www.palgrave-journals.com/jors/

Automated code generation by local search

MR Hyde*, EK Burke and G Kendall
University of Nottingham, Nottingham, UK

There are many successful evolutionary computation techniques for automatic program generation, with the best known, perhaps, being genetic programming. Genetic programming has obtained human competitive results, even infringing on patented inventions. The majority of the scientific literature on automatic program generation employs such population-based search approaches, to allow a computer system to search a space of programs. In this paper, we present an alternative approach based on local search. There are many local search methodologies that allow successful search of a solution space, based on maintaining a single incumbent solution and searching its neighbourhood. However, use of these methodologies in searching a space of programs has not yet been systematically investigated. The contribution of this paper is to show that a local search of programs can be more successful at automatic program generation than current nature inspired evolutionary computation methodologies.

Journal of the Operational Research Society (2013) **64,** 1725–1741. doi:10.1057/jors.2012.149
Published online 5 December 2012

Keywords: heuristics; genetic programming; local search

1. Introduction

In 1992, John Koza published his book 'Genetic programming: on the programming of computers by means of natural selection' (Koza, 1992), which presented a methodology and a vision for automatic programming. This vision was based on the use of biologically inspired evolutionary computation. The research field created around this idea has been very successful, and has obtained many important results (see Poli *et al* (2008) for an overview), even producing designs which infringe on existing patents (Koza and Poli, 2005). More recently, grammatical evolution has become popular, which extends the biological metaphor by clearly separating the genotype and phenotype (O'Neill and Ryan, 2001, 2003). The 'ADATE' system (Automatic Design of Algorithms Through Evolution) also draws on biological concepts to generate algorithms automatically (Løkketangen and Olsson, 2010).

Such methods for automatic program synthesis are important because they assign some of the time-consuming search process to the computer. Instead of manual creation or tuning of an algorithm, the computer performs the search through a space of programs.

This research area could represent a paradigm shift, eventually redefining the role of the human programmer. Instead of attempting to manually design the best program for a given problem, they would attempt to design the best search space, where good programs are likely to exist. The computer can then perform the time-consuming task of searching that space.

While evolutionary computation has attained a certain amount of success in automatic program generation, other established metaheuristic techniques have not yet received any significant research attention. They have been applied to search spaces of solutions, but not yet spaces of programs (which then solve problems). Population-based evolutionary computation has been applied to automatically build systems, but local search techniques have not. We believe that a possible reason for this is the difficulty in defining an effective neighbourhood of a program.

A contribution of this paper is to present an effective neighbourhood of a program, which is suitable for local search. We also present a local search methodology which we demonstrate to be successful at searching the neighbourhood for superior programs. Our approach is tested on a set of classic problems in artificial intelligence that have been used in the literature previously to showcase the effectiveness of evolutionary computation techniques. The local search methodology finds the required program more often, and faster, than grammatical evolution.

Single point local search is much more akin to the process performed by a human programmer in order to create a program. Single changes are made to 'draft' programs that seem to work well but are not quite complete, and if those changes create an improvement then further changes are made to the new program. It is rare that a programmer would consider changing many parts of

Correspondence: MR Hyde, Tyndall Centre for Climate Change Research, School of Environmental Sciences, University of East Anglia, Norwich, NR4 7TJ, UK.

a program at once, which is the effect of the 'crossover' operator in grammatical evolution. This is because a human programmer would usually wish to analyse the effects of each individual change. Indeed, in many areas of computer science (programming being a prime example), it is considered good practice to only modify one element at a time, in order to observe its effect.

Of course there are many successes of evolutionary computation in the literature, and we are not suggesting that local search will be superior to population-based approaches in every case. We are suggesting that local search should be researched in parallel with population-based approaches in the pursuit of effective automatic programming methodologies.

2. Related work

Because of the popularity of genetic programming as an automatic programming methodology, this section consists mainly of previous work from this area. We will not explain genetic programming in detail in this paper, but we point the reader to the following texts for good overviews; Koza (1992), Banzhaf *et al* (1998), Koza and Poli (2005), Poli *et al* (2008). Grammatical evolution (O'Neill and Ryan, 2001, 2003) is a grammar-based form of genetic programming. Because it is closely related to our local search methodology, we explain it extensively in Section 4.

The ADATE (Automatic Design of Algorithms Through Evolution) system (Olsson, 1995; Løkketangen and Olsson, 2010) automatically generates code in a subset of the programming language ML. The name of the methodology implies a close similarity to genetic algorithms and, indeed, it maintains a population of candidate algorithms, and employs transformation operators which could be described as 'mutation' operators. However, ADATE is arguably more systematic in its search of programs. For example, it is more successful at generating recursive structures and iterative programs. Olsson (1995) argues that, for recursive programs, the schema theorem does not apply, and so it is unlikely that the standard genetic programming crossover operator will be successful at automatic generation of iterative and recursive programs.

A closely related area of research is 'hyper-heuristics', which have been defined as heuristics which search a space of heuristics, as opposed to a space of solutions (Burke *et al*, 2003a, b; Ross, 2005). This research field has the goal of automating the heuristic design process, and the methodology presented in this paper could also be employed in this way. Two classes of hyper-heuristic have been defined by Burke *et al* (2010b). The first is the class that intelligently chooses between complete functioning heuristics which are supplied to it by the user. The second class of hyper-heuristic automatically designs new heuristics, and further information on this class can be found in Burke *et al* (2009). The methodology presented in this paper belongs to the second class of hyper-heuristic, to the extent that the automatically generated code represents a heuristic method.

Examples of hyper-heuristics of the second class include systems which evolve local search heuristics for the SAT problem (Bader-El-Den and Poli, 2007; Fukunaga, 2008). Heuristic dispatching rules have also been evolved for the job shop problem (Ho and Tay, 2005; Geiger *et al*, 2006; Tay and Ho, 2008). Constructive heuristics for one dimensional bin packing have been evolved by genetic programming by Burke *et al* (2006, 2007a, b), showing that evolved constructive bin packing heuristics can perform better than the best-fit heuristic (Kenyon, 1996) on new problems with the same piece size distribution as those on which the heuristics were evolved (Burke *et al*, 2007a). The results by Burke *et al* (2007b) show that the evolved heuristics also maintain their performance on instances with a much larger number of pieces than the training set. It is also shown in Burke *et al* (2010a) that the evolved heuristics can be further improved if the genetic programming system can also use memory components with which to build new heuristics.

The difference between human designed heuristics and automatically designed heuristics for the three dimensional packing problem is investigated by Allen *et al* (2009). Hyper-heuristics can also evolve human competitive constructive heuristics for the two-dimensional strip packing problem (Burke *et al*, 2010c). In that study, it is shown that more general heuristics can be evolved, which are not specialised to a given problem class.

