Version 2 - Submitted ---, 1999 for EUROGEN workshop in Jyvdskyld, Finland on May 30 – June 3, 1999.

Genetic Programming: Turing's Third Way to Achieve Machine Intelligence

J. R. KOZA[1], F. H BENNETT[2] III, D. ANDRE[3], AND M. A. KEANE[4]

[1]Stanford University
Stanford, California USA

[2]Genetic Programming Inc.
Los Altos, California USA

[3]University of California
Berkeley, California USA

[4]Econometrics Inc.
Chicago, Illinois USA

1. THE PROBLEM OF AUTOMATIC PROGRAMMING

One of the central challenges of computer science is to get a computer to solve a problem without explicitly programming it. In particular, it would be desirable to have a problem-independent system whose input is a high-level statement of a problem's requirements and whose output is a working computer program that solves the given problem. The challenge is to make computers do what needs to be done, without having to tell the computer exactly how to do it.

Alan Turing recognized that machine intelligence may be realized using a biological approach. In his 1948 essay "Intelligent Machines" [9], Turing made the connection between search techniques and the challenge of getting a computer to solve a problem without explicitly programming it.

> Further research into intelligence of machinery will probably be very greatly concerned with "searches" ...

Turing then identified three broad approaches by which search techniques might be used to automatically create an intelligent computer program.

One approach that Turing identified is a search through the space of integers representing candidate computer programs. This approach reflects the orientation of much of Turing's own work on the logical basis for computer algorithms.

A second approach that Turing identified is the "cultural search" which relies on knowledge and expertise acquired over a period of years from others. This approach is akin to present-day knowledge-based systems and expert systems.

The third approach that Turing identified is "genetical or evolutionary search." Turing said,

> There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being the survival value. The remarkable success of this search confirms to some extent the idea that intellectual activity consists mainly of various kinds of search.

Turing did not specify in this essay how to conduct the "genetical or evolutionary search" for a computer program. However, his 1950 paper "Computing Machinery and Intelligence" [24] suggested how natural selection and evolution might be incorporated into the search for intelligent machines.

> We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it is better or worse. There is an obvious connection between this process and evolution, by the identifications
>
> Structure of the child machine = Hereditary material
>
> Changes of the child machine = Mutations
>
> Natural selection = Judgment of the experimenter

Since the 1940s, an enormous amount of research effort has been expended on trying to achieve machine intelligence using what Turing called the "logical" approach and the "cultural" approach. In contrast, Turing's third way to achieve machine intelligence – namely, "genetical or evolutionary" approach – has received comparatively less effort.

This paper is about genetic programming – a way to implement Turing's third way to achieve machine intelligence. Genetic programming is a "genetical or evolutionary" technique that automatically creates a computer program from a high-level statement of a problem's requirements. In particular, genetic programming is an extension of the genetic algorithm described in John Holland's pioneering 1975 book *Adaptation in Natural and Artificial Systems* [7]. Starting with a primordial ooze of thousands of randomly created computer programs, genetic programming progressively breeds a population of computer programs over a series of generations. Genetic programming employs the Darwinian principle of survival of the fittest, analogs of naturally occurring operations such as sexual recombination (crossover), mutation, gene duplication, and gene deletion, and certain mechanisms of developmental biology [2, 3, 4, 10,12 – 19, 23].

When we talk about a computer program (fig. 1), we mean an entity that receives inputs, performs computations, and produces outputs. Computer programs perform basic arithmetic and conditional computations on variables of various types (including integer, floating-point, and Boolean variables), perform iterations and recursions, store intermediate results in memory, contain reusable groups of operations that are organized into subroutines, pass information to subroutines in the form of dummy variables (formal parameters), receive information from subroutines in the form of return values (or through side effects). The subroutines and main program are typically organized into a hierarchy.

---Fig. 1 is at end of paper and is also provided as a separate file in EPS format

Fig. 1 A computer program.

