Evolving A-type artificial neural networks

Ewan Orr · Ben Martin

Received: 1 December 2010 / Accepted: 17 August 2011 / Published online: 8 September 2011
© Springer-Verlag 2011

Abstract We investigate Turing’s notion of an A-type artificial neural network. We study a refinement of Turing’s original idea, motivated by work of Teuscher, Bull, Preen and Copeland. Our A-types can process binary data by accepting and outputting sequences of binary vectors; hence we can associate a function to an A-type, and we say the A-type represents the function. There are two modes of data processing: clamped and sequential. We describe an evolutionary algorithm, involving graph-theoretic manipulations of A-types, which searches for A-types representing a given function. The algorithm uses both mutation and crossover operators. We implemented the algorithm and applied it to three benchmark tasks. We found that the algorithm performed much better than a random search. For two out of the three tasks, the algorithm with crossover performed better than a mutation-only version.

Keywords Turing’s A-types · Artificial neural network · Evolutionary algorithm

1 Introduction

In this paper we report on our investigations into one of Alan Turing’s contributions to artificial intelligence. In 1948 Turing introduced a type of artificial neural network (ANN), which he called an A-type unorganised machine. Motivated by his work and by work of Teuscher, Bull, Preen and Copeland (see Sect. 2), we study a refinement of Turing’s notion, which we call an A-type.

A-types can be used to process binary data: with suitable conventions involving input and output nodes, one can input a string of binary vectors into an A-type and receive a string of binary vectors as output. Hence we can associate a function to an A-type; we say that the A-type represents this function. We devised an evolutionary algorithm (EA) to design an A-type that represents a given function \( f \).

Turing’s research on A-types is of great historical interest. As the centenary of his birth approaches, it is fitting to apply modern ideas—such as the theory of non-linear dynamical systems—to his ground-breaking work. A-types are an excellent test-bed for these ideas: they are composed of neurons with a very simple firing rule and they are easy to program, but they are also powerful. In this paper we adapt some existing ideas such as graph-based chromosomes and sequential input to the setting of A-types. The use of sequential input mode here brings up some new problems which motivated us to introduce a new kind of neuron, delay nodes, not originally envisaged by Turing (see Sect. 3.5). Our graph-based EA works in the settings of both sequential and clamped input.

In Sect. 2 we give a brief survey of previous work on A-types. In Sect. 3 we present our interpretation of A-types, and we describe our EA in Sect. 4. Section 5 contains the results of our experimental work.

Our investigations are mainly at the proof-of-concept level. Our EA has many parameters and we chose their values...
in an ad-hoc fashion to ensure that solutions were quickly discovered reasonably often; we did not search systematically for the optimum values (but see Sects. 5.6, 6.1).

2 Historical background and previous work

In 1948 Turing wrote the pioneering technical report *Intelligent Machinery* [31]. In this report he introduced a type of ANN which he called an A-type unorganised machine. This ANN is discrete, synchronously updated and, in general, recurrent. It is composed of basic and identical neurons (or nodes) each of which performs the Boolean operation NAND. The neurons are connected by arrows. For any Boolean function \( f \), there exists a feed-forward A-type unorganised machine \( A \) that ‘represents’ \( f \). That is, there is always an A-type that given an input vector of Boolean values \( x \) will output the vector of Boolean values \( f(x) \) (see Sect. 3.3.1). Throughout this paper we use the term ‘A-types’ to refer to Turing’s A-type unorganised machines. We also apply this term when we discuss our interpretation of Turing’s A-type unorganised machines and those of other researchers; we hope that the meaning is clear from the context.

In [31] Turing introduced three models of computation: A-types, B-type unorganised machines, and P-type unorganised machines. In our research we only use A-types. However, we mention these other two models to explain their relevance to our research.

The second ANN that Turing introduced was a special kind of A-type, which he called a B-type unorganised machine. These networks are effectively A-types the arrows of which can be switched on and off by changing the state of particular nodes in the network. Turing constructed these switchable arrows with a particular configuration of nodes and arrows. In the late 1940s A-types would have had to have been directly implemented in hardware; Turing’s B-type unorganised machines offer a means of effectively reconfiguring the topology of a network without reconfiguring hardware. Today, ANNs are often implemented in software that is several levels of abstraction above computer hardware; however, there may be novel architectures for which the reconfigurable architecture B-type unorganized machine is useful.

In [31] Turing introduced P-type unorganised machines. Unlike a B-type, a P-type is not a special case of an A-type (nor is a P-type a generalisation of an A-type). Turing used P-types to investigate learning. This pioneering work would now be classed as an investigation into reinforcement learning. For further details see Copeland [6].

Artificial neural networks have found wide application and are an active area of research, yet only a few researchers have continued Turing’s work on A-types. In 1996 Copeland and Proudfoot [6] re-examined this research. The most notable continuation of research into Turing’s networks was conducted in 2001 by Teuscher [29]. Teuscher experimented with A-types with fixed input states; for instance, he used A-types in this manner to solve basic pattern classification tasks and he showed that their dynamics are analogous to those of a non-linear oscillator [30]. Teuscher employed EAs to train Turing’s networks: he used linear data structures (linear chromosomes) to represent B-types. Teuscher used B-types with lists that prescribed whether each arrow in a B-type was in a ‘connected’ or ‘disconnected’ state [29, p. 88]. These lists give linear chromosomes for Teuscher’s B-types.

Today, Turing’s A-types can be considered a special class of Random Boolean Networks [29, p. 25]. Random Boolean Networks are simple discrete dynamical systems that are capable of complex behaviour; consequently, they are useful for modelling complex systems such as gene regulation mechanisms in biology and the internet [12, 24]. Teuscher investigated the non-linear dynamics of A-types [29, ch. 5], [30]. Recently, Bull [3], and Bull and Preene [4] investigated the evolution of A-type machines, and they considered this in the context of discrete dynamical systems.

3 Our interpretation of Turing’s A-types

In this section we present our definition of an A-type. This is an interpretation of Turing’s A-type unorganised machines, and has been influenced by Teuscher’s research [29]. We also provide illustrations of our A-types, and we compare our definition with those of Turing and Teuscher.

3.1 Our definition

An A-type is a discrete, recurrent, synchronously updated ANN. The firing rule for every neuron in an A-type is invoked simultaneously—we can imagine that all neurons are updated via the same clock. Each of the instants at which the neurons in an A-type are synchronously updated is called a moment.

In order to define A-types, we need the notion of an A-type graph. An A-type graph is a directed graph\(^\text{7}\) with the following properties. Every node has an indegree no

\(^1\) This was seemingly independent [7, p. 408] of the 1943 paper [17] of McCulloch and Pitts in which ANNs were first introduced.
greater than two. An A-type graph has a non-empty set of nodes called input nodes, each of which has indegree zero. An A-type graph has a non-empty set of nodes called output nodes, each of which has outdegree zero. The set of input nodes and set of output nodes do not intersect. Arrows from an input node to an output node are not permitted. Nodes that are not output nodes have no restriction on their outdegree.

An A-type consists of an A-type graph and a non-negative integer \( \delta \), called the delay time. We interpret the nodes of the graph as the neurons of an ANN, and the arrows of the graph as the interconnections. The delay time determines the number of moments from when information first enters the input nodes to when we start to collect information from the output nodes (we elaborate on this in Sect. 3.3.1). We call the number of input (output) nodes of an A-type its input (output) dimension. Because A-types are recurrent, A-type graphs can have closed paths.

An A-type is a Boolean ANN. Consider an A-type \( A \). Each interconnection of \( A \) carries exactly one bit of information per moment. That is, we associate a Boolean variable with every arrow in the A-type graph of \( A \). Every node in \( A \) has a firing rule that is a Boolean function (of the variables entering that node). Furthermore, every node in an A-type has a Boolean variable associated with it. We call this variable the state of that node. In general, the state of a node varies from moment to moment. At any moment the output of a node is equal to the state of the node.

We classify every node that is not an input node into one of two types depending on its firing rule: nand nodes and delay nodes. A nand node \( q \) has an indegree of two and its firing rule is NAND. That is, let \( a \) and \( b \) denote the Boolean values associated with the respective arrows entering \( q \) at moment \( t \). At moment \( (t + 1) \) the state of \( q \) is \( a \) NAND \( b \). A delay node \( d \) has an indegree of one and its firing rule is the identity. That is, let \( a \) denote the Boolean value associated with the arrow entering \( d \) at moment \( t \). At moment \( (t + 1) \) the state of \( d \) is \( a \). A nand node can accept two inputs from a single nand or delay node. Note that we initialize the state of every non-input node to zero. We explain the rules for initialising and updating input nodes in Sect. 3.3.