A genetic algorithm is used to evolve hyper-heuristics for the two-dimensional packing problem by Terashima-Marin *et al* (2005, 2006, 2007, 2010). The evolved individual contains criteria to decide which packing heuristic to apply next, based on the properties of the pieces left to pack. The genetic algorithm evolves a mapping from these properties to an appropriate heuristic. The work follows studies that evolve similar hyper-heuristics for the one-dimensional bin packing problem (Ross *et al*, 2002, 2003), in which the supplied local search heuristics have been created by humans, and the task of the hyper-heuristic is to decide under what circumstances it is best to apply each heuristic.

As we mentioned in Section 1, evolutionary computation is a common methodology for automatically generating search algorithms and heuristics, and methodologies for evolving evolutionary algorithms themselves are presented by Oltean (2005), Diosan and Oltean (2009). Components of existing heuristic methodologies can also be automatically designed. For example, in Poli *et al* (2005a, b), the particle velocity update rule is evolved for a particle swarm algorithm.

The idea of comparing local search systems to population-based approaches has previously been suggested by Juels and

Wattenberg (1996). Our local search methodology complements and extends this work. In their paper, they present four case studies, one of which relates to genetic programming. They present a different neighbourhood move operator for each case study. The genetic programming case employs a neighbourhood move operator which relies on the ability of any component to fit with any other component, and they obtain very good results on the boolean multiplexer problem. They state that this problem has the property that there is a direct path to the optimum, through a series of states with non-worsening fitness, and that this property helps the search process. This is an important insight, and one which we would argue is fundamental to the success of their algorithm. Unfortunately it is a property which is not present in most programming situations, because a grammar is usually involved which restricts the connectivity of the components and causes local optima. Therefore, to present local search systems as viable alternatives to genetic programming, they must incorporate the ability to utilise grammars. This paper presents such a system, which is designed to build programs from a grammar, rather than a special case where any component can fit with any other. We also show that our grammar-based neighbourhood move operator can operate over a range of code generation problems.

3. Problem domains

This section explains the problem domains that are used in this paper to assess the performance of the local search methodology. They are deliberately chosen as the standard problem domains used to analyse the success of evolutionary computation methodologies such as grammatical evolution (for example in O'Neill and Ryan, 2001, 2003).

3.1. Santa Fe ant trail

The Santa Fe ant trail problem is a standard control problem in the genetic programming and grammatical evolution literature. Its search space has been characterised as 'rugged with multiple plateaus split by deep valleys and many local and global optima' (Langdon and Poli, 1998). It has been employed many times as a benchmark to measure the effectiveness of evolutionary computation techniques (Koza, 1992; O'Neill and Ryan, 2001, 2003), and therefore it is an ideal problem with which to assess the effectiveness of this local search approach to automated programming, compared with evolutionary approaches.

The problem is to find a control program for an artificial ant, which traverses a toroidal 32 by 32 square grid. The grid contains 89 predefined locations for food items, and the ant must visit each of these locations. The ant has three actions, 'move', 'turn right', and 'turn left'. The ant has a limit of 615 actions with which to visit all of the food squares. It can also use a conditional test, which looks into the square ahead and tests if it contains food. A map of the ant trail is shown in Figure 1. The grammar used to create the control programs is shown in Figure 4, as part of the explanation of grammatical evolution.

3.2. Symbolic regression

Symbolic regression aims to find a function which successfully maps a set of input values onto their corresponding target output values. The function examined is $X^4 + X^3 + X^2 + X$, and the range of the input values is $[-1, +1]$. The grammar used to create the candidate functions is shown in Figure 2. This is an intentionally simple grammar, from which it should be easy to generate the correct code.

3.3. Even-5-parity

The even-5-parity problem aims to find a boolean expression that evaluates to true when an even number of its five inputs are true, and false when an odd number of

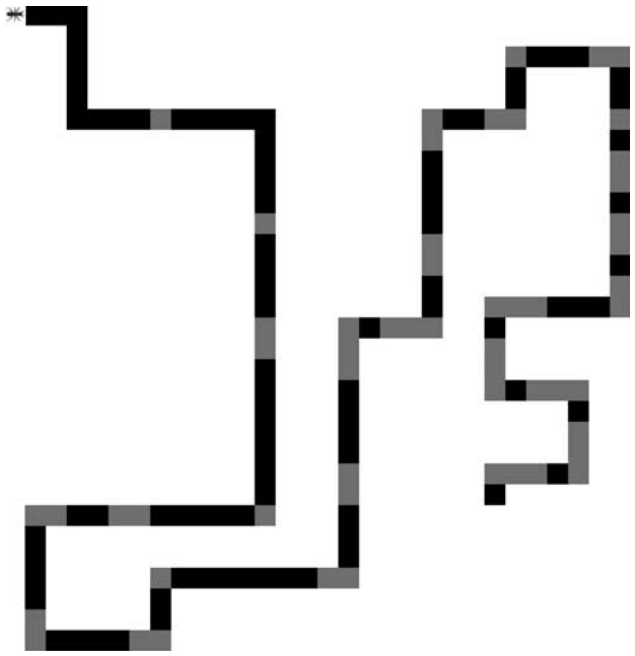

Figure 1 The Santa Fe ant trail grid. The ant begins in the upper left corner. The food is black, and the 'suggested' ideal route is included in grey. The grid is toroidal.

$$\langle expr\rangle ::= (\ \langle expr\rangle\ \langle op\rangle\ \langle expr\rangle\ )\ |\ \langle var\rangle$$

$$\langle op\rangle ::= +\ |\ -\ |\ *$$

$$\langle var\rangle ::= \mathrm{x}\ |\ 1.0$$

Figure 2 The grammar for symbolic regression.

$\langle expr \rangle$ ::= $\langle op \rangle$ ($\langle expr \rangle$, $\langle expr \rangle$) | $\langle var \rangle$ | $\neg$ $\langle var \rangle$

$\langle op \rangle$ ::= or | and | xor

$\langle var \rangle$ ::= x1 | x2 | x3 | x4 | x5

Figure 3 The grammar for even-5-parity.

its inputs are true. The grammar employed in this study to create the candidate functions is shown in Figure 3.

4. Grammatical evolution

Grammatical evolution (O'Neill and Ryan, 2001, 2003; Dempsey *et al*, 2009) is explained here because our local search code generator is based on the same grammar-based process of generating code. However, our local search methodology uses no evolution. It is based on searching the neighbourhood of one individual, not on combining elements from two 'parent' individuals. Grammatical evolution is one of a number of grammar-based forms of genetic programming, of which a survey can be found in McKay *et al* (2010).

Grammatical evolution is an evolutionary computation methodology that generates sentences in an arbitrary language defined by a Backus-Naur form (BNF) grammar. It evolves bit strings, as opposed to the classical genetic programming approach of evolving tree structures. The term 'program' is used in this section to mean a functioning fragment of code which can perform some task.

An example BNF grammar is shown in Figure 4. It consists of a set of symbols, referred to as 'non-terminals', which are shown to the left of the '::=' symbol on each line of the grammar. To construct a sentence from the grammar, the process begins with one non-terminal symbol, and a set of production rules that define with which sentences this non-terminal can be replaced. A sentence can consist of the same non-terminal, other non-terminals, and 'terminals'.

