EVOLUTIONARY COMPUTATION

INTRODUCTION

Evolutionary Computation is a general term for a group of techniques which solve complex problems quickly and efficiently by using principles derived from biological evolution and genetics. The two most prominent methods are genetic algorithms and genetic programming. The classic references are Holland (1975) for genetic algorithms (GAs) and Koza (1992) for genetic programming (GPs). Fundamental to each method is the notion of progressive adaptation towards the optimum or goal. To accomplish this task, each method draws on the processes of reproduction, crossover, and mutation as defined in genetic evolution.

Each method is concerned primarily with search and optimization in complex problem spaces. Neither is guaranteed of finding the optimal solution, but as approximation algorithms, they are powerful and frequently accurate. In this brief review, we will first address some terminology common to optimization problems, explore more traditional approaches and the domains in which they fail, and finally turn to GAs and GPs and why they work so well.

SOME BACKGROUND AND TERMINOLOGY

Optimization problems, mathematically stated, are concerned with finding the maximum or minimum of some function. This function is often used to represent profit, cost, quantity of materials, time, etc. In the neatly abstracted world of calculus, this is done simply and efficiently with derivatives. The maximal or minimal solution is realized by taking the partial derivative of the function, setting it equal to zero, and solving the resulting equation. For example, consider the equation $y = mx + b$. We are interested in the following analysis:

$$\frac{\partial y}{\partial x} = m, \quad m \overset{def}{=} 0$$

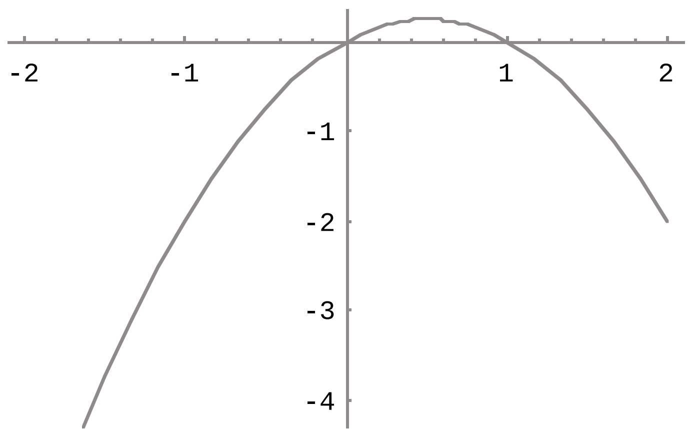
In this case, the derivative is trivial and x is not in the solution. Being unbounded, the maximum is then infinite. This makes sense because the function is the equation of a line. Without specifying a range for the value of x, the line will continue to increase to infinity. Consider, however, a function with a finite maximal value: $y = x\,(1 - x)$. In this case,

$$\begin{aligned} y &= x\,(1-x) \\ \frac{\partial y}{\partial x} &= 1 - 2x \overset{def}{=} 0 \\ 2x &= 1 \\ x &= \frac{1}{2} \end{aligned}$$

And, in fact, when we plot this function (Figure 1), we can see that 0.5 is indeed where the maximal value is found. Often, however, there are functions with multiple maxima (or minima[1]). In these cases, where does one estimate the derivative? Is there only one derivative function?

[1]Although we will speak primarily of maxima in this note, bear in mind that by the principal of duality, the methods apply equally to minima.

Figure 1



There are a variety of problems which cannot successfully be addressed in this fashion. Therefore, other techniques must be used. These techniques take the form of algorithms. An **algorithm** is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output. An algorithm is thus a sequence of computational steps that transform the input into the output (Cormen, Leiserson, and Rivest, 1990). In the case of optimization, the input is the function or the problem space and the output is the maximal or minimal (optimal) value.