3.2 Illustrations

In graph theory diagrams are employed to represent a graph. We use similar diagrams for our A-type graphs. Input nodes are represented by circles with no incoming arrows. Nand nodes are represented by circles that have two incoming arrows. Delay nodes are represented by triangles that have one incoming arrow. Output nodes are denoted by doubled circles or doubled triangles. We illustrate these conventions in Fig. 1. In Fig. 2 we depict a simple A-type.

3.3 Processing information

In this section we describe how we employ A-types to process information. By a Boolean vector we mean a vector the components of which are all either 0 or 1. We denote by \( S_m \) the set of all \( m \)-component Boolean vectors. We now explain how A-types can accept and output sequences of Boolean vectors.

3.3.1 Input and output

Consider an A-type \( A \) that has input dimension \( n \). To enable us to input information into \( A \) we adopt the following update rule for the input nodes of \( A \). Choose an ordering on the set of input nodes. Suppose we are given a sequence of \( n \)-component Boolean vectors \( (x_0, \ldots, x_q) \). For the first \( q \) moments, the states of the input nodes of \( A \) at moment \( t \) are given by the components of \( x_t \) in the appropriate order. In particular, the initial states of the input nodes are determined by \( x_0 \). We say that at moment \( t \), \( x_t \) is input into \( A \). We adopt the convention that after the input vectors are used up the states of the input nodes remain constant, keeping the values from the final input vector. That is, for moments \( t > q \) the states of the input nodes are given by the components of \( x_q \).

Consider an A-type \( A \) that has delay \( \delta \) and output dimension \( p \). We collect output information from the
A clappable A-type with delay $\delta = 2$

output nodes of $A$, starting not at moment 0 but at moment $\delta$; the idea is that it may take data some time to percolate through the A-type and reach the output nodes. At each moment $t \geq \delta$ the states of the output nodes of $A$ generate a $p$-component Boolean vector $y_t$. We say that at that moment $A$ outputs $y_t$.

A-types can be viewed as non-linear dynamical systems [29, p. 132]. Because our A-types accept sequential data, they are analogous to non-linear oscillators that are subject to a driving force that is generally not constant. Note that when we use an A-type to process binary data, the delay $\delta$ is a parameter which is independent of the input data. If an A-type is to represent a sequential function in the sense of Sect. 3.4 below, it must have the special property that the time for the input data to travel to the output(s) should not depend on the choice of input data.

3.3.2 Clamped input mode

Consider the special case when a single Boolean vector $x_0$ is input into an A-type $A$. The states of the input nodes of $A$ stay constant, with values determined by $x_0$. In this case we say that the input nodes of $A$ are clamped by $x_0$, and we say that we are operating $A$ in clamped mode.

Consider an A-type $A$, with delay $\delta$. Let $A$ be clamped by some input vector $x_0$. If the states of the output nodes of $A$ are constant for all moments $t \geq \delta$ then we say $A$ is clappable with respect to the input $x_0$. We say $A$ is clappable if it is clappable for every $x_0$.

We can operate an A-type in clamped input mode even when it is not clappable. Because the graph of an A-type is finite, if an A-type is operated in clamped mode then eventually the output becomes periodic. For a clappable A-type, this period is always one.

We now present an example of a clappable A-type. Consider the A-type $A_\wedge$, with a delay of $\delta = 2$, shown in Fig. 3. It is easy to check that $A_\wedge$ is clappable and that for every input $[a, b]^3 \in S_2$, the eventual output is $a \wedge b$.

A Boolean function is a function from $S_n$ to $S_p$ for some positive integers $n$ and $p$. Consider a Boolean function $f$ and a clappable A-type $A$. We say that $A$ represents the Boolean function $f$ if the following holds: for each $x \in S_n$, if $\text{represents } f$ when $x_0 = x$. Consider an A-type $A$ with delay $\delta$, input dimension $n$ and output dimension $p$. Recall that at every moment $t$, $A$ accepts an $n$-component Boolean vector $x_t$ and for each moment $t \geq \delta$, $A$ returns a $p$-component Boolean vector $y_t$.

We say that we are operating $A$ in sequential mode.

In Sect. 3.3.1 we defined a way of inputting information into an A-type so that A-types can accept and return sequences of Boolean vectors. We considered constant input and output sequences in Sect. 3.3.2. In general, the sequences that A-types accept need not be constant.

Consider an A-type $A$ with delay $\delta$, input dimension $n$ and output dimension $p$. Recall that at every moment $t$, $A$ accepts an $n$-component Boolean vector $x_t$, and for each moment $t \geq \delta$, $A$ returns a $p$-component Boolean vector $y_t$. We say that we are operating $A$ in sequential mode.

In Fig. 4 we illustrate a simple A-type $A$ with delay $\delta = 2$, and an input sequence of 5 Boolean vectors. The A-type $A$ returns an output sequence consisting of 3 Boolean vectors. In Fig. 5 we illustrate how $A$ changes over these moments. Each subfigure is a snapshot of the entire A-type at a particular moment. We give the input Boolean vector, the states of the nodes of $A$ and the output vector at that moment. In Fig. 6 we illustrate $A$, the input sequence for the first five moments, and the output sequence for the first five moments.4

Note that (by our convention introduced in Sect. 3.3.1) if a sequence of $l$ Boolean vectors is input into an A-type then for every moment $t > l$ the states of the input nodes of that A-type are constant. This convention serves to ‘shunt’ information through an A-type. For instance, in Fig. 4 the initial input sequence and the desired output sequence have length 3, but 5 moments are needed to collect the output because $\delta = 2$. Our shunting convention ensures that the input states are well-defined for the final two moments.

3.3.3 Sequential input mode

In Sect. 3.3.1 we defined a way of inputting information into an A-type so that A-types can accept and return sequences of Boolean vectors. We considered constant input and output sequences in Sect. 3.3.2. In general, the sequences that A-types accept need not be constant.

Consider an A-type $A$ with delay $\delta$, input dimension $n$ and output dimension $p$. Recall that at every moment $t$, $A$ accepts an $n$-component Boolean vector $x_t$, and for each moment $t \geq \delta$, $A$ returns a $p$-component Boolean vector $y_t$. We say that we are operating $A$ in sequential mode.

In Fig. 4 we illustrate a simple A-type $A$ with delay $\delta = 2$, and an input sequence of 5 Boolean vectors. The A-type $A$ returns an output sequence consisting of 3 Boolean vectors. In Fig. 5 we illustrate how $A$ changes over these moments. Each subfigure is a snapshot of the entire A-type at a particular moment. We give the input Boolean vector, the states of the nodes of $A$ and the output vector at that moment. In Fig. 6 we illustrate $A$, the input sequence for the first five moments, and the output sequence for the first five moments.

Note that (by our convention introduced in Sect. 3.3.1) if a sequence of $l$ Boolean vectors is input into an A-type then for every moment $t > l$ the states of the input nodes of that A-type are constant. This convention serves to ‘shunt’ information through an A-type. For instance, in Fig. 4 the initial input sequence and the desired output sequence have length 3, but 5 moments are needed to collect the output because $\delta = 2$. Our shunting convention ensures that the input states are well-defined for the final two moments.
Representing sequential functions

Here we explain how to associate a function to an A-type. In Sect. 3.3.2 we defined what it means for a clampable A-type operating in clamped input mode to represent a Boolean function. Here we generalize this notion.

Let $S_{m,l}$ denote the set of all sequences of length $l$ consisting of $m$-component Boolean vectors. Note that $S_{m,1} = S_m$. Consider a function $f$ from $S_{n,k}$ to $S_{p,l}$, for some positive integers $k$, $l$, $n$, and $p$. We call $f$ a sequential Boolean function. A Boolean function $f$ is the special case of a sequential Boolean function with $k = l = 1$. We say that an A-type $A$ represents $f$ if for every $x \in S_{n,k}$, when $A$ accepts $x$ it outputs the sequence $f(x)$. So if $A$ represents $f$ then the input dimension of $A$ must be $n$ and the output dimension of $A$ must be $p$.

For example, consider serial addition. We can describe this in terms of a sequential function $f$ which maps $S_{2,1}$ to $S_{1, (l+1)}$, for some positive integer $l$. Given an input sequence $x$, the first entries of the vectors in $x$ give the binary encoding for some integer $a$, the second entries of the vectors in $x$ give the binary encoding for some integer $b$, and the entries of the vectors in $f(x)$ give the binary encoding for the integer $(a + b)$. In Sect. 3.4.1 we describe another class of sequential functions, the columnwise Boolean functions.