We think that it is reasonable to expect that a system for automatically creating computer programs should be able to create entities that possess most or all of the above capabilities (or reasonable equivalents of them). A list of attributes for a system for automatically creating computer programs might include the following 16 items:

- **Attribute No. 1 (Starts with "What needs to be done"):** It starts from a high-level statement specifying the requirements of the problem.
- **Attribute No. 2 (Tells us "How to do it"):** It produces a result in the form of a sequence of steps that satisfactorily solves the problem.
- **Attribute No. 3 (Produces a computer program):** It produces an entity that can run on a computer.
- **Attribute No. 4 (Automatic determination of program size):** It has the ability to automatically determine the number of steps that must be performed and thus does not require the user to prespecify the exact size of the solution.

- **Attribute No. 5 (Code reuse):** It has the ability to automatically organize useful groups of steps so that they can be reused.
- **Attribute No. 6 (Parameterized reuse):** It has the ability to reuse groups of steps with different instantiations of values (formal parameters or dummy variables).
- **Attribute No. 7 (Internal storage):** It has the ability to use internal storage in the form of single variables, vectors, matrices, arrays, stacks, queues, lists, relational memory, and other data structures.
- **Attribute No. 8 (Iterations, loops, and recursions):** It has the ability to implement iterations, loops, and recursions.
- **Attribute No. 9 (Self-organization of hierarchies):** It has the ability to automatically organize groups of steps into a hierarchy.
- **Attribute No. 10 (Automatic determination of program architecture):** It has the ability to automatically determine whether to employ subroutines, iterations, loops, recursions, and internal storage, and the number of arguments possessed by each subroutine, iteration, loop, and recursion.
- **Attribute No. 11 (Wide range of programming constructs):** It has the ability to implement analogs of the programming constructs that human computer programmers find useful, including macros, libraries, typing, pointers, conditional operations, logical functions, integer functions, floating-point functions, complex-valued functions, multiple inputs, multiple outputs, and machine code instructions.
- **Attribute No. 12 (Well-defined):** It operates in a well-defined way. It unmistakably distinguishes between what the user must provide and what the system delivers.
- **Attribute No. 13 (Problem-independent):** It is problem-independent in the sense that the user does not have to modify the system's executable steps for each new problem.
- **Attribute No. 14 (Wide applicability):** It produces a satisfactory solution to a wide variety of problems from many different fields.
- **Attribute No. 15 (Scalability):** It scales well to larger versions of the same problem.
- **Attribute No. 16 (Competitive with human-produced results):** It produces results that are competitive with those produced by human programmers, engineers, mathematicians, and designers.

As shown in detail in *Genetic Programming III: Darwinian Invention and Problem Solving* [16], genetic programming currently unconditionally possesses 13 of the 16 attributes that can reasonably be expected of a system for automatically creating computer programs and that genetic programming at least partially possesses the remaining three attributes.

Attribute No. 16 is especially important because it reminds us that the ultimate goal of a system for automatically creating computer programs is to produce useful programs – not merely programs that solve "toy" or "proof of principle" problems. As Samuel [22] said,

> The aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.

There are 14 specific instances reported in *Genetic Programming III* where genetic programming automatically created a computer program that is competitive with a human-produced result.

What do we mean when we say that an automatically created solution to a problem is competitive with a result produced by humans? We are not referring to the fact that a computer can rapidly print ten thousand payroll checks or that a computer can compute π to a million decimal places. As Fogel, Owens, and Walsh [6] said,

> Artificial intelligence is realized only if an inanimate machine can solve problems ... not because of the machine's sheer speed and accuracy, but because it can discover for itself new techniques for solving the problem at hand.

We think it is fair to say that an automatically created result is competitive with one produced by human engineers, designers, mathematicians, or programmers if it satisfies any of the following eight criteria (or any other similarly stringent criterion):

(A) The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.

(B) The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed journal.

(C) The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.

(D) The result is publishable in its own right as a new scientific result (independent of the fact that the result was mechanically created).

(E) The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.