Terminals are components of the final sentence, which have no production rules because they are not subsequently replaced. Usually, there is a set of more than one production rule for each non-terminal, from which one must be chosen. When the non-terminal is replaced with a sentence, the non-terminals in the new sentence are each replaced with one sentence. Each non-terminal has its own set of production rules. When the sentence consists only of terminals, then the process is complete.

Grammatical evolution differs from standard genetic programming in that there is a clear separation between genotype and phenotype. The genotype can be represented as a variable length bit string, consisting of a number of 'codons', which are 8-bit sections of the genome. So, the decimal value of each codon can be

$\langle code \rangle$ ::= $\langle line \rangle$ | $\langle code \rangle$ $\langle line \rangle$

$\langle line \rangle$ ::= $\langle if \rangle$ | $\langle op \rangle$

$\langle if \rangle$ ::= if food_ahead ($\langle line \rangle$) else ($\langle line \rangle$)

$\langle op \rangle$ ::= move(); | left(); | right();

Figure 4 The grammar for the Santa Fe ant trail.

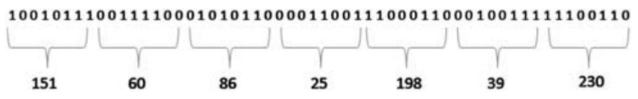

Figure 5 An example string, and its corresponding string of integers used to generate a program from the grammar.

between 0 and 255. This is the traditional grammatical evolution genotype representation, but in the grammatical evolution literature the genotype is now often represented directly as a decimal integer string. It has also been represented as a string of real valued codons in the grammatical swarm system (O'Neill and Brabazon, 2006).

The strings represent programs, because they encode a sequence of choices of production rules from the grammar. This methodology is especially applicable for generating code because the grammar can specify the programming language syntax, so that every generated program represents a valid section of code which can be compiled.

Figure 5 shows an example bit string genotype. We can see that it can be separated into 8-bit sections, which are referred to as codons. These codons each represent an integer value in the range [0, 255], and the specific integers generated from our example bit string are also shown in Figure 5. It is these integers which determine how a program is generated from the grammar.

Figure 4 shows the grammar we will use for this example, and Table 1 shows the process of generating a sentence from that grammar using the string of integers in Figure 5.

In Table 1, we can see that the generation process starts with one symbol, in this case '⟨code⟩'. From the grammar, we see that there are two production rules for this symbol: '⟨line⟩' and '⟨code⟩ ⟨line⟩'.

To make the choice between the two sentences, the first integer from the string is taken to modulus 2 (for two choices). This returns a value of 1, so the second option is chosen, as they are indexed from zero. This sentence replaces the original '⟨code⟩' symbol. This process can be seen in step 1 in Table 1, and the result can be seen in the current sentence of step 2.

We always replace the first available symbol in the current sentence, so the next integer is applied to the new

Table 1 The process of generating a program from the grammar, using the integers derived from the example bit string of figure 5 (151, 60, 86, 25, 198, 39, 230)

Step	*Current sentence*	*Choice made*
1	⟨code⟩	151 mod 2 = 1
2	⟨code⟩ ⟨line⟩	60 mod 2 = 0
3	⟨line⟩ ⟨line⟩	86 mod 2 = 0
4	⟨if⟩ ⟨line⟩	
5	if (foodahead()) { ⟨line⟩ } else { ⟨line⟩ } ⟨line⟩	25 mod 2 = 1
6	if (foodahead()) { ⟨op⟩ } else { ⟨line⟩ } ⟨line⟩	198 mod 3 = 0
7	if (foodahead()) { move(); } else { ⟨line⟩ } ⟨line⟩	39 mod 2 = 1
8	if (foodahead()) { move(); } else { ⟨op⟩ } ⟨line⟩	230 mod 3 = 2
9	if (foodahead()) { move(); } else { right(); } ⟨line⟩	151 mod 2 = 1
10	if (foodahead()) { move(); } else { right(); } ⟨op⟩	60 mod 3 = 0
11	if (foodahead()) { move(); } else { right(); } move();	

'⟨code⟩' symbol, which gets replaced by '⟨line⟩' in step 2. The process continues in the same way until there are no non-terminal symbols to replace in the sentence.

There are two exceptions to this rule. The first is that in step 4, the '⟨if⟩' symbol has only one choice of production rule in the grammar, so that rule is immediately applied without referring to the next integer. The second is that we reach the last integer in the sequence in step 8, at which point the so-called 'wrapping' mechanism occurs and we go back to the first integer again for the next choice, in step 9.

Wrapping is convenient, as it ensures valid individuals, but we argue that it does not contribute to a systematic search of the space of programs. Each part of the string in a good individual represents a set of choices, producing part of a successful program. We would suggest that including that part somewhere else in the same string (or in another string, as with the so-called 'crossover' operator) will change the original meaning of each integer value. Each integer will make a different choice when it is put in the new location, and so it will not create the same subtree that made it part of a fit individual in the first place. The results of this paper argue that there are more systematic search methodologies that have yet to emerge for automated programming.

The mechanism of generating a program from a grammar with a sequence of integers is a key concept used in our methodology, and so we have explained it as fully as possible. The fact that each 'codon' (integer) in the string defines a choice, from a pre-defined set of production rules, is the inspiration for the system presented in this paper, which searches the sequences

1	0	0	1	0	1	2	1	0
⟨code⟩	⟨code⟩	⟨line⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩

Figure 6 An overview of Section 5, the local search process.

of choices in a more systematic way than is possible through grammatical evolution.

5. Local search of programs

This section presents the local search methodology for automated code generation from a grammar. In the examples, the grammar for the Santa Fe ant trail is used. The local search methodology aims to find a string of integers which produces a fragment of code with the desired functionality, from a predefined grammar. The methodology is an iterative process by which a string's neighbourhood is systematically explored for a certain number of evaluations, and if no improvement is found then a new individual is generated at a different location in the search space. As such, it is similar to the technique of iterated local search (Baxter, 1981; Martin *et al*, 1992; Lourenco *et al*, 2003). Our methodology also includes a tabu list, which prevents fitness evaluations from being wasted on evaluating programs that have previously been evaluated.

This section is organised as follows. Section 5.1 explains the process of generating a new integer string. Section 5.2 explains the process by which the neighbourhood search of that string is started, and Section 5.3 describes how it continues. Section 5.4 then explains when the search of the neighbourhood is terminated. At this point a random new individual is created (returning the process to the beginning, at Section 5.1), or the process starts again from a better string which is found in the neighbourhood (returning the process to Section 5.2). This cycle is depicted in Figure 6. The pseudocode of the algorithm is shown in algorithm 1.

5.1. Initialising one string

This section explains the beginning of the search process, including the method by which we generate a random string to be the first 'current' string. The local search methodology also returns to this point if no superior string is found in the neighbourhood of the current string, so this part of the process is often performed many times before the end of a run.