When functions are more complicated that the ones mentioned above, an algorithm called **hill-climbing** is used. Without entertaining all the mathematical details[2], hill-climbing operates by always seeking out higher ground while never returning to lower ground. In this sense, you can think of a car driving along the surface of the function. The car only drives towards higher levels and if it can no longer attain a higher level in any direction, it stops, presuming it's at the maximal value. For simple functions this halting condition is valid. For multipeaked functions, the algorithm may stop prematurely at a **local maximum**.

For multipeaked functions, there exist multiple maxima (See Figure 2). The optimal solution to the problem requires a **global maximum**. The global maximum is the maximal value of the set of all local maxima. Algorithms such as hill-climbing often reach local maxima because they may reach a point in which incremental advancement in any direction would obtain a lower value. However, larger advancement in some direction would obtain a greater value.

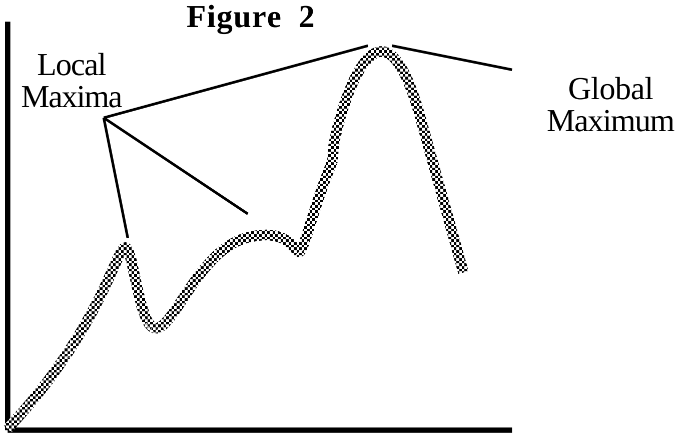
Multipeaked functions often occur in problems with **nonlinear** problem spaces. Other terms which are often encountered with nonlinearity are **continuous** and **monotonic**. For a function to be differentiable, it must be continuous. Hill-climbing and other derivative-based methods require continuous derivatives (GAs and GPs do not). In this sense, it is often required that functions be **twice-continuously differentiable** (*e.g.*, for algorithms such as Newton-Raphson). That is, that their first and second derivatives be real valued. A function which has real derivatives is called continuous. A (weakly) monotonic function is one which is everywhere either increasing or flat. Nonmonotonic functions can cause hill-climbing-type algorithms to get stuck in local maxima.

As a final set of terms used in this area, we will consider the computational side of optimization. Computational problems are often defined as members of complexity classes such as

[2]See the classic works of George Dantzig on linear programming and Richard Bellman on dynamic programming.

P or **NP** (among others).[3] This distinction refers (more or less) to the time required to compute

Figure 2



the solution to a particular problem (technically, the execution time in steps of the algorithm). In practical terms, they represent the distinction between tractable and "difficult" problems. For **P** problems, the time required to solve them grows in a polynomial fashion with the size of the problem (*e.g.*, $O(n^{\log 2})$). For **NP** problems, however, the time required to solve them grows in superpolynomial time - exponentially or faster (*i.e.*, *n* is in the exponent of the order value: $O(n^{2^n})$). Many problems involving graph theory (*e.g.*, shortest path algorithms), combinatorics (*e.g.*, Boolean networks), and queueing are in class **NP**. Problems shown to be **NP** (and especially **NP-COMPLETE**) are considered intractable and solutions must be found by way of approximation algorithms. Fortunately, GAs and GPs have been shown to be effective in a wide array of problems both **P** and **NP** -- even ones for which the vast majority of alternative solution methods cannot compare.

To more clearly see the distinctions between the types of functions which we are considering, observe the following four functions. They are presented with their fundamental characteristics and the optimization methods best suited for them.

