Available online at www.sciencedirect.com
SCIENCE DIRECT®
Information Sciences xxx (2005) xxx–xxx
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

Immune programming

Petr Musilek *, Adriel Lau, Marek Reformat, Loren Wyard-Scott

Department of Electrical and Computer Engineering, W2-030 ECERF, University of Alberta, Edmonton, Alberta, Canada T6G 2V4

Received 22 June 2004; received in revised form 2 March 2005; accepted 4 March 2005

Abstract

This paper describes 'Immune Programming', a paradigm in the field of evolutionary computing taking its inspiration from principles of the vertebrate immune system. These principles are used to derive stack-based computer programs to solve a wide range of problems.

An antigen is used to represent the programming problem to be addressed and may be provided in closed form or as an input/output mapping. An antibody set (a repertoire), wherein each member represents a candidate solution, is generated at random from a gene library representing computer instructions. Affinity, the fit of an antibody (a solution candidate) to the antigen (the problem), is analogous to shape-complementarity evident in biological systems. This measure is used to determine both the fate of individual antibodies, and whether or not the algorithm has successfully completed.

When a repertoire has not yielded affinity relating algorithm completion, individual antibodies are replaced, cloned, or hypermutated. Replacement occurs according to a replacement probability and yields an entirely new randomly-generated solution candidate when invoked. This randomness (and that of the initial repertoire) provides diversity sufficient to address a wide range of problems. The chance of antibody cloning,

* Corresponding author. Tel.: +1 780 492 5368; fax: +1 780 492 1811.
E-mail address: musilek@ece.ualberta.ca (P. Musilek).

doi:10.1016/j.ins.2005.03.009

wherein a verbatim copy is placed in the new repertoire, occurs proportionally to its affinity and according to a cloning probability. The chances of an effective (high-affinity) antibody being cloned is high, analogous to replication of effective pathogen-fighting antibodies in biological systems. Hypermutation, wherein probability-based replacement of the gene components within an antibody occurs, is also performed on high-affinity entities. However, the extent of mutation is inversely proportional to the antigenic affinity. The effectiveness of this process lies in the supposition that a candidate showing promise is likely similar to the ideal solution.

This paper describes the paradigm in detail along with the underlying immune theories and their computational models. A set of sample problems are defined and solved using the algorithm, demonstrating its effectiveness and excellent convergent qualities. Further, the speed of convergence with respect to repertoire size limitations and probability parameters is explored and compared to stack-based genetic programming algorithms.

Keywords: Evolutionary computing; Genetic programming; Immune programming; Artificial immune system; Clonal selection

1. Introduction

Having computers automatically solve problems without being explicitly programmed to do so is a central goal of Artificial Intelligence [21]. With regard to problem-solving, Fogel et al. [9] argues that methods can be sought by "mechanizing the scientific method in an algorithmic formulation so that a machine may carry out the procedure and similarly [to the scientific method] gain knowledge about its environment and adapt its behavior to meet goals". The area corresponding to this paradigm is known as evolutionary computation (EC) [8,10] and encompasses a broad range of methods such as evolution strategies (ES), evolutionary programming (EP), genetic algorithms (GA), and genetic programming (GP). Refer to [10] for a comprehensive account of these methods and their historical foundations. Evolutionary techniques can be viewed as search or optimization techniques [24] based on the principles of natural evolution. Genetic programming [19] is of special importance in the context of this paper, as it can provide solutions to problems in the form of computer programs.

There has recently been growing interest in the use of methods inspired by the immune system and its principles and mechanisms [5]. There are several analogies between Artificial Immune Systems (AIS) and EC found at different levels: both frameworks are inspired by biological systems, both employ some form of evolutionary principles, and both can be used in largely overlapping application domains. There are, however, significant differences that make AIS a separate area of research. These systems have already been applied to

numerous types of problems such as computer security, data analysis, clustering, pattern matching and parametric optimization [7]. However, an immune parallel to genetic programming has not yet been proposed: there has been no attempt to use principles of immunity to automatically create computer programs. Immune programming (IP), introduced in this paper, is a novel paradigm combining the program-like representation of solutions to problems with the principles and theories of the immune system. IP is an extension of immune algorithms, particularly the clonal selection algorithm [1,2], in AIS. However, IP is not limited to finding an optimized solution for a specific problem as in immune algorithms; it is a domain-independent approach in which solutions (computer programs) are generated that can, in turn, solve an entire class of similar problems. To paraphrase John Koza [20], a pioneer in the field of genetic programming, ''[Immune programming] is a systematic method for getting computers to automatically solve a problem''.

This paper is organized in six sections. The basic concepts and mechanisms of immunity that form the basis of the new approach are described in Section 2. A brief overview of current research related to the use of artificial immune systems for program generation is provided in Section 3. Principles of IP are outlined in Section 4, along with a detailed description of its implementation and experimental results demonstrating its enormous potential. Sensitivity studies and a comparison of IP and GP are conducted in Section 5. Finally, Section 6 presents main conclusions and outlines possible directions for future work.

2. Immune systems

The vertebrate immune system (IS) is one of the most intricate bodily systems and its complexity is sometimes compared to that of the human brain. With advances in biology and molecular genetics, comprehension of how the immune system behaves is increasing very rapidly. Knowledge of immune system functioning has revealed several of its main operative mechanisms. These mechanisms are very interesting not only from a biological standpoint, but also from a computational perspective. Similar to the way the nervous system inspired the development of artificial neural networks, the immune system has led to the emergence of artificial immune systems as a computational intelligence (CI) paradigm.

2.1. Biological immune systems

The immune system of vertebrates is composed of a great variety of molecules, cells, and organs spread throughout the body. There is no central organ controlling the immune system: various distributed elements perform complementary tasks. The main role of the immune system is to protect the organism

against disease-causing cells called *pathogens* and to eliminate malfunctioning cells. This is accomplished by *recognition*, wherein the organism is searched for these elements, followed by immune action (blocking or elimination of the disease-causing agent, for example). In addition to recognizing pathogens and malfunctioning cells, the immune system is also able to recognize the organism's own (properly functioning) cells and tissues to prevent their inadvertent destruction. All elements recognizable by the immune system are called *antigens*: pathogens, malfunctioning cells, and healthy cells. The native cells that originally belong to the organism and are harmless to its functioning are termed *self* (or self antigens), while the disease-causing elements are named *non-self* (or non-self antigens). The immune system, thus, has to be capable of distinguishing between what is self from what is non-self. This process is termed *self/non-self* discrimination.

When an antigen has been encountered and identified as non-self, the immune system initiates a response to deactivate the pathogen. Although this process is sufficient to protect the organism against certain pathogens, antigen recognition and elimination is not enough. In order to be effective in reacting to new pathogens and to improve response to pathogens already encountered, the immune system is endowed with *memory* and the ability to *learn*. The mechanisms of identification, memory, and learning are facilitated through the processes of pattern recognition, clonal selection, negative selection, and affinity maturation. These concepts are further explained in the following subsections.

2.1.1. Pattern recognition in immune system

Pattern recognition is carried out by specific components of the immune system that are produced in the bone marrow. These components are white blood cells (lymphocytes) of two types: B-cells (B-lymphocytes) produced and developed within the bone marrow, and T-cells (T-lymphocytes) that are produced in the bone marrow and then migrate to the thymus for further development. There are other elements in the immune system, such as antigen-presenting cells, natural killer cells, message molecules, and others. These additional components serve various auxiliary functions and their treatment is beyond the scope of this paper.

Both B-cells and T-cells have *receptors* present on their surfaces that are responsible for recognizing antigenic patterns. However, B- and T-cells detect different features of pathogens [32] and function quite differently: B-cells can recognize isolated antigens from outside the antigen cell, while T-cells operate on the antigen-cell complex and can recognize parts of the complex presented by organic molecules, see Fig. 1. There are various mechanism of interaction between the lymphocytes and antigens as well as among different types of lymphocytes. An important interaction is production of *antibodies* by B-cells in response to a specific antigen. Antibodies are capable of further recognition and binding to that type of antigen. However they also directly participate in deac-

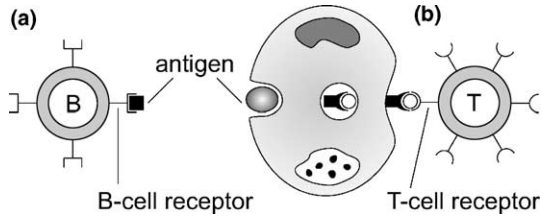

Fig. 1. Pattern recognition in an immune system: (a) B-cell recognizing an isolated antigen ■, (b) T-cell recognizing an antigen within an antigen/cell complex through part of this complex ○.

tivation of pathogens by tagging them, by activating responses of other parts of the IS, and by neutralizing toxins entering the organism.