The integer string represents a sequence of decisions of which production rule to apply next, as described in Section 4. Each integer is randomly chosen from the set of *valid choices* available at the current production rule.

The first integer in the string therefore represents the choice of production rule for the start symbol ⟨code⟩, and can only have a value of 0 or 1, as there are only two production rules for ⟨code⟩.

Algorithm 1 The pseudocode for the local search algorithm

```
$evals = 0$
$newstring \leftarrow TRUE$
while $evals \leqslant 23\,000$ do
  if $newstring = TRUE$ then
    $current\_string \leftarrow$ generate random string
  end if
  $newstring \leftarrow TRUE$
  $l \leftarrow length(current\_string)$
  $tabulists =$ new array (length $l$) of tabu lists
  {begin neighbourhood search}
  for $j = 1$ to 3 do
    for $i = 0$ to $l$ do
      string $s \leftarrow$ modify($current\_string$, index $i$)
      if $s$ is not in $tabulists[i]$ then
        $evals \leftarrow evals + 1$
        add $s$ to $tabulists[i]$
        if $s$ superior to $best$ then
          $best \leftarrow s$
        end if
      else
        $i = i - 1$
      end if
    end for
    if $best$ superior to $current\_string$ then
      $current\_string \leftarrow best$
      $newstring \leftarrow FALSE$
      break from loop
      {we return to the beginning of the while loop}
    end if
  end for
end while
return best found string
```

Figure 7 shows the string from which we will begin the example, and is typical of a string that would be created randomly in the initialisation process. In contrast to a genome in grammatical evolution, each integer corresponds to exactly one choice, and the range of values that it can take is limited by the number of options that are available to choose from. Underneath each integer is the particular non-terminal symbol from which the integer chooses.

It may further clarify the process to visualise the string representation as a tree representation, starting with the root (in this case the ⟨code⟩ symbol). Figure 8 shows the tree structure which is generated by the individual in Figure 6. At each non-terminal symbol in the tree, the integer that makes the choice of production rule is shown in brackets after the symbol. These integers correspond to the integer sequence in Figure 6, applied in a 'depth first' order.

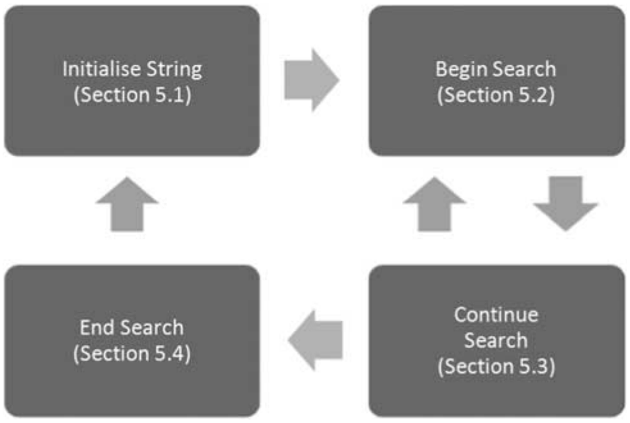
Initialise String (Section 5.1)
Begin Search (Section 5.2)
Continue Search (Section 5.3)
End Search (Section 5.4)

Figure 7 An example integer string, as used in the local search methodology presented in this paper. Each integer is linked to the symbol that it chooses the production rule for.

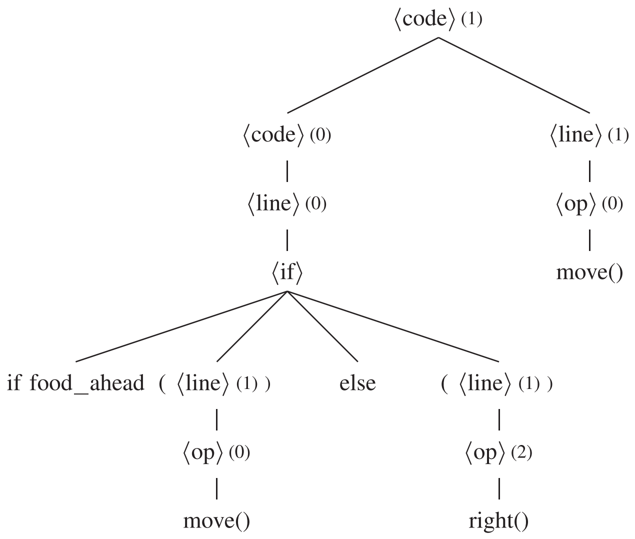

Figure 8 The tree structure represented by the integer string in Figure 6.

Algorithm 2 shows the valid program that is generated by the integer string from the grammar. One can see that the final program consists of all of the non-terminal symbols shown in the tree of Figure 8. This program is evaluated on the Santa Fe ant trail problem, to ascertain the fitness of the string that created it. We find that it has a fitness of 4, as four food squares are visited by the ant. This is a poorly performing individual, as the maximum fitness is 89. Nevertheless, the local search methodology will search its neighbourhood for a better performing individual. Section 5.2 explains how this search begins.

Algorithm 2 The program created by the integer string in Figure 6. One can see the process of the string creating the program by consulting Figure 8

```
if food_ahead then
    move()
else
    right()
end if
move()
```

5.2. Beginning the neighbourhood search

This section describes the process of beginning a search of the neighbourhood of one string. The string will either have just been created from the process in Section 5.1, or it will be a superior string that has just been found in the neighbourhood of the current string. The search involves sampling the neighbourhood, as we cannot search the full neighbourhood in a reasonable amount of time.

A neighbourhood move is defined as the modification of one of the integers in the string. In our example of Figure 6, changing one integer corresponds to a change in one branch of the parse tree in Figure 8, and so it also represents a change of one block of code in the program shown in algorithm 2.

Every integer in the string is changed once, each time creating a new string (each of which differs from the original by exactly one integer). Therefore, in total, we will create as many new strings as there are integers in the original string. In our example of Figure 5, there are nine integers, and so nine new strings will be created.

It is important to note that each integer in the string defines a substring, which represents a subtree in the tree representation of the program. For example in Figure 6, the sixth integer, '1', represents the substring '1, 2'. The reason for this can be seen in the tree structure of Figure 8, where that substring produces the branch '⟨line⟩(1) – ⟨op⟩(2) – right()'. The seventh integer, '2', chooses the third production rule to apply for the ⟨op⟩ symbol, and the ⟨op⟩ symbol was only produced because of the '1' integer at the ⟨line⟩ symbol before it. Therefore, the correct semantic meaning of the integer '2' is maintained only if it appears after the '1'.

To maintain the correct semantic meaning of all of the integers in the individual, when an integer is modified to create a new string, that integer's substring is deleted, and a new substring of integers is generated. This new substring corresponds to the new set of choices which must be made after the modified integer. As explained in the previous paragraph, the old substring is irrelevant, because those choices do not exist anymore.