[1] $$y = 5x + 2$$

•Linear, Monotonic, Continuous
•Standard calculus, hill-climbing, linear programming

[2] $$y = \frac{10x^2}{x^3} + \frac{8}{7}x$$

•Nonlinear, Nonmonotonic, Continuous
•Hill-climbing (with duality), nonlinear programming

[3] $$y = \begin{cases} e^{\sqrt{x}} & x < 10 \\ x \sin(2x) & x \geq 10 \end{cases}$$

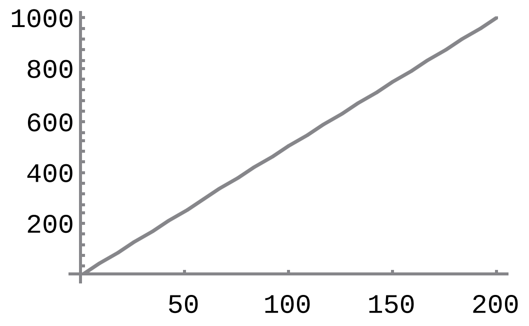
[3] P stands for **P**olynomial time and NP for **N**ondeterministic **P**olynomial time. Papadimitriou (1994) is a thorough, albeit advanced, reference for those interested in the theoretical side of computational complexity, particularly with regard to problems characterized as **NP** or worse. For those who are interested in the larger set of complexity classes, one can show that $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$. There are, however, even more classes.

•Nonlinear, Nonmonotonic, Discontinuous
•Genetic algorithms, genetic programming

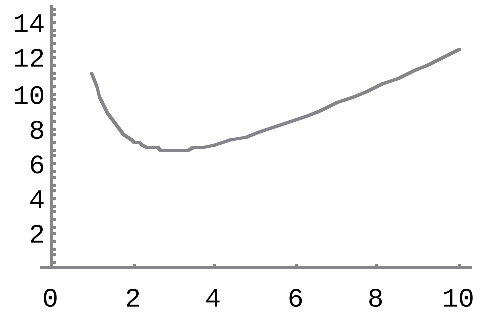
[4] $$y = \sin\left(\tanh\left(\sqrt{|\sin x|}\right) + \frac{x}{100}\right)$$

•Nonlinear, Nonmonotonic
•Genetic algorithms, genetic programming

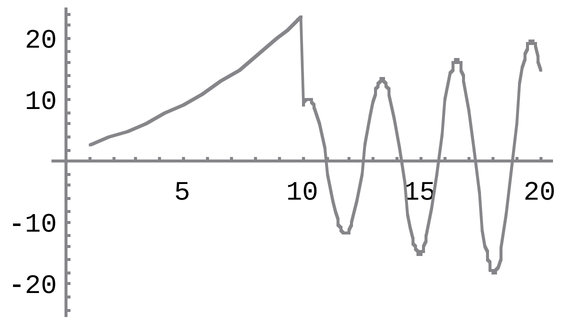
Graph 1



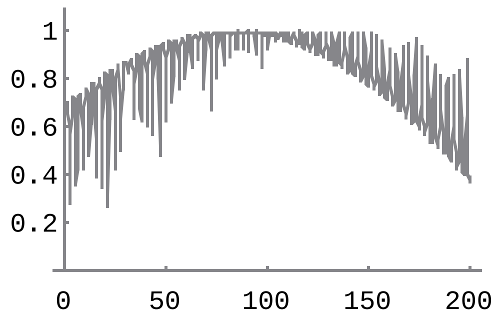
Graph 2



Graph 3



Graph 4



ALTERNATIVE METHODS

GAs and GPs are not the only methods which can be used to solve difficult computational problems. There are several other popular algorithms which can be used. All, however, are noticeably less powerful than these evolutionary methods. In this section, we will briefly consider four alternate methods for addressing complex problems: dynamic programming, randomized enumeration, simulated annealing, and heuristic approximation. Parts of this section are adapted from Chapter 1 of Goldberg (1989).

Dynamic programming is a highly specialized enumerative technique for solving programming problems which are not necessarily linear. The solution methodology used with classical dynamic programming is often some form of branch and bound. As with all of these techniques, there have been numerous variations of the same basic model which often improve the timing performance for certain types of problems. Still, however, dynamic programming suffers from the ailment common to enumerative techniques: it quickly gets overwhelmed in large problem spaces - particularly in problem spaces which are discontinuous or have step-level functions.

