and F. By an *environment* we will mean a function from `variable` to `int`. Accordingly, what we need is an interpreter

```
eval:term -> env -> int
```

that takes a term, followed by an environment, and produces the appropriate output. This is implemented as follows:

```
fun eval (var x) env = env x
  | eval (const i) _ = i
  | eval (app (bop,t1,t2)) env =
      (intOp bop) ((eval t1 env),
                   (eval t2 env))
```

where `intOp` returns the SML numeric operation corresponding to an element of the datatype `bin_op`:

```
val intOp = fn plus => op+
             | minus => op-
             | times => op*
```

An environment is readily constructed from a data point as follows:

```
fun makeEnv(v,e,f) =
  (fn V => v | E => e | F => f)
```

We can now determine whether an identity $T_1 = T_2$ holds in a given environment ρ by evaluating T_1 in ρ, evaluating T_2 in ρ, and checking to see whether the same result was obtained in both cases:

```
fun holds({left,right},env) =
  (eval left env) = (eval right env);
```

In this simple scheme, the minimum score an identity could attain would be 0 and the greatest would be 9. However, this fitness function does not take into account the fact that many trivial identities can achieve perfect scores vacuously; consider $V = V$, $1 = 1$, $V + E = E + V$, and so on. If we simply allocated points for agreeing with the data, the search would quickly produce a trivial identity as the optimal solution. To get around this problem, we introduce *negative data points*, i.e., data values for which we are sure that any correct identity must *not* hold. Such values are easy to find. For instance, any proposed identity ought to fail for $V = 0, E = 0$, and $F = 1$, as no polyhedron could possibly have a face without having any vertices or edges. Accordingly, let ρ^- be the environment that maps `V` and `E` to `0`, and `F` to `1`. We now only need to observe that a trivial identity such as $V = V$ holds in every environment, including ρ^-; while any plausible non-trivial identity will not hold in ρ^-. It is this distinction that will help us to steer clear of trivial solutions.

Specifically, our scheme for measuring the fitness of an identity $T_1 = T_2$ is the following. For each of the nine data points, construct an environment $\rho_1, \ldots, \rho_9$ that captures the corresponding values of the three variables. If $T_1 = T_2$ holds in ρ_i, it gains a point. And if it does *not* hold for the negative environment ρ^-, it gains an additional nine points. Therefore, the smallest possible score is now 0 and the maximum is 18. We also introduce a small bias in favor of shorter identities by imposing a penalty of $\lfloor n/20 \rfloor$ points for an identity of length n. Thus an identity of length 100 would incur a penalty of 5 points, while one of length less than 20 would not incur any.

For greater modularity, and particularly in order to be able to run an experiment with different sets of data, we make the evaluation of an identity a function of an arbitrary list of data points (triples of integers). This way, we can evaluate an identity on the data of figure , or on the data for the five platonic solids, or indeed on any given list of data points. The scheme is the same: Given a list of n such triples, an identity scores a single point for every triple that it covers; and it wins n additional points if it does not cover the negative data point $(0, 0, 1)$:

```
val neg_env = makeEnv (0,0,1);

fun evalIdentForGivenPolyhedron
      (identity,data_point) =
  let val pos_env = makeEnv data_point
  in
    (if holds(identity,pos_env)
       then 1 else 0)
    handle _ => 0
  end;

fun identFitness(identity,data) =
  let val scores =
        List.map
        (fn data_point =>
             evalIdentForGivenPolyhedron
               (identity,data_point))
         data
      val pos_points =
        List.foldr op+ 0 scores
      val neg_points =
        (if not(holds(identity,neg_env))
          then List.length(data) else 0)
        handle _ => 0
      val sum = pos_points + neg_points
      val size_penalty =
        (identSize identity) div 20
  in
    sum - size_penalty
  end;
```

Note that if an error such as an overflow occurs during the evaluation of an identity in a given environment, then no points are awarded.

A functor-based implementation of genetic algorithms using tournament selection can be downloaded from `www.rpi.edu/~arkouk/euler/`. It is based on the following two signatures:

```
signature GA_PARAMS =
sig
  type individual

  val individualToString:
    individual -> string
  val makeRandomIndividual:
    unit -> individual
  val crossOver:
    individual * individual ->
    individual * individual
  val mutate:
    (individual -> individual) option

  val population_size: int
  val generation_size: int
  val tournament_size: int
  val max_generations: int
end

signature GA =
sig

  structure G: GA_PARAMS
  type population

  val makeInitPopulation:
    unit -> population
  val selectIndividual:
    population -> G.individual
  val findFittest:
    population -> G.individual * real
  val evolve:
    population  -> population
  val solve:
    unit -> G.individual option
end
```

The signature `GA_PARAMS` captures the parameters of the algorithm that will vary depending on the problem. Most of the items are self-explanatory. A mutation operator is optional. If one is provided, it is used to produce 10% of each successive generation; the crossover operator produces the remaining 90%.