Receptors present on the surface of T-cells are called T-cell receptors (TCR). T-cells serve auxiliary functions by activating B-cells to promote their growth and differentiation into an antibody-secreting state [32]. Other types of T-cells, called killer T-cells, eliminate intracellular pathogens presented by B-cells. Due to their prominent role in the pattern recognition process and immune response, the main focus of this paper will be placed on B-cells and antibodies in particular.

The recognition process is based on matching the shape of an antigen with the shape presented by the surface receptors of B-cells and T-cells. In other words, the recognition in the immune system is based on shape complementarity [5]. Referring to Fig. 1, the square shaped antigen, ■, is complementary to the square shaped receptor, ⊏, while the circular shape, ○, within the antigen–cell complex is complementary to the semicircle shaped receptor, ⊂. In reality, the surface shapes to be matched are much more complex. The bind between an antigen and a receptor is often not perfect but still leads to correct recognition. The degree of binding is termed *affinity*—the attraction between an antigen and a receptor cell.

2.1.2. Clonal selection

The process of pathogen deactivation is described in [3]. For the purposes of this paper, it is sufficient to note that the lymphocytes are directly and indirectly involved in the process. Depending upon the extent of the infection, a large number of B-cells and T-cells may be required to handle the infection successfully and effectively. The size of subpopulations of these cells is controlled by a process termed *clonal selection*.

After successful recognition, cells capable of binding with non-self antigens are cloned. This replication causes proliferation—an increase in concentration of lymphocytes available for recognition and deactivation of antigens of a given type. This way, the immune system is capable of reacting to reoccurring antigens with greater speed and efficiency: an effect termed secondary

immunological response. The elements of this subpopulation also undergo mutations resulting in a subpopulation of cells that are slightly different. Due to the high mutation rates, this process is usually called *hypermutation*. The consequence of these variations is twofold. First, the variation provides a generalization ability in that the subpopulation is able to recognize not only the antigen itself but also antigens that are similar. Second, some of the mutated clones may represent an even better match to the given antigen than the original cell, i.e. they may have higher affinity. This whole process of selection and hypermutation is known as the *maturation* of the immune response [27] and is analogous to the natural selection of species [15]. In addition, the cells activated by the antigen with high affinities are selected to become *memory* cells with long life spans. These memory cells are pre-eminent in future responses to identical or similar antigenic patterns.

2.1.3. Negative selection

As mentioned earlier, the immune system must be able to recognize self antigens to prevent their inadvertent destruction. In other words, the organism must be able to discriminate between *self* and *non-self* antigens.

The process of training receptor cells for this discrimination takes place mainly in the *thymus*, an important organ of the immune system located behind the breastbone. The thymus performs a crucial role in the maturation of T-cells by removing cells which recognize self-antigens from the population. This process is termed *negative selection*.

The mechanism of negative selection can be described as follows. The thymus is protected by a blood barrier capable of efficiently excluding non-self antigens from the thymus environment. Thus, the elements found within the thymus are representative of self instead of non-self. In turn, T-cells containing receptors capable of recognizing these (self) antigens are eliminated from the repertoire of T-cells [26] and all T-cells that leave the thymus to circulate throughout the body are tolerant to self in that they do not detect self-antigens.

2.2. Artificial immune systems

AIS can be defined as abstract computational systems inspired by theoretical immunology and observed immune functions, principles, and models, applied to solve problems [4]. Similar to other paradigms of CI, an AIS can be described using a suitable framework [5] specifying its representation, evaluation, and adaptation facets. These concepts are further explained in the following subsections.

2.2.1. Representation scheme

AIS is constructed using models of antigens and antibodies analogous to biological IS [5]. Section 2.1.1 noted that the recognition mechanism in bio-

logical IS is based on shape-complementarity between an antigen and a receptor cell. To model the recognition mechanism, a suitable representation scheme is needed to express and manipulate information about the shape of these elements. Such a scheme was proposed along with the concept of *shape–space* by Perelson and Oster [28]. Shape–space, S, allows characterization of the antigens and receptor cells, and quantitative description of their interactions. The shape–space model of recognition in AIS is illustrated in Fig. 2.

The generalized shape of an element, either an antigen or an antibody, can be represented as an attribute string m [5]. The attribute string contains values of the individual coordinates of the shape in the shape–space S, i.e., $m = \langle m_1, m_2, \ldots, m_L \rangle \in S^L$. S is generally defined on the set of real numbers $S^L \subseteq \mathbb{R}^L$, with special cases leading to attribute strings in integer, $m \in \mathbb{Z}^L$, and binary, $m \in \{0,1\}^L$, forms. There is an obvious analogy with the representation schemes used in EC where the term "chromosome" is used instead of "attribute string". Indeed, other forms of representation used in EC are applicable to AIS (e.g., real–valued and permutation representations). The choice of the most suitable representation is driven by the problem at hand.

2.2.2. Evaluation

The interaction between an antigen and an antibody can be described as the degree of their binding in terms of affinity. As illustrated in Fig. 2, a complete match is not necessary for the two elements to bind: the region $L - 1$ of the antigen is not complementary to that of the antibody. As a result, the two elements can still bind, however the affinity of the bind will be lower than the highest possible affinity. In other words, the affinity of an antigen–antibody pair is related to their distance in the shape space S and can be estimated using any distance measure between the two attribute strings [5]. The distance between an antigen, Ag, and an antibody, Ab, can be defined, for example, using a general class of Minkowski distance measures

$$D_{\mathrm{M}}(\mathrm{Ag}, \mathrm{Ab}) = \sqrt[p]{\sum_{i=1}^{L} |\mathrm{Ag}_i - \mathrm{Ab}_i|^p}. \tag{1}$$

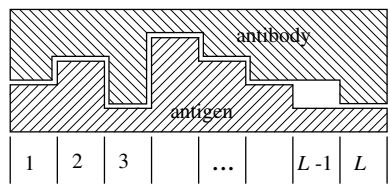

Fig. 2. Antigen recognition in L-dimensional shape–space.

By varying the value of the parameter p, special cases such as Hamming distance, ($p = 1$), and Euclidean distance, ($p = 2$), can be obtained. The choice of suitable distance measure depends on the problem to be solved and, in turn, on the representation scheme used.

2.2.3. Adaptation

The procedures of adaptation govern the evolution of the behavior of an AIS. The algorithms for adaptation can be classified as population-based and *network-based* algorithms [5]. Only population-based algorithms are relevant to the development of IP and thus only this class of immune algorithms will be described in this paper. For a comprehensive treatment of immune network theory, please refer to [16]. The population-based algorithms can be further categorized as clonal selection and negative selection algorithms, corresponding to simulation of B-cell and T-cell behavior, respectively. Systems simulating B-cells are oriented towards pattern recognition problems, while T-cell systems are used for anomaly detection.

The **Clonal Selection Algorithm** provides a model of the clonal selection process present in biological IS. The algorithm is summarized as follows:

1. n candidate solutions are generated and evaluated using a suitable affinity measure.
2. n attribute strings with highest affinity are selected to proliferate by cloning; the cloning rate of each immune cell is proportional to its affinity.
3. Newly generated clones are subjected to hypermutation; the mutation rate of each immune cell is inversely proportional to its affinity.

This algorithm results in change of antibody concentrations favoring those with high affinity, and added diversity through the process of hypermutation.

Some authors [13] have argued that a GA without crossover is a reasonable model of clonal selection. However, the standard GA does not account for important properties such as affinity-proportional reproduction and hypermutation inversely proportional to affinity.

The **Negative Selection Algorithm** provides a model of the self/non-self discrimination capability learned in the thymus of biological IS. This algorithm [12] has been proposed with application focused on the problem of anomaly detection, such as computer and network intrusion detection [11], image inspection and segmentation, and hardware fault tolerance. It can be summarized as follows:

1. n candidate detectors are generated.
2. Each candidate detector, C_i, $i = 1,2, \ldots ,n$, is compared to a set of protected elements, PE.

3. If a match occurs between C_i and PE, the detector is discarded.
4. Otherwise (if a match does not occur), C_i is stored in the detector set D.

This algorithm produces a set of detectors capable to recognize non-self patterns. The action following the recognition varies according to the problem under consideration.

3. Related research

Although AISs have been actively studied as of late [6,14,22,23], [25,31], direct use of immune principles for generation of computer programs has not been reported. However, several works related to the topic are described in the following text.

Nikolaev et al. introduced an immune version of GP (iGP) [25]. In iGP, the progressive search for programs is controlled by a dynamic fitness function based on an analogy of the immune network dynamics. The method is applied to a machine-learning task and a time-series prediction problem. The results presented demonstrate that the immune version attains fitter programs while maintaining higher population diversity when applied to these problems.

McCoy and Devaralan [23] use a negative selection algorithm to solve an aerial image segmentation problem. The procedure generates a non-self detector set used to identify a target class. The results of this system are compared to those obtained by GP and it is concluded that although both methods are suitable for parallel implementation, the immune version is expected to adapt more readily to changes in its sample data by being dynamically adjustable. It is also argued that finding a population of detectors that covers non-self features is easier than finding a single optimal detector, since there are many ways to arrive at an acceptable solution.