Randomized enumeration is a relatively inefficient method for solving these complex problems, although it is guaranteed to find the optimal solution. These methods use standard hill-climbing with the addition of a randomly determined "jump" to some other location in the problem space when the current hill-climbing process reaches a maxima. The random jump has the ability to determine (given an arbitrary time-period) whether or not the solution is a local optimum. Unfortunately, this frequently involves testing all possible values in the problem space. For all but the simplest of problems, this is not feasible.

Simulated annealing is a more intelligent approach to function optimization. At a very basic level, simulated annealing works by allowing the standard hill-climbing algorithm to make "mistakes" while climbing. That is, it allows the "driver" to head down with some probability rather than always having to search for increasing values. The probability with which a down move is allowed is regulated by the "temperature" of the function. The term temperature is used because the process is drawn from an analogy to actual annealing, which is a metallurgical process involving heating metals and cooling them slowly to form an optimal state in the finished metal. The temperature in a simulated annealing problem is a measure of the variability of the function to be optimized. The more variability in a function, the more likely the system is to allow a down move in search of a higher up move. The effect this has is to "blur" the graph of the function so that small irregularities in the function disappear and do not prevent the optimizing routine from reaching the global optimum. Simulated annealing is an excellent method for approaching optimization problems, but pales slightly in comparison to genetic methods in terms of robustness and efficiency.

Finally, **heuristic approximation** techniques are used. These techniques often involve some combination of the above techniques in conjunction with some prior knowledge of the problem. The prior knowledge (often about the location of a solution) is used to reduce the size of the problem space in order to allow these less-efficient routines to work more effectively. The chief advantage of heuristic methods is their simplicity. In environments where finding the absolute optimal solution is not critically important and the problem-solver can satisfice, heuristic approximations are often the best way to go. They can produce a decent solution extremely quickly with easily justifiable means. The downside is that the solutions they produce are often nowhere near the global optimum.

GENETIC ALGORITHMS

Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search (Goldberg, 1989). Genetic algorithms operate by using the biological principles of reproduction, crossover, and mutation to search efficiently for the solution. They differ from more conventional methods in four important ways:

[1] They work with a coding (usually binary or ternary) of the parameter set, rather than the actual parameters.

[2] They search concurrently over a population of points rather than point by point.

[3] They converge based on some objective function rather than derivatives or other auxiliary knowledge.

[4] They use probabilistic transition rules, rather than deterministic ones.

GAs operate on bit-strings or vectors of binary digits as an encoding of the problem. Over a series of iterations, the bit strings are reproduced according to their success at maximizing some fitness function. In the process of these iterations, they are probabilistically subjected to crossover and mutation. Fairly rapidly, the fitness of the population increases and an optimal or near-optimal solution is identified. It is important to note that genetic algorithms are not guaranteed to identify the optimal solution. They will always, however, reach a near-optimal solution.

To explain this process further, we will consider a simple example. Consider two bit-strings: σ_1 and σ_2.

$$\sigma_1 = \{0\ \ 0\ \ 1\ \ 1\ \ 1\}$$

$$\sigma_2 = \{1\ \ 0\ \ 1\ \ 0\ \ 1\}$$

We wish to observe the three processes of GAs on these strings. Let $f(\cdot)$ denote the fitness of a string. Assume according to some measure that $f(\sigma_1) = 25$ and $f(\sigma_2) = 75$. This indicates that the first string has 25% of the total fitness and the second string has 75% of the total fitness. One **reproduction** scheme is known as roulette-wheel reproduction. In reproducing the strings across generations proportional to fitness, we assign 25% of the space on a roulette wheel to the first string and the remaining 75% to the second string. Spinning the wheel determines the result of reproduction. This way, more fit strings are more likely to be reproduced.