Let us touch upon the possible functions that our A-types can represent when operated in sequential input mode. Recall from Sect. 3.3.2 that any clamped Boolean function can be represented by a clampable feed-forward A-type. We can regard representing functions in clamped mode as a special case of representing functions in sequential mode: the input sequences have length 1 and the output sequences are required to be constant. Because of this, sequential tasks are generally more difficult than clamped tasks. In principle, one can devise an A-type that represents binary addition of strings $s_1$ and $s_2$ of arbitrary length; however, in practice this is not trivial. It is impossible [18, p. 27] to devise an A-type that represents binary multiplication of strings $s_1$ and $s_2$ of arbitrary length.

3.4 Representing sequential functions

We define a columnwise Boolean function as follows. Let $n, p$ be positive integers and suppose we are given a positive integer $k$. For any Boolean function $f$ we define columnwise $f$ to be the function that maps $S_{n,k}$ to $S_{p,k}$ for any positive integer $k$, such that if $x_i$ denotes the $i$th term of an input sequence and $y_j$ denotes the $j$th term of the corresponding output sequence then $y_j = f(x_i)$. This says that bits of the input in different columns do not interact with each other. Conversely, if $g$ is a columnwise Boolean function then we call the underlying Boolean function clamped $g$.

5 For example, Minsky [18, p. 27] describes a McCulloch-Pitts network that performs serial addition. In principle, these details could be used to construct an A-type that represents binary addition of $s_1$ and $s_2$. 

Evol. Intel. (2012) 5:3–22
chosen from the interval $[8, 40]$. A sequence of 10 4 ran-
nodes and tested whether it represented columnwise
repeatedly constructed a random $A$-type without delay
represented columnwise Exclusive-OR. Similarly, we
random A-type with delay nodes and tested whether it
represented columnwise Exclusive-OR.

Claim: There does not exist an $A$-type without delay
supports the following claim.

For example, let us consider columnwise Exclusive OR.
The Boolean function $\text{Exclusive OR} \oplus$ maps $S_2$ to $S_1$. Col-
umwise Exclusive OR maps a sequence $x$ of $k$ 2-component
Boolean vectors to a sequence $y$ of $k$ 1-component Boolean
vectors, such that the $i$th term of $y$ is $\oplus(x_i)$, where $x_i$ denotes
the $i$th term of $x$. It is easy to check that the $A$-type shown in
Fig. 7 represents columnwise Exclusive OR.

One can show that $A$-type representing a columnwise
Boolean function $f$ also represents clamped $f$, but the
converse is false in general for the reasons discussed at the
start of Sect. 3.5.

3.5 The necessity of delay nodes

Operating $A$-types in sequential mode brings some new
challenges. Data travels through the $A$-type from input
nodes to output nodes along various paths. If these paths
have different lengths then the arrival times are not syn-
chronised. In order to represent sequential functions, it is
useful—and, we believe, sometimes necessary—to have a
way to stagger the data. This is why we introduced delay
nodes, which do not appear in Turing’s original notion of
an $A$-type.

We collected evidence that delay nodes are necessary
for $A$-types to perform certain sequential tasks. In partic-
ular, via computer simulations we collected evidence that
supports the following claim.

Claim: There does not exist an $A$-type without delay
nodes that represents columnwise Exclusive-OR.

We employed a blind search for $A$-types representing
columnwise Exclusive OR. We repeatedly constructed a
random $A$-type with delay nodes and tested whether it
represented columnwise Exclusive-OR. Similarly, we
repeatedly constructed a random $A$-type without delay
nodes and tested whether it represented columnwise
Exclusive-OR. The size of each $A$-type was randomly
chosen from the interval $[8, 40]$. A sequence of $10^4$ ran-
donaly generated $2 \times 1$ input vectors was used to test
whether an $A$-type represented columnwise Exclusive-OR:
if an $A$-type represented columnwise Exclusive-OR for
such an input sequence then it was deemed to do so for all
input sequences. The results of these searches are presented
in Table 1. In summary, we discovered many $A$-types with
delay nodes that represented columnwise Exclusive-OR;
however, we failed to find a single $A$-type without delay
nodes that represented columnwise Exclusive-OR.

It is often the case that an $A$-type representing a clamped
Boolean function can be modified to represent the corre-
sponding columnwise Boolean function by adding some
delay nodes. The delay nodes are used to stagger data
flowing through parts of an $A$-type and ensure the overall
flow is synchronised. For example, consider again the
Boolean function $\text{Exclusive OR}$. We can write $A \oplus B$ as
$(A \lor B) \land (A \bar{x} \lor B)$. From this expression we devise a way to
construct an $A$-type $A_0$ that represents clamped Exclus-
ive OR, using $A$-types that represent columnwise Inclusive-OR
and columnwise AND; we illustrate this in Fig. 8a. Next
we construct an $A$-type $A_1$ by inserting a delay node into
$A_0$; we illustrate this in Fig. 8b. This delay node ensures
that data is synchronised as it flows through $A_1$; conse-
quently, $A_0$ represents columnwise Exclusive-OR.

Can one mimic the effect of the delay node using only
nand nodes? We can formulate this question in terms of
$A$-types that represent the identity. Suppose there exists an
$A$-type $I_m$ with a delay $\delta = m$, where $m$ is an odd positive
integer, such that $I_m$ contains no delay nodes. It is
straightforward to find an $A$-type $I_{m-1}$ with even delay
$m - 1$ representing the identity function such that $I_{m-1}$
contains no delay nodes: Fig. 12a gives an $A$-type that
works for the special case of delay $\delta = 2$, and we can
obtain any even delay by concatenating copies of this
$A$-type. Let us construct an $A$-type $A_2$ as follows: we insert
$I_m$ between nodes 4 and 6, and we insert $I_{m-1}$ between
nodes 5 and 6 (if $m - 1 = 0$ then we just put a single
arrow directly from node 5 to node 6). See Fig. 8c. This
ensures that the two inputs into node 6 are synchronised.

It is clear from the above discussion that if there exists
an $A$-type $I_m$ as above then we can mimic the effect of a
delay node using only nand nodes. The converse is also
ture, since a delay node represents the identity function
with delay 1. This motivates the following claim.

---

**Table 1** The results of our blind searches for $A$-types that represent
columnwise Exclusive-OR

|                      | With delay nodes | Without delay nodes |
|----------------------|------------------|---------------------|
| Number of attempts   | $1.6 \times 10^6$| $10^6$              |
| Probability that a node is constructed as a delay node | 20% | 0% |
| Number of solutions  | 1,342            | 0                   |

---

**Fig. 7** An $A$-type with delay time $\delta = 3$ that represents columnwise Exclusive OR
Claim: There does not exist an A-type without delay nodes and with an odd delay that represents columnwise identity. The construction illustrated in Fig. 8c shows that if this claim is false then the previous claim is also false. We employed a blind search for a counter-example to the claim. We repeatedly constructed a random A-type without delay nodes and tested whether it represented columnwise identity. The size of each A-type was randomly chosen from the interval \([3, 20]\). For each A-type a sequence of \(10^4\) randomly generated \(1 \times 1\) input vectors was used to test whether the A-type represented columnwise identity: if an A-type represented columnwise identity for such a sequence then it was deemed to do so for all input sequences. The results of these searches are presented in Table 2. In summary, we discovered many A-types with even delay that represented columnwise identity; however, we failed to find a single A-type with an odd delay that represented columnwise identity.

From the above results, we conjecture that A-types with delay nodes operated in sequential mode can represent a more general class of function than A-types without delay nodes. We found experimental evidence that supports our claims, but we were not able to discover a formal mathematical proof. We leave this as an open problem.

It is clear from the above discussions that we can implement a delay of any length by concatenating the following: (a) a single delay node, and (b) an A-type with even delay and without delay nodes that represents the identity. Hence only a small number of delay nodes is needed in any given A-type.

### 3.6 Comparison with previous definitions

Our definition of an A-type, given above, differs from those of Turing and Teuscher. The differences are in our allocation of input and output nodes, and our introduction of delay nodes.

Turing did not precisely prescribe how information could be input and output for A-types. To address this issue Teuscher [29, p. 32] defined A-types with input and output nodes. Essentially, we have adopted Teuscher’s conventions for input and output nodes.

We introduce delay nodes so to allow our A-types to process sequential input. Neither Turing nor Teuscher make use of delay nodes. However, Teuscher [29, p. 67] investigates sequential tasks by in effect employing two clock speeds: one for the rate of information input and output, and one for the rate of information flow between neurons. We chose to introduce delay machines because they allow a straightforward way to interpret Turing’s A-types so that they can operate with a sequential input. Furthermore, the training algorithms that we employ for our A-types are useful in both the clamped and the sequential settings—see Sect. 5.