The signature `GA` specifies the essential aspects of the GA scheme. Fitness values are real numbers, and optimization is tantamount to maximization.[10] The main component is `solve`, which will optionally produce a individual that is sufficiently fit to be considered a solution (or else fail). Note that a structure that satisfies this signature will contain a `GA_PARAMS` substructure. The idea is that a structure of type `GA` will be produced by a functor that will take a structure of type `GA_PARAMS` as an argument. Also note that the fitnes function and goal predicate (which specifies when a given fitness is good enough) are not part of the parameter structure, and thus need to be supplied to the said functor separately, as values. This decoupling provides an extra degree of flexibility.

The parameters of the genetic algorithm for this particular problem are assembled in the following structure:

```
structure GA_PARAMS =
  struct
    type individual = ident;
    val makeRandomIndividual =
      makeRandomIdent;
    val individualToString = identToString;
    val crossOver = mateIdentities;
    val mutate = NONE;

    val population_size = 4000;
    val generation_size = 3600;
```

[10]Any other type of optimization function can be readily recast in this manner. For instance, if a fitness function

```
f:individual -> int
```

produces integer values instead, and if smaller values are better, then `fn x => Real.fromInt(~(f x))` will be an equivalent fitness function that maximizes real numbers.

```
    val tournament_size = 5;
    val max_generations = 50;
  end;
```

Two data lists, one corresponding to the table of figure and one for the five platonic solids are defined:

```
val (data_point_1,data_point_2,
     data_point_3,data_point_4,
     data_point_5,data_point_6,
     data_point_7,data_point_8,
     data_point_9) = ((8,12,6), (6,9,5),
                       ..., (10,15,7));

val data_table_1 =
  [data_point_1,data_point_2,data_point_3,
   data_point_4, data_point_5, data_point_6,
   data_point_7, data_point_8,data_point_9];

val (tetrahedron_data,
     dodecahedron_data,
     icosahedron_data) =
      ((4,6,4),(20,30,12),(12,30,20));

val data_table_2 = [data_point_1,
                    data_point_7,
                    tetrahedron_data,
                    dodecahedron_data,
                    icosahedron_data];

(* data_table_2 covers only the five
   platonic solids *)
```

A fitness function for the first data set and a corresponding goal predicate are defined, a genetic algorithm structure `S1` is created by applying the functor `makeGA` to these two values and the structure `GA_PARAMS`, and the `solve` function of `S1` is used to discover an appropriate identity:

```
fun fitness1
      (identity:GA_PARAMS.individual) =
  Real.fromInt
  (identFitness(identity,data_table_1));

fun goalFitness1(r) =
  Real.>=(r,Real.fromInt
    (Int.*(2,List.length(data_table_1))));

structure S1 =
  makeGA(structure ga_params = GA_PARAMS;
         val fitness = fitness1;
         val goalFitness = goalFitness1);

val _ = S1.solve();
```

Likewise for the second data set. In both cases the algorithm converges to Euler's identity within 4 to 8 generations.

References

Angeline, P. J. 1997. Subtree crossover: Building block engine or macromutation? In Koza, J. R.; Deb, K.; Dorigo, M.; Fogel, D. B.; Garzon, M.; Iba, H.; and Riolo, R. L., eds., *Genetic Programming 1997: Proceeding of the Second Annual Conference*. Morgan Kaufmann. 9–17.

Baader, F., and Nipkow, T. 1999. *Term Rewriting and All That*. Cambridge University Press.

Chellapilla, K. 1998. A Preliminary Investigation into Evolving Modular Programs without Subtree Crossover. In Koza, J. R.; Banzhaf, W.; Chellapilla, K.; Deb, K.; Dorigo, M.; Fogel, D. B.; Garzon, M.; Goldberg, D. E.; Iba, H.; and Riolo, R. L., eds., *Genetic Programming 1998: Proceeding of the Third Annual Conference*. Morgan Kaufmann. 23–31.

Courcelle, B. 1983. Fundamental properties of infinite trees. *Theoretical Computer Science* 25:95–169.

Koza, J. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Matsumoto, M., and Nishimura, T. 1998. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8(1):3–30.

Pólya, G. 1990. *Mathematics and Plausible Reasoning, Vol. 1: Induction and Analogy in Mathematics*. Princeton. Twelfth Printing.

S. Colton and S. Muggleton. 2006. Mathematical applications of inductive logic programming. *Machine Learning* 64(1–3):25–64.