The second operation GAs perform is crossover. The frequency of crossover is set as a parameter of the system and it thus determined exogenously. Determined randomly by the system is the crossover point. For example, with strings σ_1 and σ_2 assume that the crossover point has been identified as between elements three and four. After crossover the strings would look as follows:

$$\begin{cases} \sigma_1 = \{0\ \ 0\ \ 1\ \ |1\ \ 1\} \\ \qquad\qquad\qquad\ \ \updownarrow \\ \sigma_2 = \{1\ \ 0\ \ 1\ \ |0\ \ 1\} \end{cases}$$

$$\begin{cases} \sigma_1' = \{0\ \ 0\ \ 1\ \ 0\ \ 1\} \\ \sigma_2' = \{1\ \ 0\ \ 1\ \ 1\ \ 1\} \end{cases}$$

As a result of crossover, the fitness of the second string has increased and that of the first has decreased. Gradually, the first string will be "bred out" of consideration because as its fitness drops, the probability with which is can reproduce drops.

The third and final operator in a genetic algorithm is mutation. The mutation rate is also determined as a parameter of the system. It is generally set to be quite small (in practice, often less than 1%) since mutation is a destructive operation. It can often have the effect of changing a nearly optimal solution into a far-from-optimal solution. Still, it has value in precluding stopping at local optima. Mutation operates by randomly selecting a bit from each string and, if the probability of mutation is realized, "flipping" the bit so that it takes on a new value. If the bit was previously a 1 it is now as 0 and vice versa (obviously, ternary and *n*-ary strings are adjusted accordingly).

Fundamental to the operation and robust efficiency of genetic algorithms is the notion of the schema. Holland’s (1975) notion of the schema and his Schema Theorem, which Goldberg (1989) terms the Fundamental Theorem of Genetic Algorithms prove in detail why genetic algorithms are such powerful search techniques. The details of schemata are truly beyond the scope of this class. What is essential to note, however, is that the schema theorem proves that genetic algorithms are so efficient because they operate with **implicit parallelism**.[4] That is, they are capable of searching over vast fields of possible solutions simultaneously. This is something which other solution techniques cannot accomplish.

GENETIC PROGRAMMING

Years after the development and popularization of GAs, John Koza began working on a method of the automatic programming of computers. He noted that previous attempts at machine learning achieved their results by manipulating specialized structures (*e.g.*, the bitstrings in genetic algorithms, production rules, formal grammars, etc.) and felt that this was an unsatisfactory manner in which to have computers solve problems without being explicitly programmed (Koza, 1992). He claimed that the structures which need to be manipulated are computer programs themselves.

Therefore, genetic programming operates by manipulating the tree structures which comprise the program. The process begins with a randomly selected population of computer programs. Through the processes of reproduction, crossover, and mutation, the programs evolve over time to produce better and better programs according to some fitness measure. The fitness measures are usually quite intuitive. For example, if the problem concerns the optimal allocation of corporate resources, the fitness function could be profit - more being better than less. If the problem concerns creating a good randomizing routine, the fitness could be entropy - the higher the entropy, the better the randomizer.

The basic operations are carried out almost identically to their counterparts in genetic algorithms. However, there are some critical differences. For one, by using computer programs as the structures, any operations on the programs may render them syntactically or grammatically invalid. Consider two branches (here, as LISP S-expressions):

```
(if-then-else (> x 10) (print achieved) (do (x 1 (1+ x)))

(rand 0 999)
```

When considering crossover and mutation, it is important in genetic programming to recognize the grammatical structure of the trees. One cannot exchange a function node with a terminal node and expect to get a valid program. For example, if crossover produced:

```
(rand print (> x 10))
```