![Fig. 8](image-url)

**Table 2** The results of our blind searches for columnwise identity

| Solution A-type | Frequency |
|-----------------|-----------|
| Number of attempts | 34000000000 |
| Number of solutions with even delay | 971789859 |
| Number of solutions with odd delay | 0 |
4 Learning via evolution

We now turn to our central problem: how to find an A-type that represents a given function $f$. We implemented a reinforcement learning technique involving an EA which searches for ‘suitably small’ A-types that represent $f$.

In his pioneering paper [31] Turing examines reinforcement learning. For instance, he defines a P-type machine to elaborate on some of his ideas. Furthermore, Turing briefly mentions a ‘genetical search’, but does not provide details of such a training method. One popular modern reinforcement learning technique is EAs, and now their use to train ANNs is established [10]. Teuscher used EAs to train B-types [29]. We also use EAs to train A-types.

In this section we outline a simple EA that we employ, and we present our mutation and crossover operators. In particular, we describe our efforts to devise useful crossover operators (see Sect. 4.4); further details can be found in [20]. In Sect. 5 we explain how we tested our EA and we present the results of these tests. Our EA works for A-types in both clamped and sequential modes.

When we implement our EA, we need to assign values to various parameters. Some of these values are task-dependent. We give the parameter values in Sect. 5.

4.1 Introducing our EA

In Table 3 we give an outline of our EA. We call this EA genetic_search_one. This EA is a straightforward implementation; for example, it is similar to the scheme outlines in [9, ch. 2], and the scheme outlined in [19, ch. 9]. Note that genetic_search_one is a steady state EA in that its population has only a small variation from generation to generation. In later sections we require the listing in

| Table 3 | An outline of the EA, genetic_search_one, that we use in this paper |
|---|---|
| 1. Create initial population: Randomly generate a specified number of candidate solutions of size within a specified range |
| 2. Iterate through successive generations: Repeat until either the population contains a solution or a maximum number of attempts have been performed |
| (a) Perform a set number of crossovers: Repeat a set number of times |
| i. Parent selection: Select a pair of candidate solutions as parents. The fitter a candidate solution, the greater the probability that it is chosen as a parent |
| ii. Crossover: For each parent pair combine information from both parents to produce a new candidate solution, which is added to the population |
| iii. Survivor selection: Select a member of the population and delete it from the population |
| (b) Perform a set number of mutations: Repeat a set number of times |
| i. Mutation: Randomly select a member of the population, copy it, slightly modify the copy, and add the modified copy to the population |
| ii. Survivor selection: Select a member of the population and delete it from the population |
| 3. Return the fittest candidate solution in the population. If there is more than one candidate solution with the lowest fitness of the population then we randomly select an element from the set of such individuals |

4.2 Chromosomes

Our candidate solutions are graph-like; this is made explicit by our use of an A-type graph to define an A-type. An A-type graph can be represented by an adjacency list. Teuscher [29, p. 88] demonstrated that A-types (and B-types) can be assigned linear chromosomes. If an EA employs linear chromosomes then it is easy to implement simple crossover and mutation operators; for example, bit-flipping mutation and one-point crossover [9, ch. 3].

We choose to represent A-types with graph chromosomes because it allows a straightforward implementation of some graph-theoretic manipulations on A-types. In particular, adding and removing topologically connected subgraphs from the graph of an A-type becomes straightforward. We encode an A-type graph as an object which has a collection of node objects associated to it; each node object can reference other node objects. This approach has two advantages: it captures the topology of an A-type graph, and it does not impose an artificial ordering on the nodes.

4.3 The initial population

Our EA requires an initial population of A-types to be created. To do this, random A-types are generated with size between a specified upper bound and a specified lower bound. (We define the size of an A-type $A$ to be the number of nodes it contains, and we denote this by $|A|$.) The mutation and crossover operators can change the size of A-types, so subsequent populations can contain A-types whose size is outside the original bounds.
4.4 Evolutionary operators

Our EA involves mutation and crossover operators. Here we describe our implementation of these operators.

4.4.1 Mutation

Our mutation operator manipulates an A-type graph. The search space of our EA contains A-types that have a range of sizes. Consequently, we construct a mutation operator that can alter the size of an A-type. More precisely, our mutation operators accept an A-type $A_{in}$ and return an A-type $A_{out}$ such that either $|A_{out}| = |A_{in}| - 1$, or $|A_{out}| = |A_{in}|$, or $|A_{out}| = |A_{in}| + 1$. We achieve this by copying the input A-type: $A_{out} / A_{in}$, and performing one of the three following operations. One, a node $n$ is removed (if possible) from $A_{out}$ and there is a slight re-arrangement of the graph of $A_{out}$ in order to make $A_{out}$ into a valid A-type. Two, a single arrow is removed from $A_{out}$ and a new arrow inserted in order to make $A_{out}$ into a valid A-type. Three, a node $n$ is added to $A_{out}$ and arrows are added, and there is a slight re-arrangement of the graph of $A_{out}$ in order to ensure that $n$ has an output arrow and $A_{out}$ is a valid A-type. We illustrate these operations in Fig. 9.

4.4.2 Crossover

Our crossover operator involves operations that respect the topology of the graphs of the parent A-types. The operator exchanges subgraphs of the parents. Only topologically connected chunks of the parent graphs are exchanged, and reconnection of exchanged chunks involves only the insertion of arrows that bridge the ‘boundaries’ of these chunks. We make this more precise in the following subsections. In this section we present a crossover scheme which employs these ideas. In Sect. 5 we describe our tests of this crossover operator.

Our crossover operator accepts two parent A-types $\varnothing$, $\emptyset$ and returns one child A-type $C$. Two aspects of this crossover operator require further explanation: the acceptor and donor subgraphs are graphs of a particular type, and there are restrictions on the arrows that may be inserted to reconnect the child $C$. We elaborate on these two aspects next.

The donor and acceptor are subgraphs of a particular kind, which we call radial subgraphs. Consider a graph $G$ and a node $c \subseteq G$, which we call the centre ($c$ is chosen randomly in the crossover operator). If possible, we construct a radial subgraph of $G$ with $N$ nodes about the centre $c$ as follows. Construct a set $S$ which initially contains only $c$. Let $\bar{S}$ denote the set of all nodes that are adjacent to nodes in $S$ but are not already in $S$. We randomly select elements of $\bar{S}$ and transfer them to $S$ until $|S| = N$ or $|\bar{S}| = 0$ (where $|S|$ denotes the size of $S$). We repeat the above process of constructing a set of nodes that are adjacent to $S$ and selecting from that set until $|S| = N$ or $|\bar{S}| = 0$. At any point if $|S| = N$ then we use $S$ to generate a subgraph from $G$. This subgraph is the desired radial subgraph.

For each of the acceptor and donor sets, the size $N$ is a randomly chosen integer between 1 and a fixed proportion of the total size of the parent graph. The crossover algorithm always exchanges ‘localized’ and connected subgraphs of the graphs of the parents.

When our crossover reconnects subgraphs in the graph of the child, arrows may only be inserted between boundaries of the acceptor and donor subgraphs. To explain this process we define two types of boundaries: proximal boundaries and distal boundaries. Consider a graph $G$ with a subgraph $S$. Also, let $G - S$ denote the complement of $S$. 

---

6 Two nodes are adjacent if they are the endpoints of a particular arrow. That is, two nodes are adjacent if there is an arrow connecting them.
The proximal boundary of $S$ is the set of nodes in $S$ that are adjacent to nodes in $G - S$. The distal boundary of $S$ is the set of nodes in $G - S$ that are adjacent to nodes in $S$. For our crossover operator, the final step of constructing the child requires the insertion of arrows between the complement of the acceptor and the donor. Arrows are only inserted between nodes in the distal boundary of the acceptor and the proximal boundary of the donor.

We give an outline of our crossover operator in Table 4. We give a concrete example of our crossover operator in Fig. 10. As this example shows, the acceptor and donor subgraphs can have different sizes; also, the two parents and the child can all have different sizes.

The use of graph chromosomes is well established [2, p. 265]. Of particular relevance to our work is research conducted by Poli [21, 22]. He evolved computer programs represented by graphs and he used the topology of his graphs to devise evolutionary operators. Poli uses planar graphs, whereas our A-type graphs need not be planar. Poli’s crossover operators exchanged connected subgraphs of graphs of parents, as do our crossover operators, although we require that our subgraphs be a radial set. To our knowledge these graph-theoretic ideas have not previously been used to devise evolutionary operators for A-types.

4.5 Fitness function

We use a standardized (and normalized) fitness function. That is, our fitness function returns a real number between 0 and 1, inclusive. The lower the fitness of an A-type, the more fit that A-type is. In this section we define our fitness function.