(F) The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.

(G) The result solves a problem of indisputable difficulty in its field.

(H) The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

Table 1 shows 14 instances reported in *Genetic Programming III* [16],where we claim that genetic programming has produced results that are competitive with those produced by human engineers, designers, mathematicians, or programmers. Each claim is accompanied by the particular criterion that establishes the basis for the claim. The instances in table 1 include classification problems from the field of computational molecular biology, a long-standing problem involving cellular automata, a problem of synthesizing the design of a minimal sorting network, and several problems of synthesizing the design of analog electrical circuits. As can be seen, 10 of the 14 instances in the table involve previously patented inventions. Several of these items are discussed in a companion paper in this volume.

Engineering design offers a practical yardstick for evaluating a method for automatically creating computer programs because the design process is usually viewed as requiring human intelligence. Design is a major activity of practicing engineers. The process of design involves the creation of a complex structure to satisfy user-defined requirements. For example, the design process for analog electrical circuits begins with a high-level description of the circuit's desired behavior and characteristics and entails the creation of both the circuit's topology and the values of each of the circuit's components. The design process typically entails tradeoffs between competing considerations. The design (synthesis) of analog electrical circuits is especially challenging because there is no previously known general automated technique for creating both the topology and sizing of an analog circuit from a high-level statement of the circuit's desired behavior and characteristics.

Table 1 Fourteen instances where genetic programming has produced results that are competitive with human-produced results.

	Claimed instance	Basis for the claim	Section in *Genetic Programming III* [16]
1	Creation of four different algorithms for the transmembrane segment identification problem for proteins	B, E	16.6
2	Creation of a sorting network for seven items using only 16 steps	A, D	21.4.4

3	Rediscovery of recognizable ladder topology for filters	A, F	25.15.1
4	Rediscovery of "*M*-derived half section" and "constant K" filter sections	A, F	26.1.3
5	Rediscovery of the Cauer (elliptic) topology for filters	A, F	27.3.7
6	Automatic decomposition of the problem of synthesizing a crossover filter	A, F	32.3
7	Rediscovery of a recognizable voltage gain stage and a Darlington emitter-follower section of an amplifier and other circuits	A, F	42.3
8	Synthesis of 60 and 96 decibel amplifiers	A, F	45.3
9	Synthesis of analog computational circuits for squaring, cubing, square root, cube root, logarithm, and Gaussian functions	A, D, G	47.5.3
10	Synthesis of a real-time analog circuit for time-optimal control of a robot	G	48.3
11	Synthesis of an electronic thermometer	A, G	49.3
12	Synthesis of a voltage reference circuit	A, G	50.3
13	Creation of a cellular automata rule for the majority classification problem that is better than the Gacs-Kurdyumov-Levin (GKL) rule and better than all other known rules written by humans over the past 20 years	D, E	58.4

14	Creation of motifs that detect the D–E–A-D box family of proteins and the manganese superoxide dismutase family as well as, or slightly better than, the human-written motifs archived in the PROSITE database of protein motifs	C	59.8

There are. of course, additional instances where others have used genetic programming to evolve programs that are competitive with human-produced results. Other examples include Howley's use of genetic programming to control a spacecraft's attitude maneuvers [8]. Two additional noteworthy recent examples are Sean Luke and Lee Spectors' genetically evolved entry at the 1997 Robo Cup competition at the International Joint Conference on Artificial Intelligence in Nagoya, Japan [21] and the genetically evolved entry at the 1998 Robo Cup competition in Paris [1]. Both of these entries consisted of soccer-playing programs that were evolved entirely by genetic programming. The other entries in this annual competition consisted of programs written by teams of human programmers. Most were entirely human-written; however, minor elements of a few of the human-written programs were produced by machine learning. Both of the genetically evolved entries held their own against the human-written programs and placed in the middle of the field (consisting of 16 and 34 total entries, respectively).