Both applications described use the general concepts of artificial immunity and the principle of negative selection to modify/improve existing evolutionary techniques [6]. A paper by Gao [14] "Fast Immunized Evolutionary Programming" describes the modification to the selection in traditional evolutionary programming by principles of artificial immune system. In contrast, the evolutionary technique proposed in this paper is developed entirely from the principles of artificial immunity.

4. Immune programming

In this section, the proposed paradigm of Immune Programming is described in detail. After introducing the computing architecture chosen for execution of the generated programs, the algorithm of IP is summarized and its

individual steps are explained. A running example is used to illustrate the concepts.

4.1. Computing architecture

The choice of computing architecture on which automatically generated programs are executed has important implications regarding the efficiency of the execution and portability of the programs. Although the majority of current approaches use tree or *S*-expression based structures [30], there have been several attempts to employ stack-based frameworks [29,17].

Stack-based machines have a small size, low system complexity, high system performance, and good performance consistency under varying conditions [18]. In addition to the possibility of direct hardware implementation of stack machines, there are available stack-based virtual machines with portable, high performance interpretation techniques [30]. These reasons led to the choice of stack-based computing as the architecture on which the IP generated programs are executed and evaluated. Following this choice, principles of IP are illustrated using stack-based examples in this paper. IP systems could be implemented, however, using alternate architectures as well.

A computing architecture is functionally described by an *instruction set*. The instruction set provides a detailed list of the *operations* that the machine is capable of processing, and description of the *types*, *locations*, and *access methods* for operands. In a stack-based machine all operands are located and accessed on the stack, and not directly defined in the instructions.

Programs described using tree or *S*-expression based notation can be equivalently expressed using a stack instruction set, as illustrated in Fig. 3.

4.2. The algorithm

The IP algorithm is developed using the immune principles of clonal selection and replacement of low affinity antibodies. The algorithm is briefly described as follows:

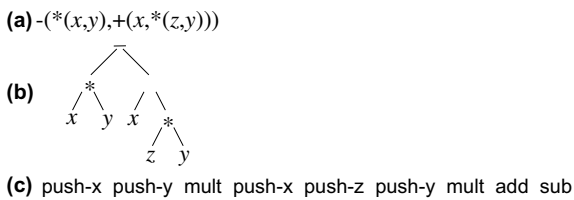

Fig. 3. Representation of arithmetic expression $xy-(x+zy)$ under different frameworks: (a) *S*-expression, (b) tree representation, (c) stack instruction string.

1. **Initialization**. An initial repertoire (population), AB, of n antibodies, Ab_i, $i = 1, \ldots, n$, is generated. The generation counter is set to $G = 1$.
2. **Evaluation**. An antigen, Ag, representing the problem to be solved, is presented. Ag is compared to all antibodies $\mathrm{Ab}_i \in \mathrm{AB}$ and their affinity, f_i, with respect to the antigen is determined.
3. **Replacement**. With a certain probability, P_r, a new antibody is generated and placed into the new repertoire. This way, low affinity antibodies are implicitly replaced. The parameter P_r is the probability of replacement.
4. **Cloning**. If a new antibody has not been generated, an antibody, Ab_i, is drawn from the current repertoire with a probability directly proportional to its antigenic affinity. With a probability, P_c, this antibody is cloned and placed in the new repertoire. The parameter P_c is termed probability of cloning.
5. **Hypermutation**. If the high-affinity antibody selected in the previous step has not been cloned, it is submitted for hypermutation with a probability inversely proportional to its antigenic affinity. If the antibody is selected for hypermutation, each component of its attribute string is mutated with probability of mutation P_m.
6. **Iteration–repertoire**. Steps 3–5 are repeated until a new repertoire AB′ of size n is constructed.
7. **Iteration–algorithm**. The generation counter is incremented, $G = G + 1$, and the new repertoire is submitted to step 2, evaluation. The process continues iteratively until a stopping criteria is met.

4.2.1. Initialization

The initial repertoire of antibodies is generated by concatenating components randomly selected from gene libraries [5]. In the IP algorithm these libraries contain instructions available in the instruction set of the underlying computing architecture. For example, a simple instruction set for a stack-based machine can contain the instructions described in Table 1.

The instructions from the instruction set are encoded for representation in the attribute string m. As indicated in Table 1, non-negative integers have been

Table 1
Instruction set

Instruction	Code	Description
`nop`	0	No operation
`dup`	1	Duplicate the top of the stack ($x \Rightarrow x\ x$)
`swap`	2	Swap the top two elements of the stack ($x\ y \Rightarrow y\ x$)
`mult`	3	Multiply the top two elements of the stack ($2\,3 \Rightarrow 6$)
`add`	4	Add the top two elements of the stack ($2\,3 \Rightarrow 5$)
`over`	5	Duplicate the second item on the stack ($x\ y \Rightarrow y\ x\ y$)

selected to represent the instructions in the current implementation of the IP algorithm, leading to an attribute string $m \in \mathbb{Z}^L$. The greatest code number has value $r - 1$, where r is the number of instructions in the instruction set. Alternate representations are possible, including symbols, real numbers, etc. Integers are chosen for code efficiency.

Theoretically, the length of the program is restricted only by limits of the underlying hardware. In practical implementation, the length of the programs can be variable or the length can be fixed to an arbitrary value corresponding to the expected complexity of the problem solution. In the latter case, programs of varying length can be obtained by padding with the `nop` instruction. Following the notation introduced in Table 1, the attribute string $m = \langle 1, 3 \rangle$ represents the simple program

```
dup mult
```

Assuming a stack with value x on top, this program represents a solution to the expression x^2.

The size of the instruction set, r and the program length L, dictate the size of the set of all antibodies available for encoding solutions (the available repertoire). This size, r^L, needs to be large enough to ensure sufficient antibody diversity. However, even with a small instruction set and limited program length, the available repertoire is quite large. For example, assuming the instruction set with $r = 6$ operations and program length $L = 6$, the available repertoire is equal to $r^L = 6^6 = 46{,}656$ variations.

At this point, an example is introduced to illustrate the operation of the algorithm. As mentioned previously, a repertoire, AB, of n integer attribute strings, m, is randomly generated during initialization. Let us assume that $n = 100$ and program size is limited to $L = 6$. Let us consider the following five antibodies from the entire repertoire

$$\begin{aligned}
\mathrm{Ab}_1 &= \langle 1, 3, 2, 5, 1, 3 \rangle, \\
\mathrm{Ab}_2 &= \langle 4, 1, 2, 2, 3, 0 \rangle, \\
\mathrm{Ab}_3 &= \langle 0, 2, 0, 5, 5, 5 \rangle, \\
\mathrm{Ab}_4 &= \langle 3, 3, 4, 3, 3, 3 \rangle, \\
\mathrm{Ab}_5 &= \langle 5, 4, 4, 4, 1, 2 \rangle.
\end{aligned}$$

These antibodies, themselves, are the strings corresponding to possible program solutions.

4.2.2. Evaluation

At the beginning of each iteration, an antigen, Ag, representing the problem to be solved, is presented. This antigen can take different forms depending on the way the problem is described. For example, it can contain pairs of input–output numerical values representing a desired mapping for which a program is

to be found. Alternatively, it can take the form of an arithmetic expression for which a code implementation is to be found. Let us assume that the second approach is used and that the antigen takes the form:

$$\mathrm{Ag} = x^2 + 2xy + y^2.$$

This antigen is compared to all antibodies in the repertoire and their affinity with respect to the antigen is determined. For this purpose, the antibodies must be first decoded to the program space, PG, and then evaluated for certain values from the input space.

The process of decoding is rather trivial and involves substitution of the instruction names for the corresponding codes. Assuming the instructions set from Table 1 and antibodies from the example introduced earlier, the corresponding programs are

$$\mathrm{Pg}_1 = \texttt{dup mult swap over dup mult}$$
$$\mathrm{Pg}_2 = \texttt{add dup swap swap mult nop}$$
$$\mathrm{Pg}_3 = \texttt{nop swap nop over over over}$$
$$\mathrm{Pg}_4 = \texttt{mult mult add mult mult mult}$$
$$\mathrm{Pg}_5 = \texttt{over add add add dup swap}$$

To evaluate the affinity of the antibodies, particular values of variable(s) have to be placed on the stack and the programs have to be executed. Because the problem is described in a symbolic form, no numerical argument values are explicitly prescribed and they must be generated. For the antigen used as an example, this corresponds to generation of x and y values. This leads to two potential problems. First, if only a single set of arguments is used, there is a chance that the program may provide a correct result for that specific case, but not correspond to the relation implied by the problem description antigen, Ag. For this reason, multiple sets of test values have to be generated. The second problem stems from the inability to automatically generate arguments that would effectively cover the entire domain on which the solution should be evaluated. This problem can be neglected for simple antigenic forms but must be considered for more complex cases. Alternatively, this problem is avoided if the domain is specified explicitly along with the description of the antigen.