Recall from the start of Sect. 4 that we use our EA to search for ‘suitably small’ A-types that represent a particular function $f$. We require training data $T$ that is a set of input-output pairs of $f$. That is, $T = \{(x_i, f(x_i))\}$ where $i$ is an element of some index set. We call each pair in $T$ a

| Table 4 Our A-type crossover operator |
|---------------------------------------|
| Crossover                             |
| 1. The child $C$ is assigned simply to be a copy of the parent $\emptyset$; that is, $C \leftarrow \emptyset$ |
| 2. A subgraph of $C$ is chosen; we call this the acceptor $A$ |
| 3. A subgraph of $\emptyset$ is chosen; we call this the donor $D$ |
| 4. The subgraph $A$ is removed from $C$ (any arrows bridging $(C - A)$ and $A$ are also removed) and a copy of $D$ is inserted into $C$ |
| 5. Arrows are added to $C$ so that $C$ is a valid A-type: |
| (a) Inserting arrows from $C - A$ to $D$. For each arrow the source is randomly selected from the distal boundary of $A$ |
| (b) Inserting arrows from $D$ to $C - A$. For each arrow the source is randomly selected from the proximal boundary of $D$ |

Fig. 10 A concrete example of our crossover operator. The numbers inside the nodes are labels for the nodes. a The mother $\emptyset$ and its acceptor subgraph $A = \{3\}$. Note that the proximal boundary $= \{3\}$, and the distal boundary $= \{1, 2, 4, 5\}$. b The father $\emptyset$ and its donor subgraph $D = \{1', 2', 3', 4', 5', 6'\}$. The proximal boundary $= \{1', 5', 6'\}$, and the distal boundary $= \{0', 7'\}$. c Inserting $D$ into $\emptyset - A$. d Inserting arrows from $\emptyset - A$ to $D$. For each inserted arrow the source is randomly selected from the distal boundary (shaded nodes). e Inserting arrows from $D$ to $\emptyset - A$. For each inserted arrow the source is randomly selected from the proximal boundary (shaded nodes).
training example. We also require an upper value \( u \) for A-type sizes: A-types larger than \( u \) are considered unsuitable solutions. We call \( u \) the penalty bound. Note that in our algorithms, we always take the value of \( u \) to be equal to the upper bound of the size of A-types in the initial population (see Sect. 4.3).

Consider a candidate solution \( A \). We determine the fitness of \( A \) as follows.

1. **Determining the performance of \( A \) with respect to \( T \).** Let \( A(x_i) \) denote the output of \( A \) given an input \( x_i \). For each training example \((x_i, f(x_i))\) we calculate the normalized Hamming distance between \( A(x_i) \) and \( f(x_i) \). Let \( d \) denote the average of all of these Hamming distances.

2. **Including a penalty if \( A \) is larger than \( u \).** Choose a positive real number \( m \), which we call the pressure gradient. If \(|A| \leq u \) then \( A \)'s fitness is \( d \). Otherwise the fitness of \( A \) is the minimum of \( \{1, dm(|A| - u + 1)\} \).

   Thus our fitness function is a continuous piecewise function \( g \). It is initially constant with \( g = d \), then linear with a gradient \( m \), then constant with \( g = 1 \). This enables us to ‘pressure’ the population so that it is unlikely to contain A-types of size much greater than \( u \).

### 4.6 Selection rules

In our EA, for each generation we have three operations which require A-types to be selected from the population: crossover, mutation, and elimination. In this section we explain how we perform the selections.

For crossover our parent selection is a fitness proportional selection. The fitter the A-type the greater the probability that that A-type is selected as a parent. A-types are chosen by their fitness weighted by a function \( h \); we chose \( h \) to be an exponential. The choice of \( h \) was the same for all the tasks we considered.

For the elimination operation A-types are also chosen by their fitness weighted by an exponential. However, the less fit A-types are more likely to be chosen for elimination.

For mutation our selection operator is random.

### 4.7 Implementation

#### 4.7.1 Candidate solutions

When our EA searches for an A-type \( A \) that represents a given concept function it has to search for both the graph of \( A \) and the delay time \( \delta \) of \( A \). For each candidate solution the EA chooses an A-type graph, estimates a range of possible delays for that graph, and determines the fitness of each (graph + delay) A-type. That is, each candidate solution consists of a set of A-types all with the same underlying A-type graph but with different delays coming from some interval. So in our algorithm descriptions when we say that we make an A-type we are actually making a set of A-types. We chose this implementation because it is easy to code and efficient to run.

#### 4.7.2 Estimating a range of delays

When our EA constructs an A-type graph \( G \) (either a randomly constructed graph for the initial population, or the result of crossover or mutation) it must estimate a suitable range of delays for \( G \). Let \( N \) denote the number of nodes in \( G \). Let \( A \) denote an A-type with graph \( G \) and a delay time \( \delta = 0 \). The larger the range of delays for each individual, the longer it takes to train each individual. We take a somewhat pragmatic approach to estimate the range of delays. To estimate the minimum delay we perform the following four steps. First, we input a random sequence of vectors into \( A \). We collect the output vectors from \( A \) and call this sequence \( S_{out} \). Second, we repeat the above step, yielding a second output sequence \( S'_{out} \). Third, we determine the position \( q \) where \( S_{out} \) and \( S'_{out} \) first differ (if \( S_{out} = S'_{out} \) then we set \( q = -1 \)). This gives a reasonable estimate of the minimum possible time for data to percolate through the network from the input nodes to the output nodes. Fourth, we subtract the sum of the input dimension of \( A \) and the output dimension of \( A \) from \( q \). If \( q \) is negative then we set it to zero. Our estimate of the minimum delay is \( q \). We take the maximum delay to be \( N \); this gives a reasonable estimate of the maximum possible time it can take data to percolate through the network from the input nodes to the output nodes.

### 5 Simulations

To investigate the performance of our EA we implemented the algorithm using Java and ran many simulations with it. Here we describe our simulations and present our results. Further detail can be found in [20].

#### 5.1 Experimental method

Our simulations investigated the performance of our EA. We concentrated on two main questions: whether our implementation of an evolutionary search is useful, and whether our crossover operator aids our EA.

##### 5.1.1 Comparing algorithms

We compared three algorithms: a blind search, a mutation-only EA, and an EA with crossover. First, we employed a blind search. This algorithm simply creates a random
A-type, and checks whether it is a solution; if it is not then it is destroyed and the process is repeated. This is a very special case of our EA; however, each candidate solution is entirely independent of all previous candidate solutions—in the blind search all hereditary information is lost from one generation to the next. Second, we employed a mutation-only EA. Asexual evolution is seen in biology and it is a straightforward special case of our EA—we simply ensure that no crossovers are performed. Comparing our EA to the mutation-only special case offers a test of the efficacy of our crossover operator. Third, we employed our EA in its entirety. We name these algorithms blind_search_one, mutation_search_one, and genetic_search_one respectively.

5.1.2 Benchmark learning tasks

To assess the performance of our EA we chose three simple supervised learning tasks. These tasks involved searching for A-types that represent simple classes of functions: \( n \)-identity, \( n \)-multiplexer and \( n \)-carry. Their simplicity allowed us to investigate performance of our algorithm as the complexity of the problem is scaled up. Also, it is easy to write down exact solution A-types for each task investigated. In Sects. 5.3–5.5 we describe each task and the performance of our EA as it searches for that task. In this section we give further details of our experimental method.

5.1.3 What we measured

To gauge the performance of our algorithms we conducted several trials. For each trial we recorded the number of attempts required for a solution to be discovered: that is, the total number of A-types constructed in the initial population, via mutation and via crossover. Note that mutation_search_one constructs one new A-type in each generation (by mutation), whereas genetic_search_one constructs two or more (by mutation and crossover). For this reason we count the number of attempts rather than the number of generations.

Each learning task that we consider is a class of concept functions parametrised by a positive integer \( n \) (usually \( n \) is just the input dimension). For each value of \( n \) we employed three algorithms and with each algorithm we conducted many trials. To display our results we present a plot of \( n \) versus attempts required. A data point on these plots represents an average of all trials for a particular algorithm searching for a particular concept function. For all trials associated with one data point we employ Student’s \( t \) test (for instance see [15, sec. 24.6]) to determine a 90% confidence interval. This determines the error bars displayed around each data point. We assume that our results are normally distributed, as is required for the \( t \) test to be valid.