Of course, we do not claim that genetic programming is the only possible approach to the challenge of getting computers to solve problems without explicitly programming them. However, we are not aware at this time of any other method of artificial intelligence, machine learning, neural networks, adaptive systems, reinforcement learning, or automated logic that can be said to possess more than a few of the above 16 attributes.

Conspicuously, the above list of 16 attributes does not preordain that formal logic or an explicit knowledge base be the method used to achieve the goal of automatically creating computer programs. Many artificial intelligence researchers and computer scientists unquestioningly assume that formal logic must play a preeminent role in any system for automatically creating computer programs. Similarly, the vast majority of contemporary researchers in artificial intelligence believe that a system for automatically creating computer programs must employ an explicit knowledge base. Indeed, over the past four decades, the field of artificial intelligence has been dominated by the strongly asserted belief that the goal of getting a computer to solve problems automatically can be achieved *only* by means of formal logic inference methods and knowledge. This approach typically entails the selection of a knowledge representation, the acquisition of the knowledge, the codification of the knowledge into a knowledge base, the depositing of the knowledge base into a computer, and the manipulation of the knowledge in the computer using the inference methods of formal logic. As Lenat [20] stated,

> All our experiences in AI research have led us to believe that for automatic programming, the answer lies in *knowledge*, in adding a collection of expert rules which guide code synthesis and transformation. (Emphasis in original).

However, the existence of a strenuously asserted belief for four decades does not, in itself, validate the belief. Moreover, the popularity of a belief does not preclude the possibility that there might be an alternative way of achieving a particular goal. In particular, the popularity over the past four decades of the "logical" and the "cultural" (knowledge-based) approach does not preclude the possiblity that Turing's third way to achieve machine intelligence may prove to be more fruitful.

Genetic programming is different from all other approaches to artificial intelligence, machine learning, neural networks, adaptive systems, reinforcement learning, or automated logic in all (or most) of the following seven ways:

(1) **Representation:** Genetic programming overtly conducts its search for a solution to the given problem in program space.

(2) **Role of point-to-point transformations in the search:** Genetic programming does not conduct its search by transforming a single point in the search space into another single point, but instead transforms a set of points into another set (population) of points.

(3) **Role of hill climbing in the search:** Genetic programming does not rely exclusively on greedy hill climbing to conduct its search, but instead allocates a certain number of trials, in a principled way, to choices that are known to be inferior.

(4) **Role of determinism in the search:** Genetic programming conducts its search probabilistically.

(5) **Role of an explicit knowledge base:** None.

(6) **Role of in the inference methods of formal logic in the search:** None.

(7) **Underpinnings of the technique:** Biologically inspired.

First, consider the issue of representation. Most techniques of artificial intelligence, machine learning, neural networks, adaptive systems, reinforcement learning, or automated logic employ specialized structures in lieu of ordinary computer programs. These surrogate structures include if-then production rules, Horn clauses, decision trees, Bayesian networks, propositional logic, formal grammars, binary decision diagrams, frames, conceptual clusters, concept sets, numerical weight vectors (for neural nets), vectors of numerical coefficients for polynomials or other fixed expressions (for adaptive systems), genetic classifier system rules, fixed tables of values (as in reinforcement learning), or linear chromosome strings (as in the conventional genetic algorithm).

Tellingly, except in unusual situations, the world's several million computer programmers do not use any of these surrogate structures for writing computer programs. Instead, for five decades, human programmers have persisted in writing computer programs that intermix a multiplicity of types of computations (e.g., arithmetic and logical) operating on a multiplicity of types of variables (e.g., integer, floating-point, and Boolean). Programmers have persisted in using internal memory to store the results of intermediate calculations in order to avoid repeating the calculation on each occasion when the result is needed. They have persisted in using iterations and recursions. They have similarly persisted for five decades in organizing useful sequences of operations into reusable groups (subroutines) so that they avoid reinventing the wheel on each occasion when they need a particular sequence of operations. Moreover, they have persisted in passing parameters to subroutines so that they can reuse those subroutines with different instantiations of values. And, they have persisted in organizing their subroutines and main program into hierarchies.