The arguments for the example program evaluation are generated randomly. The values of these arguments are restricted to 1-byte integer values, [0, ... ,255]. This restriction on the values, along with their actual representation using long integer (4-byte) containers, minimizes the chance of stack overflow. For this example, five sets of values x and y are generated to execute each antibody and to compare the execution results to the antigens behavior. The randomly generated values are $x = [157, 202, 235, 188, 45]$ and $y = [103, 239, 234, 105, 228]$. The results of executing the antibodies Ab_i are

Pg_1: [607573201, 1664966416, 3049800625, 1249198336, 4100625],
Pg_2: [67600, 194481, 219961, 85849, 74529],
Pg_3: [157, 202, 235, 188, 45],
Pg_4: N/A,
Pg_5: N/A.

Symbol N/A indicates that program failed to return a result. The antigen Ag yields the values

[67600, 194481, 219961, 85849, 74529].

Affinity can be determined by calculating the distance (for example, Euclidean distance (1), $p = 2$) of the results of the generated programs, Pg_i, from the expected results defined by the antigen, Ag. Using this approach, the resulting range of possible affinity values is very large and direct evaluation of programs that do not execute correctly (e.g., programs Pg_4 and Pg_5) is not possible. To overcome these difficulties, an alternate approach based upon a three-tiered measure has been developed to consider three important properties of the generated programs:

Executability. If a program executes correctly (no program error, such as the stack pointer pointing to an invalid memory address, is encountered), it is assigned a score T_1.
Completeness. If a program execution returns only a single value, score T_2 is added.
Correctness. If program execution yields result identical to the expected result obtained by evaluating the antigen with the same set of arguments, score T_3 is added.

The tier scores are defined such that

$$T_1 < T_2 < T_3. \tag{2}$$

This states that correctness is more important than program completeness, which is in turn more important than its executability. The tier scores are evaluated in sequence so only executable programs are evaluated for completeness, and only executable and complete programs are evaluated for correctness. Particular values of the scores are not important as long as they satisfy the condition (2) for an arbitrary number of independent evaluations (sets of attributes), c. For example, the scores can be defined as

$$\begin{aligned} T_1 &= 1, \\ T_2 &= c(T_1 + 1), \\ T_3 &= c(T_1 + T_2 + 2), \end{aligned} \tag{3}$$

leading to values $T_2 = 20$ and $T_3 = 230$ for $c = 10$. The overall affinity of each antibody can be expressed as

$$f_i = \sum_{j=1}^{c} T_1(\mathrm{Pg}_i, \mathrm{arg}_j) + T_2(\mathrm{Pg}_i, \mathrm{arg}_j) + T_3(\mathrm{Pg}_i, \mathrm{arg}_j), \tag{4}$$

where arg_j is j-th set of arguments used for program evaluation.

In addition to clear separation of executable, complete, and correct programs, this affinity measure allows explicit determination of the maximum possible affinity needed for calculation of normalized affinity of each individual. The maximum affinity for a given number of independent evaluations c is

$$f^M = c(T_1 + T_2 + T_3). \tag{5}$$

Returning to the running example, the affinities f_i and normalized affinities f_i^N of the generated programs have values $f = [5, 1255, 5, 0, 0]$ and $f^N = [0.004, 1.0, 0.004, 0, 0]$. It can be seen that program Pg_2 has relative affinity $f_2^N = 1$ and thus provides an executable, complete, and correct solution to the problem defined by the antigen, Ag. The sequence of stack states corresponding to the execution of this program is illustrated in Fig. 4.

This program is not optimal since the two `swap` operations following `dup` do not change stack contents and could be omitted. It is desirable to exert selection pressure on the algorithm to favor optimal, or generally shorter, programs. This is accomplished by including a term describing the length of the program when calculating affinity. Eq. (4) thus becomes

$$f_i = \sum_{j=1}^{c} T_1(\mathrm{Pg}_i, \mathrm{arg}_j) + T_2(\mathrm{Pg}_i, \mathrm{arg}_j) + T_3(\mathrm{Pg}_i, \mathrm{arg}_j) - kL_\mathrm{a}, \tag{6}$$

where L_a is the actual length of the program, and k is a constant of proportionality. The actual program length L_a is obtained by discounting all occurrences of the redundant instruction `nop`. The term $-kL_\mathrm{a}$ is considered only for executable, complete, and correct programs to avoid negative values of affinity. For the antibodies $\mathrm{Ab}_1, \ldots \mathrm{Ab}_5$ used as an example, the normalized affinities modified according to (6) have values $f^N = [0.004, 0.98, 0.004, 0, 0]$, considering $k = 1$.

4.2.3. Replacement

After the evaluation phase is complete, the relative affinity of all antibodies $\mathrm{Ab}_i \in \mathrm{AB}$ is known, and the algorithm proceeds with generation of a new

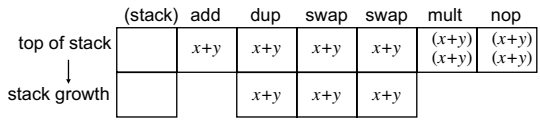

Fig. 4. Sequence of stack states during execution of program Pg_3.

repertoire. This is an iterative process involving three major steps: replacement, cloning, and hypermutation. The goal of the first step, replacement, is to replace low-affinity antibodies present in the current population. As the two subsequent steps are proportionally applied only to antibodies of high affinity, replacement can take place by simply generating a new antibody and placing it directly in the new repertoire. This replacement affects low-affinity antibodies implicitly: these antibodies will not be considered for cloning and mutation in subsequent steps and will not survive to the next repertoire.

The process of replacement is implemented as follows. A random number $\texttt{RAND} \in [0,1]$ is generated and compared to a parameter called probability of replacement, P_r. If $\texttt{RAND} \leqslant P_r$, a new antibody is generated and placed in the new repertoire and the algorithm proceeds to step 6, iteration–repertoire. The process of generating new antibodies is identical to the initialization process described in Section 4.2.1. If $\texttt{RAND} > P_r$, no antibody is placed in the new repertoire and the algorithm proceeds to the next step. Please note that the affinity f_i^N is not explicitly considered.

Besides the implied replacement of low-affinity antibodies, a replacement process implemented this way has two other important consequences. First, computational resources are saved as there is no need to perform comparison of affinity for all antibodies in the current repertoire. This saving is quite significant because a large portion of the current repertoire (about 50%) is typically replaced. Second, for the same value of replacement probability, P_r, a different number of antibodies is actually replaced depending on the average affinity of all antibodies in the current repertoire. If a repertoire contains a large number of high-affinity antibodies, the algorithm produces a new repertoire in only a few iterations and the proportion of antibodies actually replaced corresponds to the probability of replacement. If, on the other hand, a repertoire contains only a small number of high-affinity antibodies, the process of generating new repertoire will take significantly more iterations and the proportion of antibodies actually replaced will be greater.

To illustrate the process of replacement, let us consider an empty new repertoire, $|\mathrm{AB}'| = 0$, and probability of replacement $P_r = 0.5$. A random number is generated, e.g., $\texttt{RAND} = 0.0099$, and compared to the probability of replacement. As $0.0099 > P_r$, a new antibody is randomly generated and placed in the new repertoire. This leads to the new repertoire of size $|\mathrm{AB}'| = 1$ containing a single antibody $\mathrm{Ab}'_1 = \langle 5,2,5,0,2,4 \rangle$. The algorithm then proceeds to step 6 (iteration–repertoire).

4.2.4. Cloning

When creating a new repertoire, if a new antibody has not been generated (in step 3, replacement), an antibody, Ab_i, from the current repertoire, AB,

is considered for cloning. The antibodies are selected in a sequential manner starting from the beginning of the repertoire, $i = 1$.

The selected antibody is first examined for affinity. A random number RAND is generated and compared to the relative affinity of the antibody f_i^N. If $\mathtt{RAND} \leqslant f_i^N$, the antibody is cloned and put into the new generation with probability of cloning P_c. This concludes the current iteration of creating the new repertoire, step 6, and the algorithm returns to step 3 (replacement) unless the new repertoire is complete, $|AB'| = n$, in which case the algorithm proceeds to step 7 (iteration–algorithm). If $\mathtt{RAND} \leqslant f_i^N$ but Ab_i has not been cloned due to the stochastic character of the cloning process, the antibody is submitted to hypermutation.

There are two stochastic aspects of the cloning process. The first aspect, driven by affinity f_i^N, ensures that mainly high-affinity antibodies are considered for cloning. At the same time, low-affinity antibodies can be cloned but with much smaller probability proportional to their affinity. The second aspect limits the proportion of antibodies that are actually cloned to a portion, P_c, of the high-affinity subpopulation.

To illustrate the process of cloning, let us consider the incomplete new repertoire created in the previous iteration. A new random number is generated, e.g., RAND = 0.9501. As $0.9501 > P_r$, a new antibody is not generated and the process of cloning begins. The first antibody in the current population, Ab_1, is selected, and another random number is generated, e.g., RAND = 0.1389. As this number is greater than $f_1^N = 0.004$, the algorithm proceeds back to step 3, replacement, while setting the antibody pointer to 1.