5.1.4 Suitable training data

Although we define A-types to accept and return sequential data, two of the three concepts that we searched for are tasks that require A-types to be operated in the clamped input mode. When we consider \( n \)-identity and \( n \)-multiplexer concepts we do so with clamped examples. This makes our investigations computationally easier. Conducting numerous trials with several \( n \) values is very computationally expensive if we search for A-types that operate in the more general sequential mode. In order to test our EA with A-types that operate in the sequential mode, we also devised a sequential task, namely \( n \)-carry.7

Performing searches with long training examples takes a long time; performing searches with short examples usually leads to inexact solutions. Mindful of this we adopted the following procedure. We chose relatively short training examples to discover possible solutions, then tested these possible solutions with longer training examples. If a possible solution fits these longer training examples then we deem it to be an exact solution (see below for more details).

When we searched for clampable A-types that represented a function \( f \), we used a training set containing all possible examples \( (x, f(x)) \) such that \( f(x) \) is a sequence of three vectors.8 That is, when the fitness of a candidate solution \( A \) was assessed with an example \( (x, f(x)) \), the sequence containing the first three Boolean vectors output by \( A \) was compared with the sequence \( f(x) \). For example, Fig. 11 shows the training set that we used when we searched for 2-identity.

When we search for clampable A-types that represent a Boolean function \( f \) we define an exact solution as follows. Let \( x_0 \) denote a Boolean vector in the domain of \( f \). An A-type \( A \) is an exact solution to \( f \) if when \( x_0 \) is input into \( A \), the constant sequence \( f(x_0) \) is returned by \( A \) for \( t \) moments, where \( t \) is some large but fixed positive integer. That is, the output nodes of \( A \) have constant value \( f(x_0) \) for \( t \) moments starting from moment \( \delta \). When searching for A-types that represent clamp \( n \)-identity and clamp \( n \)-multiplexer, we deemed solutions to be exact when \( t = 1,000 \).

When searching for A-types that represented a sequential function, our training data contained a single example \( (x, f(x)) \), where \( x \) was a random sequence of Boolean vectors. We chose \( x \) to be short so that solutions would often be found relatively quickly. As with the clamped case, to cater for the chance that a discovered solution is

---

7 Note that we use A-types with delay nodes for all three learning tasks. However, it can be shown that there exist A-types without delay nodes that represent clamp \( n \)-identity and clamp \( n \)-multiplexer; see Fig. 13 for \( n = 1 \) and \( n = 2 \).

8 With the exception of \( n \)-identity when \( n \in \{7, 8, 9, 10\} \); see Sect. 5.3.
incorrect we defined exact solutions for sequential searches. We deemed a solution to be exact if it represents a training example \((x, f(x))\) where \(x\) consists of a random sequence of \(10^4\) Boolean vectors.

5.1.5 Other search parameters

For each of the three algorithms tested there are several parameters that require arguments; for instance, the population size, and the mutation to crossover ratio. To optimize each algorithm we need to search for appropriate arguments; furthermore, these arguments may be specific to each benchmark concept. We performed some rather informal investigations to decide upon arguments for these parameters. Those common to all three tasks are presented in Table 5. Further details are presented as we introduce the investigations for each concept.

5.1.6 Task management

We conducted our investigations using many cores of numerous computers. Consequently, we had to minimize any bias that this introduced into our results. For each learning task we considered a set of concept functions, each of which had a particular value of \(n\). When we searched for a concept function \(f\) we used a set of a suitable number of training examples \(\{x, f(x)\}\). In the cases where we did not use an exhaustive set of training examples, we randomly selected a suitably sized training set from all possible examples. However, we ensured that the training examples remained constant as the training algorithms varied. That is, when we searched for an A-type representing a concept function \(f\), the \(i\)th trial using each algorithm had the same set of training examples. The processing time may vary from computer to computer, but the number of attempts required should remain constant.

For both \(n\)-identity and \(n\)-multiplexer, for each integer \(n\) tested we performed at least twenty trials for each algorithm. The exception to this is for some blind searches, because on occasions the blind search took an excessively long time to complete. We note below when twenty trials were not performed for the blind search.

5.2 Actual solutions

In this section we give examples of solutions obtained by our algorithms. In Fig. 12 we present some of the solutions found when we employed \texttt{mutation_search_one} to search for clamped identity function with one input and one output (clamped 1-identity function in the language of Sect. 5.3). The details of the search are given in Sect. 5.3. We found simple solutions without delay nodes; see Fig. 12a. We found solutions with subgraphs that did not contribute to the output of the solutions; see Fig. 12b, c. Note that such subgraphs may be considered ‘junk’; however, A-types with such subgraphs may prove to be useful intermediary forms in an algorithm based on a population of A-types. In Sect. 3.3.2 we explained that there always exists an A-type without delay nodes that represents a given clamped function. However, for all simulations in this section we used A-types with delay nodes. Consequently, we found solutions that involve delay nodes; see Fig. 12d. Because A-types that represent clamped functions do not require the synchronisation of data, we found solutions having paths of differing lengths from the input node to the output node; see Fig. 12e, f.

5.3 Searching for clamped \(n\)-identity

The first class of concept functions that we consider is clamped \(n\)-identity. Given a positive integer \(n\), \(n\)-identity is

\[
\begin{align*}
&\left( \left[ \begin{array}{c}
1 \\
1
\end{array} \right], \left[ \begin{array}{c}
1 & 1 & 0 \\
1 & 1 & 1
\end{array} \right] \right) \\
&\left( \left[ \begin{array}{c}
1 \\
0
\end{array} \right], \left[ \begin{array}{c}
1 & 0 & 1 \\
0 & 1 & 0
\end{array} \right] \right) \\
&\left( \left[ \begin{array}{c}
0 \\
1
\end{array} \right], \left[ \begin{array}{c}
0 & 0 & 1 \\
1 & 1 & 1
\end{array} \right] \right) \\
&\left( \left[ \begin{array}{c}
0 \\
0
\end{array} \right], \left[ \begin{array}{c}
0 & 0 & 0 \\
0 & 0 & 0
\end{array} \right] \right)
\end{align*}
\]

Fig. 11 The training set that we used for our searches for 2-identity. Note that this training set is exhaustive in that this set contains all possible examples of 2-identity that have output sequences with length \(l = 3\).

Table 5 Parameters common to all to three investigations in this section

| Parameter                        | Argument |
|---------------------------------|----------|
| When using any algorithm        |          |
| Population size                 | 100      |
| Worst fitness of a solution     | 0.00     |
| Probability that a node is       | 20%      |
| constructed as a delay node | | |
| Penalty bound                   | \(u\)    |
| Pressure gradient               | \(\frac{1}{2}\) |
| When using \texttt{genetic_search_one} |          |
| Crossovers per generation       | 1        |
| Mutations per generation        | 1        |
| Upper bound of size (\% of      | 80%      |
| internal nodes of parent)       |          |
| of donor or acceptor subgraphs  |          |
| for crossover                   |          |

Note that the we set penalty bound \(u\) equal to the upper bound for the size of A-types in the initial population. This is specific to each learning task.

The number of crossovers and number of mutations per generation are allowed to vary in Sect. 5.6.
the Boolean function $f_{id}$ from $S_n$ to $S_n$ that maps each $n$-component Boolean vector to itself. In Fig. 13 we illustrate two examples (found by inspection) of A-types that represent clamped $n$-identity—note that these also represent the more general function columnwise $n$-identity.

In this section we describe our searches for A-types that represent $n$-identity for values of $n$ that range from 1 to 10. In Table 6 we list the arguments that we chose for this search. When we searched for $n$-identity we employed all examples with output sequences of length 3 unless there were more than 100 of these (this was the case when $n \in \{7, 8, 9, 10\}$). In the latter case we randomly chose 100 examples for each trial. Our choice of training data almost always gave exact solutions and, as described above, we ensured that this choice was not a variable when we compared our algorithms.

In Fig. 14 we compare the performances of blind_search_one and mutation_search_one when searching for A-types that represent $n$-identity with $n$ ranging from 1 to 10. Note that we do not display results for blind_search_one for $n > 4$. This is because all trials using blind_search_one failed to find a solution within $10^9$ attempts. These results show that mutation_search_one outperforms blind_search_one by orders of magnitude.

In Fig. 15 we compare the performances of mutation_search_one and genetic_search_one when searching...
5.4 Searching for clamped \( n \)-multiplexer

The second class of concept functions that we consider is clamped \( n \)-multiplexer. An A-type \( A \), with delay \( \delta \), that represents \( n \)-multiplexer has \( n \) regular input nodes \((x_1, \ldots, x_n)\) and \( \log_2(n) \) (rounded up to the next integer) extra input nodes called selector pins \( s_j \). Consider the input on the selector pins of \( A \) at moment \( t \). This gives a binary representation of some integer \( i \). At moment \((t + \delta)\) the output of \( A \) is equal to the value of \( x_i \) at moment \( t \).