All of the above tools of ordinary computer programming have been in use since the beginning of the era of electronic computers in the l940s. Significantly, none has fallen into disuse by human programmers. Yet, in spite of the manifest utility of these everyday tools of computer programming, these tools are largely absent from existing techniques of automated machine learning, neural networks, artificial intelligence, adaptive systems, reinforcement learning, and automated logic. On one of the relatively rare occasions when one or two of these everyday tools of computer programming is available within the context of one of these automated techniques, it is usually available only in a hobbled and barely recognizable form. In contrast, genetic programming draws on the full arsenal of tools that human programmers have found useful for five decades. It conducts its search for a solution to a problem overtly in the space of computer programs. Our view is that computer programs are the best representation of computer programs. We believe that the search for a solution to the challenge of getting computers to solve problems without explicitly programming them should be conducted in the space of computer programs.

Of course, once one realizes that the search should be conducted in program space, one is immediately faced with the task of finding the desired program in the enormous space of possible programs. As will be seen, genetic programming performs this task of program discovery. It provides a problem-independent way to productively search the space of possible computer programs to find a program that satisfactorily solves the given problem.

Second, another difference between genetic programming and almost every other automated technique concerns the nature of the search conducted in the technique's chosen search space. Almost all of these non-genetic methods employ a point-to-point strategy that transforms a single point in the search space into another single point. Genetic programming is different in that it operates by explicitly cultivating a diverse population of often-inconsistent and often-contradictory approaches to solving the problem. Genetic programming performs a beam search in program space by iteratively transforming one population of candidate computer programs into a new population of programs.

Third, consider the role of hill climbing. When the trajectory through the search space is from one single point to another single point, there is a nearly irresistible temptation to extend the search only by moving to a point that is known to be superior to the current point. Consequently, almost all automated techniques rely exclusively on greedy hill climbing to make the transformation from the current point in the search space to the next point. The temptation to rely on hill climbing is reinforced because many of the toy problems in the literature of the fields of machine learning and artificial intelligence are so simple that they can, in fact, be solved by hill climbing. However, popularity cannot cure the innate tendency of hill climbing to become trapped on a local optimum that is not a global optimum. Interesting and non-trivial problems generally have high-payoff points that are inaccessible to greedy hill climbing. In fact, the existence of points in the search space that are not accessible to hill climbing is a good working definition of non-triviality. The fact that genetic programming does not rely on a point-to-point search strategy helps to liberate it from the myopia of hill climbing. Genetic programming is free to allocate a certain measured number of trials to points that are known to be inferior. This allocation of trials to known-inferior individuals is not motivated by charity, but in the expectation that it will often unearth an unobvious trajectory through the search space leading to points with an ultimately higher payoff. The fact that genetic programming operates from a population enables it to make a small number of adventurous moves while simultaneously pursuing the more immediately gratifying avenues of advance through the search space. Of course, genetic programming is not the only search technique that avoids mere hill climbing. For example, both simulated annealing [11] and genetic algorithms [7] allocate a certain number of trials to inferior points in a similar principled way. However, most of the techniques currently used in the fields of artificial intelligence, machine learning, neural networks, adaptive systems, reinforcement learning, or automated logic are trapped on the local optimum of hill climbing.

Fourth, another difference between genetic programming and almost every other technique of artificial intelligence and machine learning is that genetic programming conducts a probabilistic search. Again, genetic programming is not unique in this respect. For example, simulated annealing and genetic algorithms are also probabilistic. However, most existing techniques in the fields of artificial intelligence and machine learning are deterministic.

Fifth, consider the role of a knowledge base in the pursuit of the goal of automatically creating computer programs. In genetic programming, there is no explicit knowledge base. While there are numerous optional ways to incorporate domain knowledge into a run of genetic programming, genetic programming does not require (or usually use) an explicit knowledge base to guide its search.