After several iterations, the algorithm arrives at antibody Ab_2. The newly generated random number, e.g., RAND = 0.6038, is less than $f_2^N = 0.98$. Therefore this antibody is considered for cloning. The cloning actually occurs if the next random number is less than the probability of cloning, P_c. Let us assume $P_c = 0.1$ and RAND = 0.0538. In this case cloning is performed and the current antibody, Ab_2, is placed in the new repertoire. Assuming that another new antibody has been generated during the previous iteration(s), this leads to the new repertoire of size $|AB'| = 3$ containing antibodies $Ab'_1 = \langle 5,2,5,0,2,4\rangle$, $Ab'_2 = \langle 3,2,4,1,3,5\rangle$, and $Ab'_3 = \langle 4,1,2,2,3,0\rangle$. Please note that the antibodies Ab'_1 and Ab'_2 are newly generated to replace low affinity antibodies in the current repertoire AB, while Ab'_3 is an exact copy (clone) of the high-affinity antibody $Ab_2 \in AB$.

4.2.5. Hypermutation

If the high-affinity antibody selected in the previous step has not been cloned (due to P_c), it is submitted to the process of hypermutation. This process walks through the attribute string $m = \langle m_1, m_2, \ldots, m_L\rangle \in S^L$ of the antibody, Ab_i, and replaces each attribute $m_j \in m$ with a new randomly-generated value. This replacement is driven by the probability of mutation, P_m, so only a portion of

the attributes is actually replaced. To make the probability of mutation inversely proportional to the affinity of a given antibody, P_m is scaled by its normalized affinity, f_i^N, and bounded over the interval [0, 1]. The resulting probability $\min[(P_m/f_i^N), 1]$ is used for the actual decision to replace a particular attribute, m_j, of the antibody Ab_i.

The hypermutation provides the algorithm with the ability to introduce new material into the repertoire and expands the solution space searched. The inverse proportionality of hypermutation ensures that high-affinity antibodies are disturbed only slightly while low-affinity ones are modified to a high extent. In fact, for affinities less than or equal to the probability of mutation, hypermutation is equivalent to replacement as all components of the attribute string are exchanged with probability $P = 1$.

To illustrate the process of hypermutation, let us assume that there is an incomplete new repertoire of size $|AB'| = 4$ containing the following antibodies: $Ab'_1 = \langle 5, 2, 5, 0, 2, 4\rangle$, $Ab'_2 = \langle 3, 2, 4, 1, 3, 5\rangle$, $Ab_3 = \langle 4, 1, 2, 2, 3, 0\rangle$ and $Ab'_4 = \langle 5, 2, 3, 2, 3, 1\rangle$. Similar to the previous case, the antibodies Ab'_1, Ab'_2 and Ab'_4 are newly generated to replace low affinity antibodies in the current repertoire AB, while Ab'_3 is an exact copy (clone) of the high-affinity antibody $Ab_2 \in AB$.

Let us further assume that in the current iteration replacement does not take place and antibody Ab_2 is again considered for cloning due to its high affinity. This time the random number, e.g., RAND = 0.5161, is greater than the probability of cloning, $P_c = 0.1$, and cloning is not performed. Subsequently, the antibody Ab_2 is submitted for hypermutation. Each component m_j of its attribute string is mutated with probability $P_m/f_i^N = 0.102$ by generating a new number. A sequence of L random numbers is generated, e.g., 0.4565, 0.0185, 0.8214, 0.4447, 0.6154, 0.7919. Only the second number in the sequence is less than 0.102, and therefore only the second component of the attribute string is replaced by a newly generated instruction code, e.g., 5. Subsequently, the new antibody $Ab'_5 = \langle 4, 5, 2, 2, 3, 0\rangle$ is placed in the new repertoire.

4.2.6. Iteration–repertoire

Steps 3–5 (replacement, cloning and hypermutation) are repeated until a complete new repertoire, $|AB'| = n$, has been created. After an antibody, Ab_i, has been selected in step 4 (cloning), its position, i, is marked using a pointer so the following antibody, Ab_{i+1}, can be considered next time. This pointer is set whether or not the i-th antibody has been actually cloned or mutated. After the entire current repertoire has been considered, the pointer is reset to $i = 1$ and the process continues until the new repertoire is complete. In this way high-affinity antibodies can be cloned and/or mutated several times and, consequently, their concentration in the new repertoire increases. At the same time, low-affinity antibodies can be skipped several times and be implicitly replaced by newly generated ones.

4.2.7. Iteration–algorithm

After the new repertoire has been constructed, the generation counter (set to value $G = 1$ during initialization) is incremented, $G = G + 1$. The algorithm then proceeds iteratively through steps 2–6 (evaluation, replacement, cloning, hypermutation, iteration–repertoire) until a stopping criterion is met. The criterion can take various forms: preset number of iterations, fitness threshold, or no fitness improvement, etc. After the criteria is met, the resulting attribute string is presented and can be decoded to provide a stream of instructions for a stack machine to solve the problem described using the antigen, Ag, in step 2.

4.2.8. Summary of IP algorithm

The algorithm of IP is based on the concept of evolving a repertoire of antibodies that encode candidate solutions to a given problem. At the start, candidate solutions to the problem are randomly generated providing an initial repertoire of adequate diversity. The evolution of the repertoire is driven by cloning, hypermutation, and replacement of the antibodies. These processes maintain the diversity of the repertoire and expand the space searched for solutions. The choice of antibodies for these processes is facilitated by affinity-based probabilistic selection: the antibodies with high affinity are selected for cloning or hypermutation while the low-affinity antibodies are replaced.

Affinity evaluation provides a way of rating the quality of solutions provided by the individual antibodies. Based on the affinity values and algorithm parameters, the selection decisions are applied to form a new repertoire. In each iteration one antibody, at most, is placed in the new repertoire through one of the operations of replacement, cloning or mutation. After the new repertoire has been created, the algorithm continues by repeated evaluation of the antibodies and selection for cloning, hypermutation, or replacement. When an antibody of maximal affinity is found, the algorithm stops and presents the final result of the search process. The result is then decoded to the program space and can be used as an implementation of the solution to the specified problem.

The algorithm is summarized in a block diagram depicted in Fig. 5.

4.3. Results

To demonstrate the capabilities of the IP system, several experiments are conducted with antigens in the form of arithmetic expressions. The instruction set shown in Table 1 is considered, and although simple it can be used to solve a variety of nontrivial problems.

To facilitate execution of the IP algorithm, a stack of a depth corresponding to the number of variables in the user-entered expression is created. During the run of the algorithm, values of variables are loaded in alphabetical order

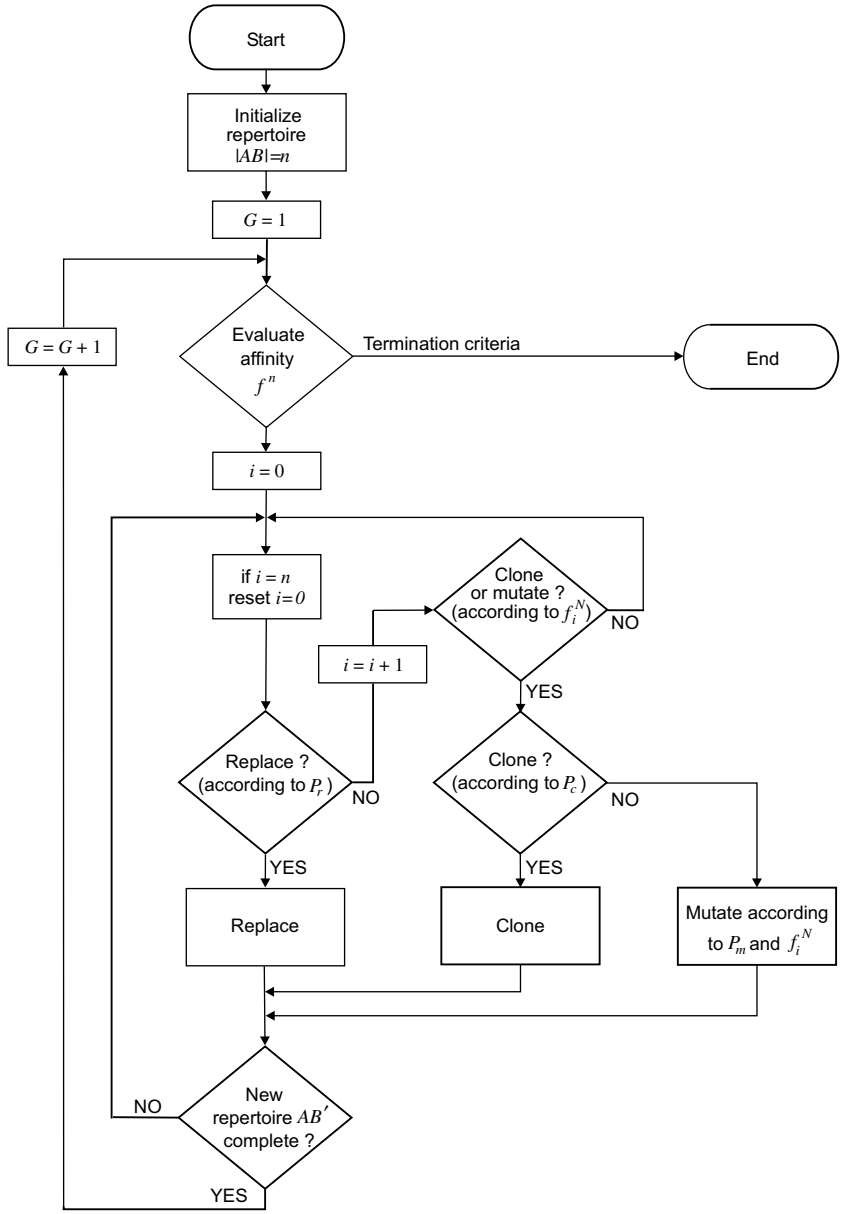

Fig. 5. Flowchart of the IP algorithm.