Figure 9 shows this process. In the upper string, we change the integer '1', marked with two '$*$' symbols. It must change to a zero, because the ⟨line⟩ symbol only has two production rules, as can be seen in the grammar of Figure 4. The '0' integer to its immediate right is not relevant any more in the string, as it only had meaning when the choice of '1' was made before it. In other words, the ⟨op⟩ symbol does not exist because the choice of the second production rule for the ⟨line⟩ symbol was not chosen. Instead, the first production rule is chosen, changing the ⟨line⟩ symbol to an ⟨if⟩ symbol.

The lower string of Figure 9 shows the new individual. The integers which now follow the new choice of '0' are randomly generated, but they are only chosen from the choice of production rules at the point at which they are required. They each represent the choice of one specific production rule for the symbol shown underneath them. This process of generating a new substring is functionally equivalent to generating a new valid subtree.

Figure 10 shows the parse tree generated by the new integer string. When it is compared with the original parse tree of Figure 8, one can see that the change of one integer has replaced one subtree with another, and therefore changed the code at one location. This is similar to the 'mutation' operator in genetic programming (Koza, 1992) (in which a subtree is replaced). However, in genetic programming, the choice of where to insert the new subtree is made at random, and it is only made once. It is executed only occasionally, as a method to diversify the search.

In this paper, the 'mutation' (or subtree replacement) process is more systematic, because all of the possible modification positions are attempted. The subtree replacement is used as an integral part of the search process, rather than as a method to occasionally diversify the search by mutating a random part of the tree.

To return to our example, the change of the integer '1' to '0' resulted in a second conditional statement being added to the code. This can be seen clearly when comparing Figure 8 with Figure 10. However, modifying any integer

1	0	0	1	0	1	2	***1***	0
⟨code⟩	⟨code⟩	⟨line⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩

1	0	0	1	0	1	2	***0***	1	1	1	0
⟨code⟩	⟨code⟩	⟨line⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩

Figure 9 An example of how the change in one integer may mean that a substring is rendered obsolete, and a new substring must be generated. The string on the top is the original string. We change the integer marked with two '$*$' symbols. The '0' integer to its immediate right is not relevant any more in the string, as it only had meaning when the choice of '1' was made before.

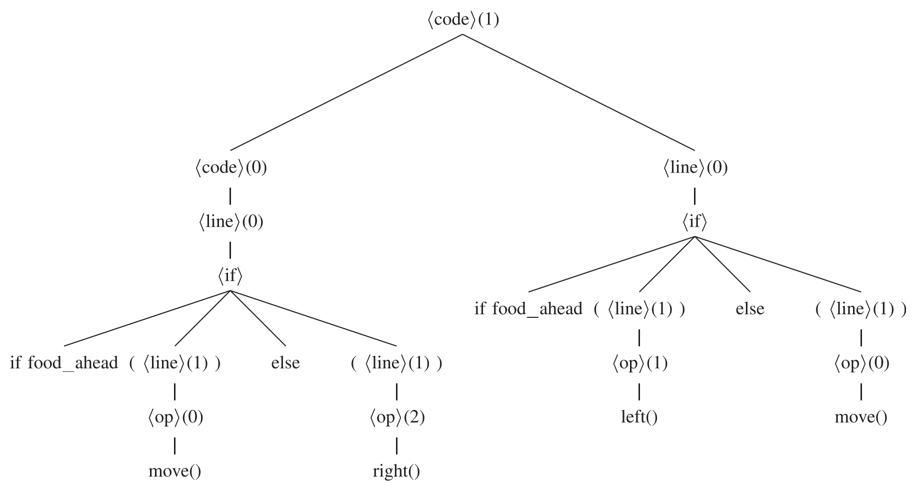

Figure 10 The tree structure represented by the lower integer string in Figure 9.

Original Integer String	1	0	0	1	0	1	2	1	0
Corresponding symbol	⟨code⟩	⟨code⟩	⟨line⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩
Location Index	0	1	2	3	4	5	6	7	8
Fitness of new Individual	**0**	**4**	**0**	**1**	**0**	**6**	**3**	**1**	**1**

Figure 11 An example of the fitnesses of strings which could be produced from the original string from Figure 6.

which acts on an ⟨op⟩ symbol will only result in exchanging 'move()', 'left()', and 'right()'. This is because they are terminal sentences, which result in no further choices, so no new subtrees are generated.

When each new individual has been created, it is added to a tabu list unique to that location on the original string. Every integer's location on the original string has its own tabu list, so that identical points in the search space are not revisited. So, as we continue to explore the neighbourhood of the original string, if a second modification of the '1' integer produces, by chance, exactly the same individual as shown in the lower string of Figure 9, then we do not accept it. In such a case the system will attempt to generate another individual by again modifying the '1'.

Figure 11 shows the fitnesses of the strings that were generated by changing each integer in the original string. In our example, the programs control an artificial ant, and the fitness is the number of food squares visited by the ant when under the control of the program. In this example, the individual created from changing the '1' integer, at index 5, obtains a fitness of 6. Recall that the fitness of the original individual was 4. Therefore, this new string is better than the original, and so at this point the neighbourhood search has found a superior string. The neighbourhood search process begins again (from the start of Section 5.2), but this time using the superior individual as the starting point.

So that we can fully explain the system, assume that a string with fitness 0 was found by changing the integer at index 5, instead of a string with fitness of 6. In such a case, a superior string has not been found in this beginning phase of the neighbourhood search (which modifies each integer in the string once). The search process then continues in the next section. A 'superior' string is one that produces superior code from the grammar, in this example this means code that controls the artificial ant to visit more food squares.

In addition, an individual that performs equally as well as the original individual is classed as better if it contains less integers. This exerts a form of parsimony pressure on the candidate strings, and is designed so that less fitness evaluations are required. Longer strings require more fitness evaluations to search their neighbourhood, because the allocated number of evaluations is set to three times the length of the string.

After the beginning phase of the neighbourhood search, each integer will have been changed once, each time creating a new string. Each location on the original string will have an independent tabu list with one entry, and each new individual will have been evaluated.

5.3. Continuing the neighbourhood search

The process described in Section 5.2 begins the neighbourhood search by modifying each integer in the string once.

The search then continues until either a superior string is found in the neighbourhood, or the termination criterion is reached.

The neighbourhood search continues by repeating the process of generating one new string for each of the locations on the original string, exactly as described in the previous section. However, it should be noted that each newly created string is added to the tabu list for that location if it doesn't exist already. If it does exist in the tabu list, then that individual has been explored before in the neighbourhood, and the system generates another string which has not been evaluated before.

Of course, it could be that all of the possible changes at a location have been explored. For example, at the last location on our example string (see Figure 11), there are only three possible values that this integer can take, and so there are two possible changes from what it started as. The choice at the '$\langle$op$\rangle$ ::= *move*()|*left*()|*right*()' rule can only be $0, 1$, or 2. There are also no new substrings that are generated after them, as they all represent terminal symbols in the grammar. If all of the two possible modifications have been exhausted at this location, then a random other location index is chosen, and a new string is created from changing the integer there instead.