---

[4]To perhaps better illustrate the critical importance of parallel search, consider that if it were possible to test 1 billion points per second (Deep Blue, the massively parallel supercomputer which bested Garry Kasparov at chess in 1997, was only capable of 200 million calculations per second) and that a blind random search program had been running since the beginning of the universe (some 15 billion years ago), it would only have searched 10^{27} or approximately 2^{90} points by now. That corresponds to a binary string with a length of merely 90 binary digits. Such a length is hardly sufficient to adequately characterize many complex problems. John Miller’s (1988) paper on automata playing a simple repeated prisoner’s dilemma game reported a space of 2^{148} values and Koza’s (1992) Boolean 11-Multiplexer problem has a search space of an astounding 2^{2048} values! This complicated problem, at 1 billion calculations per second, would take $\approx 10^{600}$ years to solve with blind random search.

the program would cease to have meaning and would crash because the program is being asked to generate a random number between the command `print` and the inequality $x > 10$. Facilities must be in place to present such meaningless crossover. Within these constraints, however, the procedures are the same.

In the context of function optimization, consider the following representation of genetic programming. Koza (1992) considers the problem (initially used in a thesis by DeJong) of finding the global optimum (minimum, in this case) of the following five dimensional function:

$$f(x_1, x_2, x_3, x_4, x_5) = (x_1 - 1)^2 + \left(x_2 - \sqrt{2}\right)^2 + \left(x_3 - \sqrt{3}\right)^2 + (x_4 - 2)^2 + \left(x_5 - \sqrt{5}\right)^2$$

Although it cannot be graphed, one can show that the minimum value is obtained for $\mathbf{x} = \left\{1, \sqrt{2}, \sqrt{3}, 2, \sqrt{5}\right\}$. The genetic program is allowed to use the four basic arithmetic functions (add, subtract, multiply, and divide) plus a function called `list` which is simply 5 sample values for each of the unknown *x*s.

Over a series of trials, the following structure (LISP S-Expression) is obtained:

```
(LIST (- (* (* (- (+ (÷ -0.7672 -0.6696) 0.793) (- (÷
         0.7384 -0.338303) -0.0279999)) -0.0279999) (*
         -0.788605 (÷ -0.2602 -0.9012))) (- -0.468903
         0.50569))
      (+ (+ 0.65669 0.810394) (+ -0.139603 0.086395))
      (- (÷ -0.375504 -0.246803) (* -0.788605 (÷
          -0.2602 (- -0.468903 0.50569))))
      (+ (- 0.804504 (÷ 0.5775 (÷ (- -0.6696 0.50569)
         (÷ 0.541 0.76779)))) (- 0.804504 (* (- (* (-
         (+ 0.50569 (* -0.9012 (* -0.066803 (÷ -0.2602
         -0.9012)))) (+ 0.810394 0.195404)) (*
         -0.279404 -0.0108032)) (+ (÷ 0.816696
         -0.451797) (+ 0.6409 -0.451797))) -0.0279999)))
      (+ (- 0.804504 (- -0.042 0.244095)) (÷ -0.7672
         -0.6696)))
```