starting at the top of the stack. The algorithm itself is implemented exactly as described in Section 4.2.

The parameters used for the experiments are $n = 100$, $P_r = 0.5$, $P_c = 0.1$, $P_m = 0.2$. The program size, L, is varied for the individual experiments. Besides the form of the generated programs, the number of generations to arrive at the solutions is recorded. Due to the stochastic nature of the algorithm, 10 independent trials are conducted to avoid bias, and the results are averaged.

4.3.1. High-order operations

The first experiment is designed to examine performance of the IP system to deal with operations of high order. The particular expression considered is x^8. The performance of the system varies according to the program size, L, as indicated in Table 2.

It can be seen that the number of generations to find a solution is greatest for the minimal program length, $L = 6$. This is caused by the iterative nature of solutions to high-order operations. The shortest possible programs require one unique sequence of instructions arranged in a repetitive pattern. Such repetitive structures are more difficult to arrive at using randomly-generated and mutated strings than heterogeneous ones.

The shortest possible program is of the form

```
dup mul dup mul dup mul,
```

corresponding to the expression $x^{2^{2^2}}$.

Longer programs of various forms evolved. Some of these are of the same form as the shortest program, above, but padded with `nop` instructions or other redundant sequences. Other trials produce functionally longer sequences, such as

```
dup mul dup over over mul mul mul,
```

corresponding to the expression $x^2 \cdot x^2 \cdot x^2 \cdot x^2$.

4.3.2. Multiple variables

The second experiment is provided to illustrate the ability of the IP system to handle multiple variables. The particular case considered involves three variables x, y, z in expression $x \cdot y + y^2 + z$. The performance of the system for various values of L is summarized in Table 3.

The dependence of the number of generations to find a solution is reversed and much milder than in the previous case. The heterogeneity of the expression does not dictate repeating patterns in the solution space and such strings are more likely to be generated or arrived at by random modifications.

Table 2
Performance of the IP system: high-order operations

Constrained program size, L	6	7	8	9	10
Number of generations, $\overline{G}$	318.3	58.3	55.2	54.6	33.3

Table 3
Performance of the IP system: multiple variables

Constrained program size, L	4	5	6	7	8	9	10
Number of generations, $\overline{G}$	10.7	14.6	10.9	17.1	15.9	26.8	43.4

The shortest possible programs ($L = 4$) have the form

```
over add mul add,
```

corresponding to the expression $(x + y) \cdot y + z$. Most successful longer programs are of the same form with added redundant operations, changed operand order, or modified order of operators. For instance $y \cdot (x + y) + z$ results in a different stack-based program than $(x + y) \cdot y + z$.

Other trials lead to solutions representing expanded forms of the original expression.

For example

```
over mul swap dup mul add add,
```

corresponds to the expression $x \cdot y + y \cdot y + z$.

4.3.3. Factorization

The goal of the third experiment is to verify whether the IP system is capable of simplifying arithmetic expressions, e.g., by factorization. For instance, the antigen expression $x^2 \cdot y^2$ can be reformulated as $(x \cdot y)^2$. The performance of the system for various values of L is summarized in Table 4. As in the previous case, there is no strong dependence between the number of generations to find a solution and the program length L.

The shortest possible programs ($L = 3$) have the form

```
mul dup mul,
```

corresponding to the expression $(x \cdot y) \cdot (x \cdot y) \equiv (x \cdot y)^2$. Indeed, when the size of solution is constrained, the system is capable of factorization. Longer solutions yield correct results but often without factorization. The two most common solutions are

```
over over mul mul mul,
```

corresponding to the expression $x \cdot y \cdot x \cdot y$, and

```
dup mul over mul mul,
```

corresponding to $x \cdot x \cdot y \cdot y$.

Table 4
Performance of the IP system: factorization

Constrained program size, L	3	4	5	6	7	8	9	10
Number of generations, $\overline{G}$	4.8	4.7	4.5	6.4	6.3	11.4	9.2	12

Table 5
Performance of the IP system: no simplification

Constrained program size, L	10	11	12	13	14	15
Number of generations, $\overline{G}$	488.1	292.3	210.4	165.2	135.3	224

4.3.4. No simplification

The fourth experiment examines the performance of the IP system while solving more complicated expressions that cannot be further simplified. The expression considered is $x^2 + y^2 + x + y$. The performance of the system for program lengths $L = \{10, 11, 12, 13, 14, 15\}$ is summarized in Table 5.

There is a minimum of 10 instructions necessary to solve this expression in a stack-based program. This increased minimal program length causes a significant increase in the number of generations needed to find a solution. A variety of solutions evolve for this expression, primarily differing in the sequence of the instructions. A typical solution is

```
dup dup mul add swap dup dup mul add add,
```

corresponding to $y^2 + y + x^2 + x$. The repeated pattern in the solution (`dup dup mul add`, in this case) leads to a large number of generations for minimal program length.

5. Performance evaluation

To evaluate performance of the IP system, several additional experiments are conducted to study the evolution of affinity and the sensitivity of convergence to algorithm parameters. A comparison of the new system to its main competitor, genetic programming, is also provided.

5.1. Evolution of affinity

The first experiment is designed with the aim of analyzing evolution of the repertoire with respect to its affinity.

The expression $x^2 + y^2 + 2 \cdot x \cdot y$ is considered for the experiment. The parameters of the IP system are set to $n = 100$, $P_r = 0.5$, $P_c = 0.1$, $P_m = 0.2$, where the P_r, P_c, and P_m parameters are selected based on the empirical results from the sensitivity analysis in Section 5.2. The program size is set to $L = 6$.

The evolved instruction sequence for this expression is `add dup mul`. This program corresponds to the expression $(x + y) \cdot (x + y) \equiv (x + y)^2$, which is a simplification of the original expression through factorization. In the trial under examination, the system finds the solution in generation 20. In each generation an average of 51 antibodies are replaced, 8 cloned, and 38 instructions are altered by hypermutation. Fig. 6 shows the growth of affinity during evo-

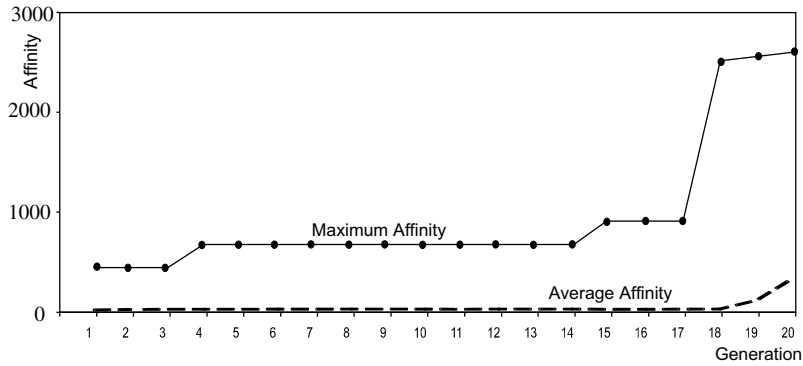

Fig. 6. Evolution of affinity over time.

lution of the solution. The plot shows gradual improvement of the maximum affinity until generation 17 at which there is a step improvement in the repertoire's maximum and average affinities. The maximum affinity of the population then hovers near the highest affinity until generation 20 when the problem is solved. The final ratio of average to maximum affinity is 72.06/2513 = 0.03.

5.2. Sensitivity analysis

To examine the sensitivity of the IP algorithm with respect to its parameters, the expression from Section 4.3.4 is considered: $x^2 + y^2 + x + y$. Varied parameters are listed, below, with their default values indicated in parentheses

Population size n ($n = 1000$),
Probability of replacement P_r ($P_r = 0.5$),
Probability of cloning P_c ($P_c = 0.1$),
Probability of mutation P_m ($P_m = 0.2$).