The programs represented by the new candidate strings are all evaluated to determine their fitness. If none of the new strings are superior to the original string, then the neighbourhood search repeats again from the start of this section, unless the termination criteria for the neighbourhood search has been reached. By default, we execute this process three times in total. Which means that for an original string with a length of 9, there will be 27 evaluations of strings in its neighbourhood, because three new strings will be generated from modifying each location on the original string. Section 7.1 investigates the impact of evaluating different numbers of strings in the neighbourhood, thereby increasing or decreasing the 'depth of search' of the neighbourhood.

5.4. Ending the neighbourhood search

The neighbourhood search ends when we have evaluated a certain number of neighbours of the original string without finding a string which produces a superior program from the grammar. We set this value to be three times the length of the original individual.

The value is related to the length of the original string because we predict that larger strings require more evaluations to adequately explore their neighbourhood. A small string consisting of three integers does not require 100 samples to adequately explore its neighbourhood, but this may be appropriate for a larger string with 30 or more integers. Strings with more integers correspond to more complex programs, and so it makes sense to allocate more resources to the neighbourhood search of such programs.

Table 2 Summary of the grammatical evolution parameters

Parameter	*Value*
Population Size	250
Generations	100
Generational Model	Steady State
Generation Gap	0.9
One Point Crossover Probability	0.9
Point Mutation Probability	0.01
Duplication Probability	0.01
Pruning Probability	0.01
Codons in Initial Population	10–200
Maximum Genome Wraps	10

The value of 3 can be thought of as the depth to which we search the neighbourhood around each individual.

When the limit on the number of samples of the neighbourhood of the original string has been reached, without finding a neighbour with superior fitness, we end the neighbourhood search around that particular string, and begin again from a new randomly generated individual (from the start of Section 5.1).

This process of iterated local search can continue for an arbitrary total number of iterations, but for the purposes of this study, we terminate the local search methodology after 23 000 fitness evaluations, as this is identical to a run of grammatical evolution with a steady state population, 0.9 replacement proportion, 50 generations, and a population size of 500 $((450 \times 50) + 500)$. These are settings commonly used in the grammatical evolution literature. However, in our experiments (shown in Section 7.2) we have found different parameters to be superior.

6. Results

In the results section we compare to a standard run of grammatical evolution, implemented by the jGE java software package (http://www.bangor.ac.uk/~eep201/jge/), using the parameters shown in Table 2. We will show that our local search methodology finds successful code more often than grammatical evolution, and with less fitness evaluations.

The results in Figure 12(a)–(c) each show the results of 200 runs of our local search methodology, compared with 200 runs of grammatical evolution. Both are allowed to run for a maximum of 23 000 fitness evaluations, to ensure a fair comparison. One fitness evaluation is a test of the code (produced by the integer string) on the given problem, to ascertain its fitness.

The figures show the cumulative frequency of success at each number of fitness evaluations, where a 'success' is finding a string which produces the ideal program from the grammar. For example, in the Santa Fe ant trail problem, the ideal string would be one which produces a program

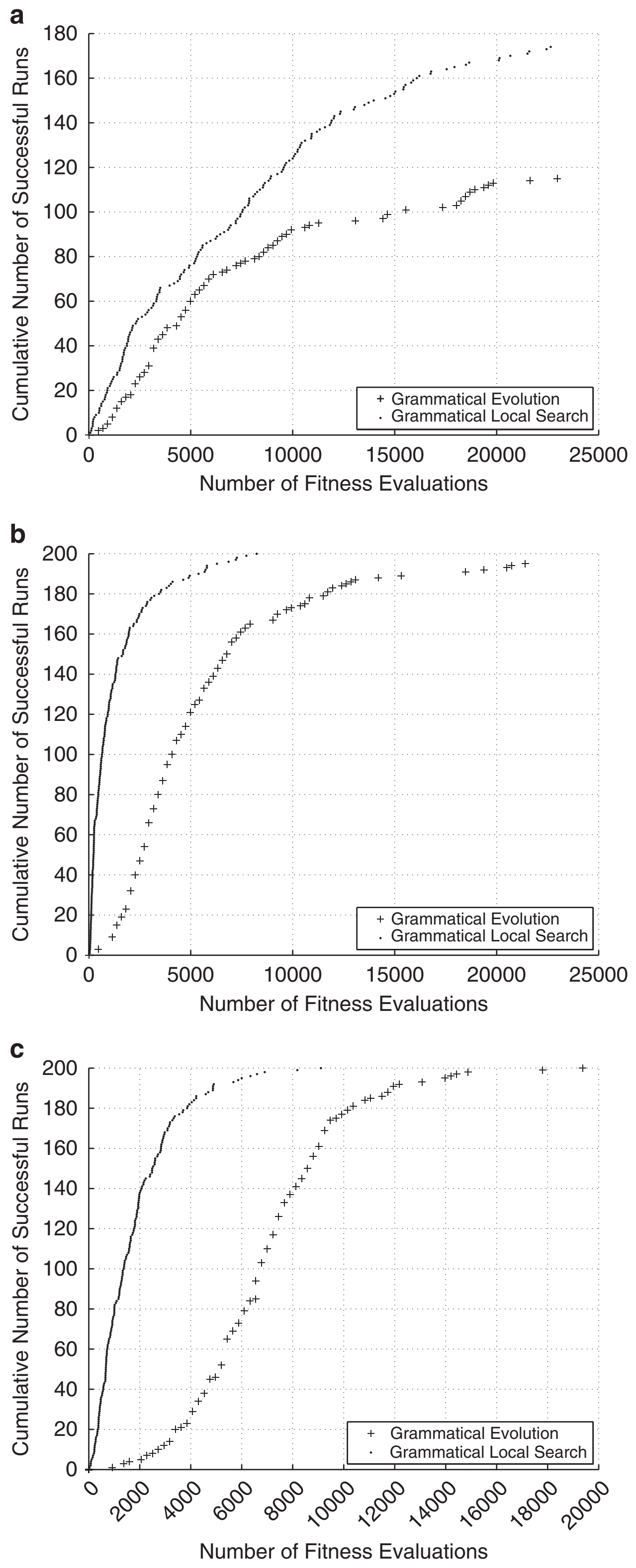

Figure 12 Our local search methodology compared with grammatical evolution. (a) Santa Fe ant trail; (b) Symbolic regression; (c) Even-5-parity.