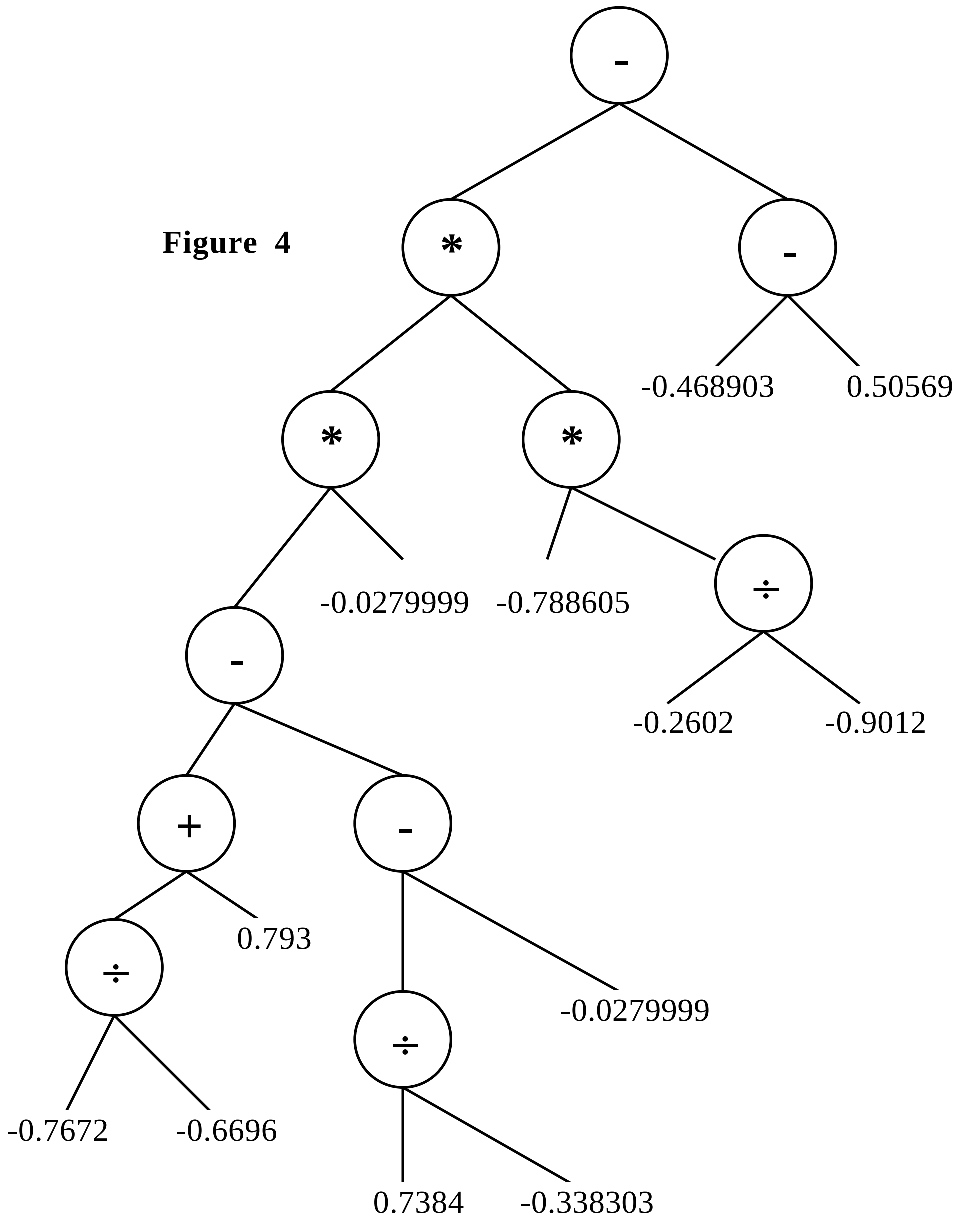
which, upon reduction, is equivalent to

```
(LIST 1.0007 1.414 1.732 2.0002 2.2364)
```

which, in turn, is within rounding error of the exact solution. For further clarification, Figure 4 displays the tree structure of the first argument in the `LIST` structure above which begins with `(- (* (*`...

Lastly, it is worthwhile to note that the approximately optimal expression above was obtained within 45 trials - a trivial amount of computational time for any problem, let alone one this complicated.

Figure 4



References

Cormen, T., C. Leiserson, and R. Rivest. *Introduction to Algorithms*. (Cambridge, MA: MIT Press, 1990).

Goldberg, D. *Genetic Algorithms in Search, Optimization, and Machine Learning*. (Reading, MA: Addison-Wesley, 1989).

Holland, J. *Adaptation in Natural and Artificial Systems*. (Ann Arbor, MI: University of Michigan Press, 1975).

Koza, J. *Genetic Programming*. (Cambridge, MA: MIT Press, 1992).

Papadimitriou, C. *Computational Complexity*. (Reading, MA: Addison-Wesley, 1994).