The program size, L, is kept constant at 10, and no limit is placed on the number of generations. Similar to previous experiments, the results of ten independent trials are averaged to avoid bias.

5.2.1. Effect of repertoire size, n

To examine the effect of repertoire size, n, its value is varied from 10 to 5000 in increments of 10. The number of generations required to produce the solution is plotted against the repertoire size in Fig. 7.

The shape of the curve corresponds to expectations: the greater the repertoire size, n, the lower the average number of generations, $\overline{G}$, needed to arrive at a solution.

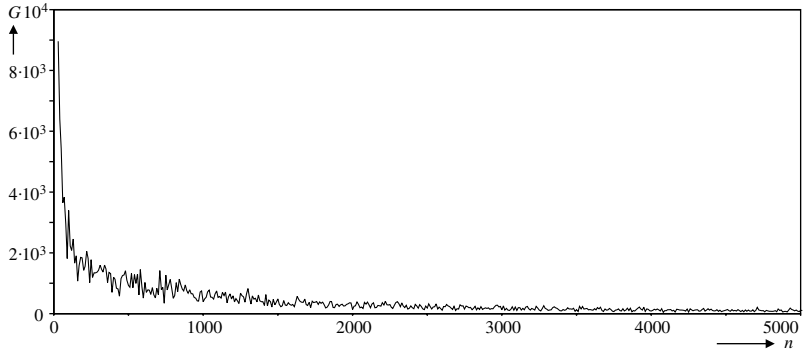

Fig. 7. Sensitivity of IP performance with respect to population size, n.

It is interesting to note that the algorithm converges and successfully finds a solution even with an extremely small repertoire size, such as $n = 10$. This is not possible with the GP algorithm, as discussed later in this section. Although the number of generations required is relatively high (e.g., 30,000 for $n = 10$) the proportion of the solution space searched is small. There are $r = 6$ instructions available and the program size is $L = 10$ yielding a search space of size $r^L = 6^{10} \doteq 6 \times 10^7$. The number of solutions actually considered by the algorithm is $G \cdot n = 30{,}000 \times 10 = 3 \times 10^5$, less than 0.5% of the solution space.

5.2.2. Effect of probability of replacement, P_r

In this experiment, P_r is varied from 0.01 to 0.80 in increments of 0.01. The number of generations required to produce the solution is plotted against the replacement probability, P_r, in Fig. 8. The curve is of a 'V' shape with the minimum around 0.55. This finding corresponds to replacement rates in biological immune systems where a large number of cells are newly produced to replace old cells. Despite the existence of an optimal value of the probability of replacement, the algorithm is able to find a solution consistently across the entire range $P_\mathrm{r} \in [0.01, 0.80]$.

5.2.3. Effect of probability of cloning, P_c

The probability of cloning, P_c, is varied from 0.01 to 0.99 in increments of 0.01. The number of generations required to produce the solution is plotted against the replacement probability, P_c, in Fig. 9. Overall, the curve has an increasing trend. There is, however, a minimum of $\overline{G}$ at a value of P_c around 0.1. Therefore, a relatively small portion of the repertoire is retained in subsequent generations. Larger values of P_c lead to stagnation of the algorithm due to loss of repertoire diversity.

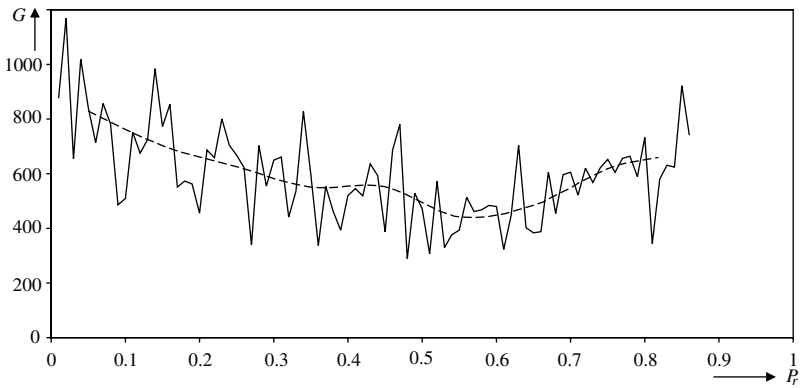

Fig. 8. Sensitivity of IP performance with respect to probability of replacement P_r. The solid line connects results for individual values of the parameter and the dashed line represents values averaged over an interval of size 0.1.

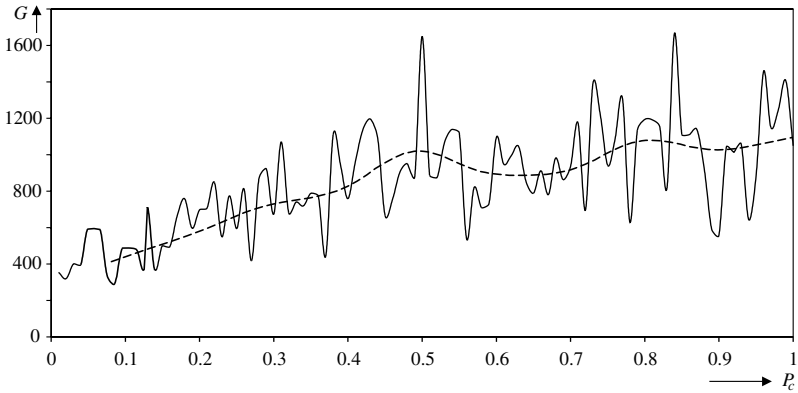

Fig. 9. Sensitivity of IP performance with respect to probability of cloning, P_c. The solid line connects results for individual values of the parameter and the dashed line represents values averaged over an interval of size 0.1.

5.2.4. Effect of probability of mutation, P_m

The value of P_m is varied from 0.01 to 0.99 in increments of 0.01. The number of generations required to produce the solution is plotted against the probability of mutation, P_m, in Fig. 10. The curve is of a 'V' shape with the minimum between 0.2 and 0.3. Please note that P_m affects the extent to which hypermutation is applied to a selected antibody. The antibody is selected based upon its affinity and the complement to the probability of cloning.

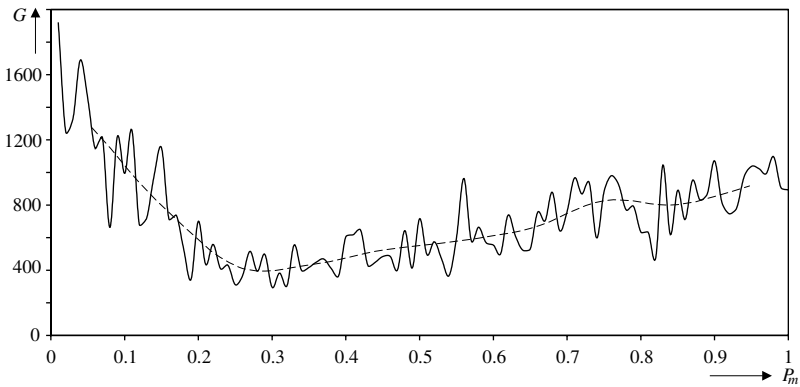

Fig. 10. Sensitivity of IP performance with respect to probability of mutation P_m. The solid line connects results for individual values of the parameter and the dashed line represents values averaged over an interval of size 0.1.

5.3. Comparison of immune programming and genetic programming

To compare the performance of the IP system with the stack-based GP algorithm, a stack-based version of the GP algorithm is implemented. To allow an unbiased comparison, this implementation reflects the differences between the two approaches but maintains the same method of generating test cases and evaluation of candidate solutions, as introduced in Section 4.2.

The expression considered for this experiment is

$$x^3 + 3 \cdot x^2 \cdot y + 3 \cdot x \cdot y^2 + y^3,$$

an expanded form of $(x + y)^3$. The minimal program length to solve this expression is $L = 5$:

`add dup dup mul mul` or `add dup over mul mul`.

The experiments are conducted for the minimal program length, $L = 5$, and also for twice this length, $L = 10$. The stopping criteria is either success in finding a solution or a maximum number of generations, $G_{max} = 1000$. Each of the experiments is repeated ten times and the results, listed in Table 6, represent the average taken over all trials.

The table also relates the number of trials (as a percentage of the ten trials) in which a solution is found within G_{max} generations. This measure is provided since in some trials neither algorithm finds a solution within the stopping criteria. The average number of generations, $\overline{G}$, to find a solution for 100% successful sets of trials are set in a bold font.

Table 6
Comparison of convergence of GP and IP

L	System	Repertoire/population size n									
		10	50	100	200	300	400	500	1000	2000	5000
5	GP	N/A	N/A	506	154	87	39	36	**10**	**3**	**1**
		0%	0%	20%	40%	60%	60%	60%	100%	100%	100%
	IP	298	**67**	**50**	**24**	**14**	**11**	**9**	**8**	**2**	**1**
		90%	100%	100%	100%	100%	100%	100%	100%	100%	100%
10	GP	N/A	468	266	103	130	130	32	**10**	**2**	**2**
		0%	40%	60%	60%	90%	90%	90%	100%	100%	100%
	IP	553	**76**	**30**	**28**	**17**	**18**	**19**	**8**	**2**	**2**
		90%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Each cell displays the average number of generations, $\overline{G}$, to find a solution and the percentage of trials that found a solution within $G_{\max} = 1000$.