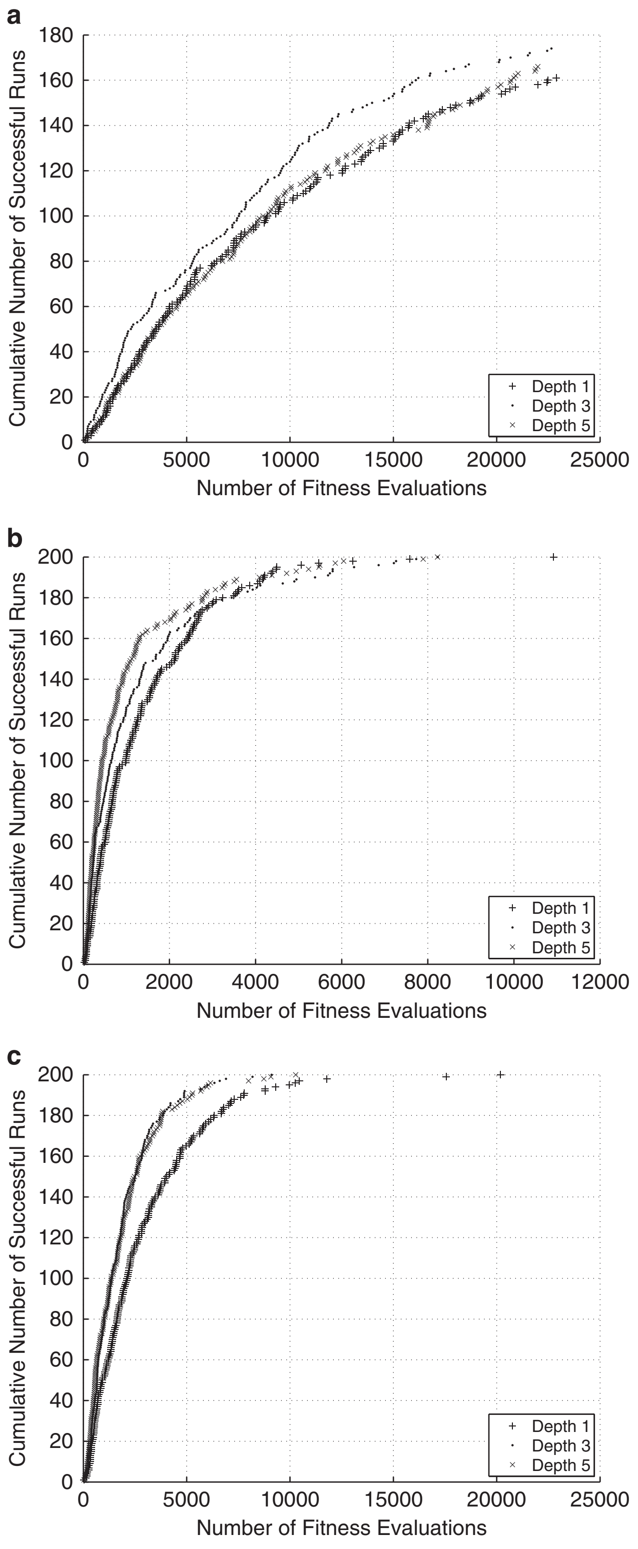

Figure 13 The results of different depth of search values. A depth of search of 3 means three new strings are generated from mutating each integer in the string. (a) Santa Fe ant trail; (b) Symbolic regression; (c) Even-5-parity.

that controls the ant to move over all of the food squares. For the grammatical evolution results, we test for an ideal solution after every 'generation', which means the evaluation of all of the individuals in the population. In every generation, 225 new individuals are produced (0.9 generation gap), and so there are often many successes reported at each interval of 225 fitness evaluations. Therefore, there are often large 'jumps' in the number of successes reported for the grammatical evolution results in Figure 12(a)–(c).

Figure 12(a) shows the comparison of results on the Santa Fe ant trail problem described in Section 3.1. One can observe that the performance of our local search methodology is superior to grammatical evolution, because it finds the required code more often, and using less fitness evaluations. Local search finds the required code in 174 of the 200 runs, while grammatical evolution finds the required code in 115 of the 200 cases.

The difference between the two methodologies for code generation is more clear on the symbolic regression problem and the even-5-parity problem, in Figure 12(b) and (c). For symbolic regression, all 200 of the runs of local search generate the correct code, while grammatical evolution succeeds in 195 runs. For even-5-parity, all runs of grammatical evolution and local search produce the correct code, but local search does so in significantly less fitness evaluations. For example, for even parity, half of the grammatical evolution runs completed with a success by 6775 fitness evaluations, while for local search this is the case after just 1366 fitness evaluations. For symbolic regression, half of the grammatical evolution runs completed with a success by 4075 fitness evaluations, while for local search this is the case after 648 fitness evaluations.

The results are statistically confirmed by Mann-Whitney Rank Sum tests. For each problem domain, we compare the two distributions of 200 results. Each of the 200 results in a distribution is the number of fitness evaluations required during that run to obtain the correct code. Some runs did not find the correct code in 23 000 fitness evaluations, and those results are set to 23 000. The null hypothesis is that there is no difference between the two distributions. The alternative hypothesis is that one distribution tends to have larger values than the other. For each of the three problem domains, the probability that the null hypothesis is true is less than 0.0001, so we can reject the null hypothesis with a confidence level of at least 99%.

7. Further analysis of results

7.1. Impact of the depth of search

The depth of search refers to the number of neighbours that are sampled from a candidate integer string before the neighbourhood search is terminated, at which point a new randomly generated string is created. Owing to the experiments presented in this section, we set this parameter