Several researchers have applied EAs to the task of discovering multiplexers. This started with Wilson [32] and others have also investigated this task, for example Koza [14, ch. 7], Butz [5, ch. 3]. In particular, Bull and Preene [4] used simulated evolution to design clampable A-types that represent clamped \( n \)-multiplexers although \( n \)-multiplexer is more complex than \( n \)-identity, it is another class of problem that scales easily.

In this section we describe our searches for A-types that represent \( n \)-multiplexer for values of \( n \) that range from 2 to 5. In Table 7 we list the arguments that we chose for this search. In Fig. 16 we illustrate A-types (found by inspection) that represent \( n \)-multiplexer where \( n \in \{2, 3\}\) —note that these also represent columnwise \( n \)-multiplexer.

In Fig. 17 we compare the performances of \( \text{blind\_search\_one} \) and \( \text{mutation\_search\_one} \). When we used \( \text{blind\_search\_one} \) to search for A-types that represent 3-multiplexer, only two of the twenty trials returned a solution (before \( 10^8 \) attempts). We include the data point corresponding to \( n = 3 \) for \( \text{blind\_search\_one} \) as a lower bound; that is, we expect that had we allowed a greater maximum number of generations, the point corresponding to 3-multiplexer for \( \text{blind\_search\_one} \) would be greater than...
In conclusion, the results in this section provide evidence that when our EA searches for n-carry it significantly outperforms a blind search. They also provide evidence that our EA searches for n-carry for values of n that range from 1 to 8. For each n-carry search we chose a training example with a random input sequence of length 50. For each value of n we conducted 20 trials per algorithm and the training example for the ith trial was the same for all algorithms. In Table 8 we list the arguments that we chose for this search.

In Fig. 21 we compare the performances of blind_search_one and mutation_search_one as they search for n-carry, for n ranging from 1 to 8. Note that when using blind_search_one all trials for n > 4 failed to find a solution. From these two figures we see that mutation_search_one significantly outperforms blind_search_one.

In Fig. 22 we compare the performances of genetic_search_one and mutation_search_one as they search for n-carry. These results show that genetic_search_one significantly outperforms mutation_search_one.

In conclusion, the results in this section provide evidence that when our EA searches for n-carry it significantly outperforms a blind search. They also provide evidence that:

\[
\left(\begin{array}{c}
[1], [0], [1] \\
[0], [1], [0]
\end{array}\right)
, \quad
\left(\begin{array}{c}
[1], [0], [0] \\
[0], [1], [0]
\end{array}\right)
, \quad
\left(\begin{array}{c}
[1], [1], [1] \\
[1], [1], [1]
\end{array}\right)
, \quad
\left(\begin{array}{c}
[1], [0], [0] \\
[1], [0], [0]
\end{array}\right)
, \quad
\left(\begin{array}{c}
[0], [0], [1] \\
[0], [1], [0]
\end{array}\right)
\]

Fig. 19 Four input–output pairs of 2-carry

considered: it does not consist of columnwise Boolean functions. We devised this class of functions to investigate a sequential task that has no clamped analogue. We call this class of functions n-carry. Informally, n-carry maps a single bit string to a set of n bit strings; each of these output strings is a segment of the input string. More formally, for some positive integer n and some integer l ≥ n, n-carry is a function fn from S1,l to Sn,(l-n+1), such that each sequence x = ([a1], [a2], ..., [an]) is mapped to

\[f_n(x) = \left(\begin{array}{c}
a_{l-n+1} \\
a_{l-n} \\
\vdots \\
a_1 \\
a_{l+1} \\
a_n
\end{array}\right)\]

For example in Fig. 19 we show four input-output pairs for 2-carry. In Fig. 20 we illustrate two examples (found by inspection) of A-types that represent n-carry.

We searched for A-types that represent n-carry for values of n that range from 1 to 8. For each n-carry search we chose a training example with a random input sequence of length 50. For each value of n we conducted 20 trials per algorithm and the training example for the ith trial was the same for all algorithms. In Table 8 we list the arguments that we chose for this search.

In Fig. 21 we compare the performances of blind_search_one and mutation_search_one as they search for n-carry, for n ranging from 1 to 8. Note that when using blind_search_one all trials for n > 4 failed to find a solution. From these two figures we see that mutation_search_one significantly outperforms blind_search_one.

In Fig. 22 we compare the performances of genetic_search_one and mutation_search_one as they search for n-carry. These results show that genetic_search_one significantly outperforms mutation_search_one.

In conclusion, the results in this section provide evidence that when our EA searches for n-carry it significantly outperforms a blind search. They also provide evidence

\[\text{Fig. 17} \quad \text{Searching for A-types that represent n-multiplexer with blind_search_one and mutation_search_one. Here we show the average number of attempts required before a solution was discovered.}\]

\[\text{Fig. 18} \quad \text{Searching for A-types that represent n-multiplexer with genetic_search_one and mutation_search_one. Here we show the average number of attempts required before a solution was discovered.}\]
that when our EA searches for \( n \)-carry our crossover operator aids our EA.

5.6 Parameter bias

The above results suggest that our crossover operator is useful; however, we must be mindful that \texttt{genetic_search_one} has many parameters that require arguments for a particular search. Because our investigations were a ‘proof of concept’ we simply chose parameter values that ensured that we found solutions. These values were held constant as we varied the algorithms.

We did investigate the effect of varying the \((\text{crossovers per generation}) : (\text{mutations per generation})\) ratio in \texttt{genetic_search_one} when searching for \( n \)-carry. The other parameter values for these simulations were those specified in Tables \ref{tab:parameters} and \ref{tab:parameters2}. The results are presented in Fig. \ref{fig:7-carry}. Having a \((\text{crossovers per generation}) : (\text{mutations per generation})\) ratio of 1:1 gave optimal performance. Note that in the special case when the ratio is 0:1, \texttt{genetic_search_one} is effectively the same as \texttt{mutation_search_one}.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Parameter & Argument \\
\hline
Lower bound for size of initial machines & \( 3 + 2(n - 1) \) \\
Upper bound for size of initial machines & \( 3 + 2^n \) \\
Maximum number of attempts & \( 10^9 \) \\
Trials per training example & \( 20^a \) \\
Length of exact solution & \( 10^4 \) \\
\hline
\end{tabular}
\caption{Parameters used for our sequential \( n \)-carry searches}
\label{tab:parameters}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Parameter & Argument \\
\hline
Lower bound for size of initial machines & \( 3 + 2(n - 1) \) \\
Upper bound for size of initial machines & \( 3 + 2^n \) \\
Maximum number of attempts & \( 10^9 \) \\
Trials per training example & \( 20^a \) \\
Length of exact solution & \( 10^4 \) \\
\hline
\end{tabular}
\caption{Parameters used for our sequential \( n \)-carry searches}
\label{tab:parameters2}
\end{table}

5.7 Is our crossover simply macromutation?

The above results provide evidence that our A-type crossover operator is useful. However, we have yet to
investigate whether this is simply because our crossover operator is a ‘macromutator’; that is, whether our crossover operator is only useful because it mixes the population more effectively than our mutation operators. We turn to this question now.

The results from the $n$-identity searches and the $n$-carry searches demonstrate that for some tasks the crossover of our EA is useful. In many EAs crossover is useful because it provides sudden large variation in the population, rather than because it recombines individuals [2, ch. 6]. Such an operator is called a macromutation operator. This is not the case in biology: the utility of biological crossover is due to its ability to recombine individuals’ information [16, p. 276].

The ‘headless chicken’ search offers a relatively simple means of testing whether a crossover operator is simply acting as a macromutator [11, 23]. The headless chicken search is an EA where only one parent is selected from the population and the other parent is an entirely new individual [2, p. 153]. We implemented the headless chicken algorithm by duplicating \texttt{genetic\_search\_one} with the following modification. For each crossover, after we have selected the parents $P_1, P_2$ we randomly choose one parent $P$ and then construct a random A-type $P'$ that is the same size as $P$. We then perform the crossover using $P'$ and the other parent.

We compare \texttt{genetic\_search\_one} and our headless chicken search for clamped $n$-identity and $n$-carry, the benchmark tasks that demonstrated the utility of our crossover. Figure 24 shows that when searching for clamped $n$-identity, \texttt{genetic\_search\_one} outperforms our headless chicken search. Figure 25 shows that when searching for $n$-carry, \texttt{genetic\_search\_one} also outperforms our headless chicken search.

This provides evidence that for some tasks our crossover operator is more useful than a macromutation operator.

5.8 Size bias

We now briefly turn to the size of solutions obtained by different algorithms. Consider $n$-carry, for example. The graph in Fig. 26 shows that there is not a great difference between the solution sizes found by \texttt{mutation\_search\_one} and \texttt{genetic\_search\_one}. Hence the difference in performance of these algorithms is not due to size differences in the populations.