Sixth, consider the role of the inference methods of formal logic. Many computer scientists unquestioningly assume that every problem-solving technique must be logically sound, deterministic, logically consistent, and parsimonious. Accordingly, most conventional methods of artificial intelligence and machine learning possess these characteristics. However, logic does not govern two of the most important types of complex problem solving processes, namely the invention process performed by creative humans and the evolutionary process occurring in nature.

A new idea that can be logically deduced from facts that are known in a field, using transformations that are known in a field, is not considered to be an invention. There must be what the patent law refers to as an "illogical step" (i.e., an unjustified step) to distinguish a putative invention from that which is readily deducible from that which is already known. Humans supply the critical ingredient of “illogic” to the invention process. Interestingly, everyday usage parallels the patent law concerning inventiveness: People who mechanically apply existing facts in well-known ways are summarily dismissed as being uncreative. Logical thinking is unquestionably useful for many purposes. It usually plays an important role in setting the stage for an invention. But, at the end of the day, logical thinking is the antithesis of invention and creativity.

Recalling his invention in 1927 of the negative feedback amplifier, Harold S. Black of Bell Laboratories [5] said,

> Then came the morning of Tuesday, August 2, 1927, when the concept of the negative feedback amplifier came to me in a flash while I was crossing the Hudson River on the Lackawanna Ferry, on my way to work. For more than 50 years, I have pondered how and why the idea came, and I can't say any more today than I could that morning. All I know is that

> after several years of hard work on the problem, I suddenly realized that if I fed the amplifier output back to the input, in reverse phase, and kept the device from oscillating (singing, as we called it then), I would have exactly what I wanted: a means of canceling out the distortion of the output. I opened my morning newspaper and on a page of *The New York Times* I sketched a simple canonical diagram of a negative feedback amplifier plus the equations for the amplification with feedback.

Of course, inventors are not oblivious to logic and knowledge. They do not thrash around using blind random search. Black did not try to construct the negative feedback amplifier from neon bulbs or doorbells. Instead, "several years of hard work on the problem" set the stage and brought his thinking into the proximity of a solution. Then, at the critical moment, Black made his “illogical” leap. This unjustified leap constituted the invention.

The design of complex entities by the evolutionary process in nature is another important type of problem-solving that is not governed by logic. In nature, solutions to design problems are discovered by the probabilistic process of evolution and natural selection. There is nothing logical about this process. Indeed, inconsistent and contradictory alternatives abound. In fact, such genetic diversity is necessary for the evolutionary process to succeed. Significantly, the solutions evolved by evolution and natural selection almost always differ from those created by conventional methods of artificial intelligence and machine learning in one very important respect. Evolved solutions are not brittle; they are usually able to easily grapple with the perpetual novelty of real environments.

Similarly, genetic programming is not guided by the inference methods of formal logic in its search for a computer program to solve a given problem. When the goal is the automatic creation of computer programs, all of our experience has led us to conclude that the non-logical approach used in the invention process and in natural evolution are far more fruitful than the logic-driven and knowledge-based principles of conventional artificial intelligence. In short, "logic considered harmful."

Seventh, the biological metaphor underlying genetic programming is very different from the underpinnings of all other techniques that have previously been tried in pursuit of the goal of automatically creating computer programs. Many computer scientists and mathematicians are baffled by the suggestion biology might be relevant to their fields. In contrast, we do not view biology as an unlikely well from which to draw a solution to the challenge of getting a computer to solve a problem without explicitly programming it. Quite the contrary – we view biology as a most likely source. Indeed, genetic programming is based on the only method that has ever produced intelligence – the time-tested method of evolution and natural selection. As Stanislaw Ulam said in his l976 autobiography [25],

> [Ask] not what mathematics can do for biology, but what biology can do for mathematics.

REFERENCES

[1] Andre, David and Teller, Astro. 1998. Evolving team Darwin United. In Asada, Minoru (editor). *RoboCup-98: Robot Soccer World Cup II*. Lecture Notes in Computer Science. Berlin: Springer Verlag. In press.