The overall trend of the dependency between the population/repertoire size, n, and the number of generations to find a solution, $\overline{G}$, corresponds to the findings described earlier in Section 5.2.1. For a relatively large repertoire/population, $n > 1000$, both systems perform comparably well, in that they are both able to find a solution within 10 generations or less. For a smaller repertoire size, however, immune programming is clearly superior not only in terms of the average number of generations needed to find a solution, but also in the ability of the algorithm to find a solution within a restricted number of iterations. GP is not able to find solutions in all trials for any population of size smaller than 1000. The IP algorithm, on the other hand, achieves complete success for populations as small as $n = 50$. Even for $n = 10$ the IP algorithm is able to find a solution in 90% of trials.

The ability of IP to perform successfully with small populations can be attributed to the way in which the algorithm maintains repertoire diversity. This diversity is introduced during initialization steps which are analogous to GP population initialization. However, in IP diversity is subsequently maintained via replacement and mutation. Additionally, due to the affinity-based selection process, replacement and mutation rates are self-regulatory in that the lower the average affinity of a repertoire, the larger portion of antibodies of the repertoire are replaced or extensively mutated.

6. Conclusions

This paper introduced a new paradigm of evolutionary computing entitled 'Immune Programming' (IP). As an extension to Artificial Immune System

(AIS) concepts, IP is a systematic, domain-independent method to intelligently solve programming problems with no human interaction.

The problem to be solved is described by an antigen, and a set of candidate solutions is described by a repertoire of antibodies. Each antibody contains a string representing a sequence of stack-based assembly instructions, the set of which is analogous to gene libraries in immune systems. Randomly generated at initialization, the antibody repertoire is evaluated against the problem specification and a measure of its fit, affinity, is derived by executing each program corresponding to an antibody on a virtual machine. If affinity relates that a solution is not contained in the repertoire, the algorithm proceeds by creating a new generation of antibodies through replacement, cloning, and hypermutation processes. Application of a specific process is governed by an antibody's affinity and a set of probabilistic parameters. Processing continues until a solution is found or a predefined stopping criteria is attained.

Convergence of IP is superior to stack-based GP for the problems tested: successful solutions are found in fewer generations with the most dramatic improvement evident when using a small antibody repertoire. Additionally, IP converges in situations that cannot be handled by GP, arising from IP's inherently improved repertoire diversity. Sensitivity of IP's convergence with respect to algorithm parameters is explored in this paper's body. The generalization capabilities of the system are demonstrated by its capacity to provide a variety of alternative solutions to a given problem.

Still in its infancy, IP is showing great promise. Application to new problem areas is a natural extension of this body of work and is planned for the near future. Improvements to performance (convergence) are also planned, including

- modification of the algorithm to operate on repertoire in a deterministic, rather than probabilistic, manner;
- exploitation of the inherent memory of the modeled immune system by evolving a repertoire over a set of similar programming problems;
- increase of gene library (instruction set) size, perhaps to include instructions performing flow-control; and
- modification of or addition to hypermutation techniques to model mutation processes found in biological systems and to reflect the nature of many programming problems.

As an example of the latter, the iterative nature of many programs could be accommodated by an antibody structure with repeated instruction sequences. A hypermutation algorithm supporting this concept is expected to provide faster convergence.

References

[1] G.L. Ada, G.J.V. Nossal, The clonal selection theory, Scientific American 257 (2) (1987) 50–57.
[2] L.N. De Castro, F.J. Von Zuben, The clonal selection algorithm with engineering applications, GECCO'00—Workshop Proceedings (2000) 36–37.
[3] L.N. De Castro, F.J. Von Zuben, Learning and optimization using the clonal selection principle, IEEE Transactions on Evolutionary Computing 3 (6) (2002) 239–251.
[4] L.N. De Castro, J.I. Timmis, Artificial Immune Systems: A New Computational Intelligence Paradigm, Springer-Verlag, 2002.
[5] L.N. De Castro, J.I. Timmis, Artificial immune systems as a novel soft computing paradigm, Soft Computing 7 (2003) 526–544.
[6] C.A. Coello, N.C. Cortés, Hybridizing a genetic algorithm with an artificial immune system for global optimization, Engineering Optimization 36 (5) (2004) 607–634.
[7] D. Dasgupta, A. Ji, F. González, Artificial immune system (AIS) research in the last five years, in: Proceedings of the Evolutionary Computation Conference (CEC2003), Canbara, Australia, 2003, pp. 123–130.
[8] A.E. Eiben, J.E. Smith, Introduction to Evolutionary Computing, Springer, 2003.
[9] L.J. Fogel, A.J. Owens, M.J. Walsh, Artificial Intelligence through Simulated Evolution, John Wiley, 1966.
[10] D.B. Fogel, Evolutionary Computation: Toward a New Philosophy of Machine Intelligence, 2nd ed., IEEE Press, Piscataway, NJ, 1999.
[11] S. Forrest, S. Hofmeyr, A. Somayaji, Computer Immunology, Communications of the ACM 40 (10) (1997) 86–96.
[12] S. Forrest, A. Perelson, L. Allen, R. Cherukuri, Self-nonself discrimination in a computer, Proceedings of the IEEE Symposium on Research in Security and Privacy (1994) 202–212.
[13] S. Forrest, B. Javornik, R.E. Smith, A.S. Perelson, Using genetic algorithms to explore pattern recognition in the immune system, Evolutionary Computation 1 (3) (1993) 191–211.
[14] W. Gao, Fast immunized evolutionary programming, Proceedings of the Congress on Evolutionary Computation 1 (2004) 666–670.
[15] J.H. Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, MI, 1975.
[16] N.K. Jerne, Towards a network theory of the immune system, Annales D Immunologie (Inst. Pasteur) 125C (1974) 373–389.
[17] H. Juillé, J.B. Pollack, Parallel Genetic Programming on Fine-Grained SIMD Architectures, in: E.V. E.V., E.V. Siegel, Koza J.R. (Eds.), Working Notes of the AAAI-95 Fall Symposium on Genetic Programming, AAAI Press, 1995.
[18] P.J. Koopman, Stack Computers: The New Wave, Ellis Horwood, 1989.
[19] J.R. Koza, Genetic Programming, MIT Press, Cambridge, MA, 1992.
[20] J.R. Koza, M.J. Streeter, M. Keane, Routine high-return human-competitive machine learning, in: Proceedings of the International Conference on Machine Learning and Applications, Los Angeles, CA, 2003, pp. 6–12.
[21] G.F. Luger, Artificial Intelligence: Structures and Strategies for Complex Problem Solving, 4th ed., Addison–Wesley, 2002.
[22] W. Luo, X. Cao, X. Wang, An immune genetic algorithm based on immune regulation, in: Proceedings of the 2002 Congress on Evolutionary Computation, IEEE Press, Honolulu, Hawaii, 2002, pp. 801–806.
[23] D.F. McCoy, V. Devaralan, Artificial immune systems and aerial image segmentation, Proceedings of the SMC'97 (1997) 867–872.
[24] Z. Michalewicz, Genetic Algorithms + data Structures = evolution Programs, third ed., Springer-Verlag, Heidelberg, 1996.

[25] N.I. Nikolaev, H. Iba, V. Slavov, Inductive genetic programming with immune network dynamics, in: Advances in Genetic Programming 3, MIT Press, 1999, pp. 355–376.
[26] G.J.V. Nossal, Negative selection of lymphocytes, Cell 76 (1994) 229–239.
[27] G.J.V. Nossal, The molecular and cellular basis of affinity maturation in the antibody response, Cell 68 (1993) 1–2.
[28] A.S. Perelson, G.F. Oster, Theoretical studies of clonal selection: Minimal antibody repertoire size and reliability of self-non-self discrimination, Journal of Theoritical Biology 81 (4) (1979) 645–670.
[29] T. Perkis, Stack-based genetic programming, in: Proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, 1994, pp. 148–153.
[30] K. Stoffel, L. Spector, High-performance, parallel, stack-based genetic programming, in: J.R. Koza et al. (Eds.), Genetic Programming 1996: Proceedings of the First Annual Conference, MIT Press, 1996, pp. 224–229.
[31] J. Timmis, C. Edmonds, J. Kelsey, Assessing the performance of two immune inspired algorithms and a hybrid genetic algorithm for function optimisation, Proceedings of the Congress on Evolutionary Computation 1 (2004) 1044–1051.
[32] S.T. Wierzchoń, Function optimization by the immune metaphor, Task Quarterly 6 (3) (2002) 493–508.