More generally, one can consider the diversity of the population as the algorithm progresses. It can be seen from Fig. 26 that the algorithms found solutions of different sizes for each fixed value of $n$; in particular, these solutions were not all the same. This indicates the presence of at least some diversity. We did not investigate population diversity systematically. See also Fig. 12, which shows a sample of solutions obtained by using \texttt{mutation\_search\_one} to search for A-types that represent 1-identity.
Recall from Sect. 4.4 that our method for fitness-based selection employs an exponential function. This strongly favours fitter individuals, which may reduce the diversity of the population. Our choice of exponential sufficed for our algorithm comparisons. One advantage of our method for fitness-based selection is that it would be easy to vary: one can replace the exponential with any other monotone function.

6 Conclusion

We devised a graph-based EA for finding A-types that represent a given function. When applied to the three benchmark problems, the EA performed considerably better than a purely random search. For clamped $n$-identity and $n$-carry, the full version of the EA performed better than the mutation-only version. Our algorithm worked in both the clamped and the sequential settings.

We now suggest directions for future research. A-types are relatively simple, yet they are recurrent Boolean ANNs capable of representing any Boolean function and operating in a sequential mode. Consequently, we believe A-types are a useful tool for investigating the learning and behaviour of Boolean ANNs. In particular, the simplicity of A-types means that manipulations of their graphs are often straightforward to implement. We suggest two areas of future research with A-types: further investigations into evolutionary techniques, and using the symmetry of a concept function to improve the search for an A-type that represents that function.

6.1 Evolving evolutionary operators

Here we propose that it is worthwhile to continue to search for useful evolutionary operators for A-types. Furthermore, we propose that evolutionary searches can be applied to discover these operators. The evolution of parameters of a search is an established technique in evolutionary computing [9, ch. 4]. Many researchers have extended this idea to include the evolution of evolutionary operators [26]. In terms of evolving networks, Teller’s research [27, 28] is of particular interest. Teller solved signal classification tasks by evolving two populations simultaneously. One population was a set of programs, which were represented with graphs. The other population was a set of evolutionary operators that operated on the programs. We believe that it would be worthwhile to co-evolve evolutionary operators in a manner analogous to Teller’s research. This would allow a more complete investigation of what happens when one varies the many parameters in our EA.

The results in Sect. 5 show that, for some problems, our crossover operator is more useful than a macromutation operator. Although our crossover operator employs relatively simple graph-theoretical ideas, its implementation is rather involved. By evolving evolutionary operators for A-types, one may be able to find more complicated but better-performing A-type crossover operators and test whether certain properties (such as the out-degree of nodes, connectedness of subgraphs, network activity, and perhaps some measure of symmetry) are useful.

6.2 Making use of symmetry

The notion of symmetry, which is made precise by group theory, leads to useful problem-solving techniques. Consider the A-type shown in Fig. 7, which represents columnwise Exclusive-OR. This function is symmetric in its arguments: that is, $A \oplus B = B \oplus A$ for all $A$ and $B$. In Fig. 27 we redraw this A-type to show that it has ‘mirror symmetry’ about a horizontal line. So columnwise Exclusive-OR has a symmetry; when searching for an A-type that represents it, both the concept function and one of its solutions share this property. We hypothesise that this idea can be formalised using group theory for a class of concept functions admitting a symmetry and used to cut down the size of the search space of an EA. This is work in progress.

Recently Kondor [13] investigated the use of group-theoretic methods to improve some modern machine learning techniques. Other researchers have also applied symmetries to ANNs for this purpose [1, 25, 33]. Recently Dong and Zhang [8] incorporated group-theoretic techniques into EAs with populations of ANNs, using relatively

\[ \text{Loosely, we can define the activity of a node as the average number of changes of state per moment it undergoes when a large random data packet is processed by the network. Furthermore, we can define the activity of a subgraph of an A-type as an average of the activity of all nodes in that network. Note that Teuscher [29, ch. 5] defines activity of A-types and uses this to investigate the non-linear dynamics of these networks.} \]
simple operators. The simplicity of A-types makes them a good setting in which to further implement and test the application of group-theoretic ideas on a population of evolving ANNs.

Acknowledgments The first author wishes to thank the Department of Physics and Astronomy, University of Canterbury, for financial support. We thank the referees for their thorough reading of the paper and their helpful suggestions for improving it. We are grateful to Jack Copeland for many stimulating discussions, Paul Brouwers and Orlon Petterson for technical support, Larry Bull for helpful comments, and Mike Reid for general advice and encouragement. Nikolai Krueitzmann gave valuable feedback on an earlier draft of this paper.

References

1. Baldi P (1987) Symmetries and learning in neural network models. Phys Rev Lett 59(17):1976–1978
2. Banzhaf W, Nordin P, Keller R, Francone F (1998) Genetic programming: an introduction. Morgan Kaufmann, San Francisco
3. Bull L (2009) On dynamical genetic programming: simple Boolean networks in learning classifier systems. Int J Parallel Emergent Distrib Syst 24(5):421–442
4. Bull L, Preen R (2009) On dynamical genetic programming: random Boolean networks in learning classifier systems. Lect Notes Comput Sci 5481:37–48
5. Butz M (2002) Anticipatory learning classifier systems. Kluwer, Dordrecht
6. Copeland B, Proudfoot D (1996) On Alan Turing’s anticipation of connectionism. Synthese 108(3):361–377
7. Copeland B, Turing A (2004) The essential Turing: seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life plus the secrets of Enigma. Oxford University Press, USA
8. Dong J, Zhang J (2006) Detection of the permutation symmetry in pattern sets. Discrete Dyn Nat Soc 2006. http://www.hindawi.com/journals/ddns/2006/081503/ref/
9. Eiben A, Smith J (2003) Introduction to evolutionary computing. Springer, Berlin
10. Floreano D, Diur P, Mattiussi C (2008) Neuroevolution: from architectures to learning. Evol Intell 1(1):47–62
11. Jones T (1995) Crossover, macromutation, and population-based search. In: Proceedings of the 6th international conference on genetic algorithms, pp 73–80
12. Kauffman S (2000) Investigations. Oxford University Press, USA
13. Kondor R (2008) Group theoretical methods in machine learning. Ph.D. thesis, Columbia University
14. Koza J (1992) Genetic programming: on the programming of computers by means of natural selection. The MIT Press, Cambridge
15. Kreyszig E (1994) Advanced engineering mathematics, 7th edn. Wiley, New York
16. MacKay D (2003) Information theory, inference, and learning algorithms. Cambridge University Press, Cambridge
17. Mc Culloch W, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. Bull Math Biol 5(4):115–133
18. Minsky M (1967) Computation: finite and infinite machines. Prentice-Hall, Upper Saddle River
19. Mitchell TM (1997) Machine learning. McGraw-Hill, New York
20. Orr EP (2010) Evolving Turing’s artificial neural networks. Ph.D. thesis, University of Canterbury, Department of Physics and Astronomy
21. Poli R (1996) Parallel distributed genetic programming. Tech. rep., University of Birmingham
22. Poli R (1997) Evolution of graph-like programs with parallel distributed genetic programming. In: Proceedings of 7th international conference on genetic algorithms, pp 346–353
23. Poli R, McPhee N (2002) Exact GP schema theory for headless chicken crossover and subtree mutation. In: Proceedings of the 2001 congress on evolutionary computation, 2001, vol 2. IEEE, pp 1062–1069
24. Rohlf T, Bornholdt S (2009) Self-organized criticality and adaptation in discrete dynamical networks. Adapt Netw 51:73–106
25. Shawe-Taylor J (1993) Symmetries and discriminability in feedforward network architectures. IEEE Trans Neural Netw 4(5):816–826
26. Tang W, Wu Q (2009) Biologically inspired optimization: a review. Trans Inst Meas Control 31:495–515
27. Teller A (1996) Evolving programmers: the co-evolution of intelligent recombination operators. In: Advances in genetic programming, MIT Press, pp 45–68
28. Teller A, Veloso M (1995) Program evolution for data mining. Int J Expert Syst 8(3):213–236
29. Teuscher C (2001) Turing’s connectionism: an investigation of neural network architectures. Springer, Berlin
30. Teuscher C, Sanchez E (2001) Self-organizing topology evolution of Turing neural networks. Lect Notes Comput Sci 2130:820–826
31. Turing A (1948) Intelligent machinery. Tech. rep., National Physical Laboratory
32. Wilson SW (1987) Classifier systems and the animat problem. Mach Learn 2(3):199–228
33. Wood J, Shawe-Taylor J (1996) A unifying framework for invariant pattern recognition. Pattern Recogn Lett 17(14):1415–1422