[2] Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

[3] Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. 1998. *Genetic Programming – An Introduction*. San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt.

[4] Banzhaf, Wolfgang, Poli, Riccardo, Schoenauer, Marc, and Fogarty, Terence C. 1998. *Genetic Programming: First European Workshop. EuroGP'98. Paris, France, April 1998 Proceedings. Paris, France. April l998*. Lecture Notes in Computer Science. Volume 1391. Berlin, Germany: Springer-Verlag.

[5] Black, Harold S. 1977. Inventing the negative feedback amplifier. *IEEE Spectrum*. December 1977. Pages 55 – 60.

[6] Fogel, Lawrence J., Owens, Alvin J., and Walsh, Michael. J. 1966. *Artificial Intelligence through Simulated Evolution*. New York: John Wiley.

[7] Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

[8] Howley, Brian. 1996. Genetic programming of near-minimum-time spacecraft attitude maneuvers. In Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference, July 28-31, 1996, Stanford University*. Cambridge, MA: MIT Press. Pages 98–106.

[9] Ince, D. C. (editor). 1992. *Mechanical Intelligence: Collected Works of A. M. Turing*. Amsterdam: North Holland.

[10] Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

[11] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science* 220, pages 671-680.

[12] Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: MIT Press.

[13] Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: MIT Press.

[14] Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: MIT Press.

[15] Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick L. (editors). *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22-25, 1998, University of Wisconsin, Madison, Wisconsin*. San Francisco, CA: Morgan Kaufmann.

[16] Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III*: *Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

[17] Koza, John R., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference* San Francisco, CA: Morgan Kaufmann.

[18] Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

[19] Koza, John R., and Rice, James P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: MIT Press.

[20] Lenat, Douglas B. l983. The role of heuristics in learning by discovery: Three case studies. In Michalski, Ryszard S., Carbonell, Jaime G., and Mitchell, Tom M. (editors) *Machine Learning: An Artificial Intelligence Approach, Volume I*. Los Altos, CA: Morgan Kaufmann. Pages 243-306.

[21] Luke, Sean and Spector, Lee. 1998. Genetic programming produced competitive soccer softbot teams for RoboCup97. In Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick. (editors). *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22-25, 1998, University of Wisconsin, Madison, Wisconsin*. San Francisco, CA: Morgan Kaufmann. Pages 214 – 222.

[22] Samuel, Arthur L. 1983. AI: Where it has been and where it is going. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann. Pages 1152 – 1157.

[23] Spector, Lee, Langdon, William B., O'Reilly, Una-May, and Angeline, Peter (editors). 1999. *Advances in Genetic Programming 3*. Cambridge, MA: The MIT Press.

[24] Turing, Alan M. 1950. Computing machinery and intelligence. *Mind*. 59(236) 433 – 460. Reprinted in Ince, D. C. (editor). 1992. *Mechanical Intelligence: Collected Works*

of A. M. Turing. Amsterdam: North Holland. Pages 133 – 160.

[25] Ulam, Stanislaw M. 1991. *Adventures of a Mathematician*. Berkeley, CA: University of California Press.

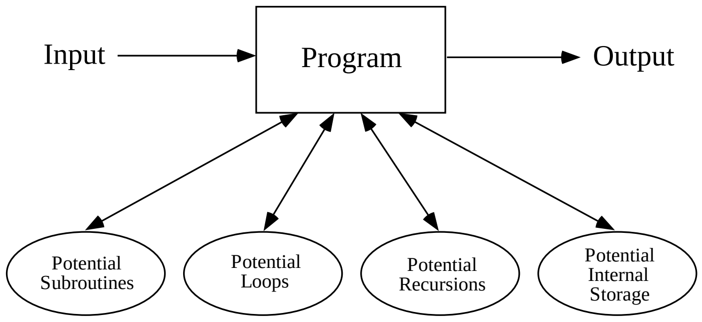

Fig. 1 A computer program.