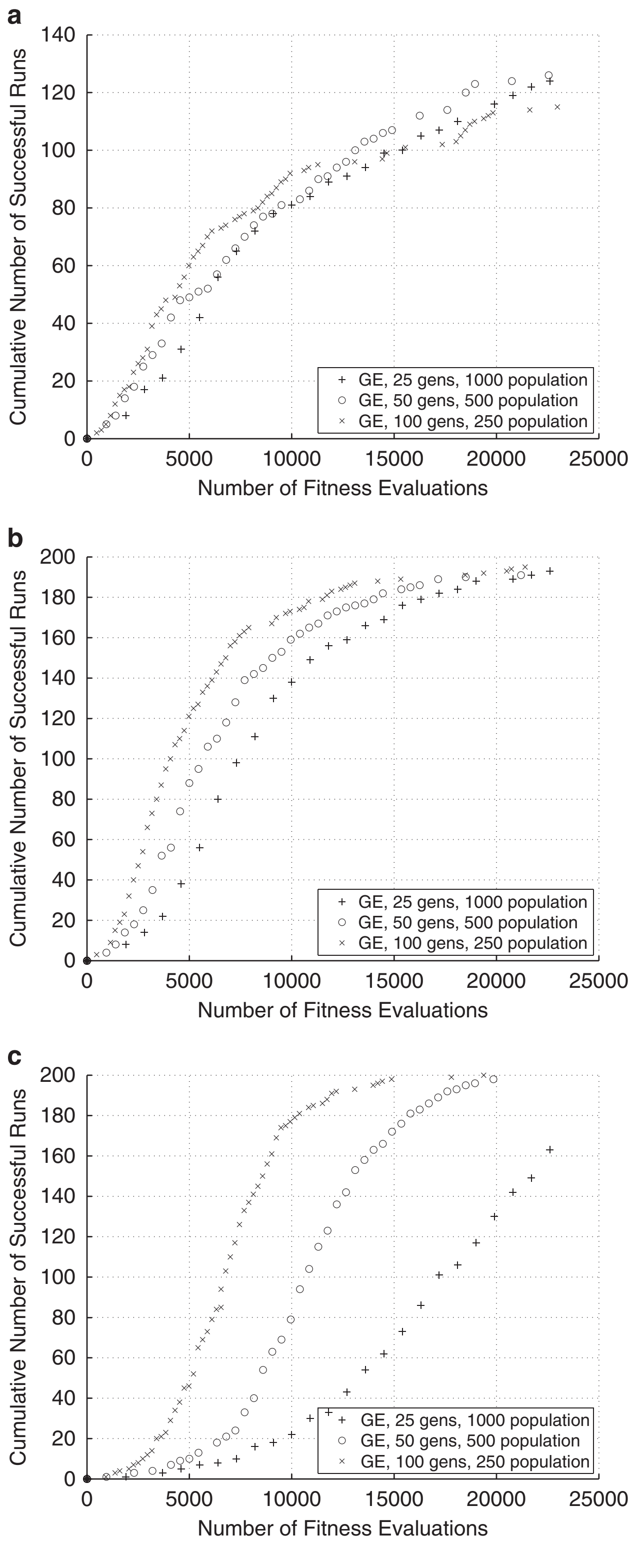

Figure 14 The results of different parameters for grammatical evolution. (a) Santa Fe ant trail; (b) Symbolic regression; (c) Even-5-parity.

to be three times the length of the candidate string, which seems to represent a good balance between the need for

Location Index	0	1	2	3	4	5	6	7	8
Original Integer String	1	0	0	1	0	1	2	1	0
Corresponding symbol	⟨code⟩	⟨code⟩	⟨line⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩	⟨line⟩	⟨op⟩
Fitness of new individual	0	4	0	1	0	1	3	1	1
Number of times selected	**0**	**3**	**0**	**0**	**0**	**1**	**3**	**0**	**2**

Figure 15 An example of how many times each integer location on the string is selected to be modified in the next iteration, based on the fitness change that each location has caused in the past. One can see that the locations with a higher fitness are chosen more frequently, through the stochastic process of tournament selection.

exploration of new strings and the search around an existing string.

From Figure 13, we can see that employing a depth of 1 is always inferior to searching the neighbourhood more deeply. For the even-5-parity problem, the performance of a 3 or 5 depth is very similar. For the Santa Fe ant trail, a depth of 3 is clearly superior, but for the symbolic regression problem a depth of 5 is superior.

7.2. Comparison of different grammatical evolution parameters

The results of different configurations of grammatical evolution can be seen in Figure 14. They show that our default parameters used in the main body of the paper cannot be considered a purposely selected 'inferior' set of parameters. In our experiments, we compare with grammatical evolution with a 250 population size, and 100 generations, and it can be seen in Figure 14 that this obtains the best results. In fact, the 50 generations and 500 population size configuration can be seen in many grammatical evolution papers. Our results here show that this is not necessarily the best set of parameters to employ.

7.3. Including a bias in the neighbourhood search

Recall, from Section 5.3, that our local search methodology searches the neighbourhood by sampling three new individuals from each location on the original string. Three new individuals are created by changing the first integer, three from the second, and so on.

Instead of this uniform approach, it is possible to include a bias into the methodology, which chooses integer locations to modify based on the past performance of the new strings which were generated from modifying that location.

The goal of such a bias would be to avoid wasting fitness evaluations. There will be some integers in the string which are integral to the good functioning of the resultant program. Modifying those, and evaluating the resulting string, could be a waste, as the code is likely to be poor.

Regarding just the 'fitness of new individual' row in Figure 15, one can see that you get a fitness of zero if you change the integer at index 4. This is because, in the resulting program, this is equivalent to changing the 'move' command that occurs after the 'if food_ahead' conditional statement. It is beneficial to move forward to collect the food, and so changing this to a 'right' or 'left' command results in the ant not responding correctly to seeing a food square ahead. So, recording the fitness obtained by changing each part of the program could potentially help to avoid changes to the program that will probably only reduce the fitness of the individual.

The results of this section show that one particular mechanism that attempts to avoid unproductive areas of the neighbourhood is, in fact, not beneficial. When this bias is applied to the neighbourhood search, the results of the local search methodology are inferior to when there is no bias in the search. It seems that this particular bias is not successful at directing the search towards better areas of the neighbourhood.

7.3.1. Explanation of the search bias. To bias the neighbourhood search, we first modify each integer in the string once (as before), and record the fitness of each new string. However, to provide information to help bias the search, these values will represent a running mean average value for each location on the original string. The individual in our example (Figure 6) had a length of 9, so nine average values are maintained. At the start of the search, the average values are all calculated from the fitness of exactly one different string each. Later, they will be calculated from more than one string, because we will modify each integer more times.

We assume that the points of the string at which a change produced a relatively good neighbour are more likely to produce further good neighbours. So, we select those points more often as candidates for modification. We use 'tournament' selection to generate a new set of integer indices for modification.

Tournament selection balances exploration and exploitation in the search process, and is commonly used in the evolutionary computation literature to select individuals from a population. Here, we use it to select integer positions from one string. To select one integer location, the tournament selection mechanism considers a randomly selected group of half of the integer locations on the string. Of those, the location is chosen whose modification has previously produced new strings with the best average fitness. Tournament selection is applied repeatedly, each time selecting one location. As before, we select the same number of locations for modification as the length of the string. However, now, some locations may be selected more than once, and some will not be chosen if they previously produced inferior strings.

Figure 15 shows an example of how many times each integer location on the string is selected to be modified in the next iteration. One can see that the locations with a higher fitness are chosen more frequently, through the stochastic process of tournament selection. Now that the locations for the changes have been selected, the new strings are generated from each chosen location. For example, this means that the integer at location index 1 will be changed three times, therefore creating three new strings in the neighbourhood of the original string.

The programs represented by the new candidate strings are all evaluated to determine their fitness, and the average fitness of the locations is updated accordingly. So the average fitness value of each location could now still be based on just one result (the original change performed on each integer), or based on multiple results if the location has been selected more times by the tournament selection method.

At this point, the biased neighbourhood search process repeats again. We have a new average fitness value for each selected location in the original string, and this updated knowledge is utilised in the next iteration. As more strings are sampled from a location on the original string, we get a more accurate view of the quality of strings produced when we change the integer at that location in the string. This makes it more likely or less likely that the location will be chosen to be modified in the next iteration. As before, we terminate the neighbourhood search after a certain number of neighbours have been sampled, set by default to be three times the length of the original string.

7.3.2. Results of including a bias to the search. In Figure 16, one can see that the results obtained with the bias disabled are superior to the results with the search bias enabled. Without a search bias, the required code is found using less fitness evaluations. This can be seen most clearly in the Santa Fe ant trail results of Figure 16(a). The results are less obvious in Figure 16(c), for the even-5-parity problem. There is no notable difference in the results on symbolic regression.

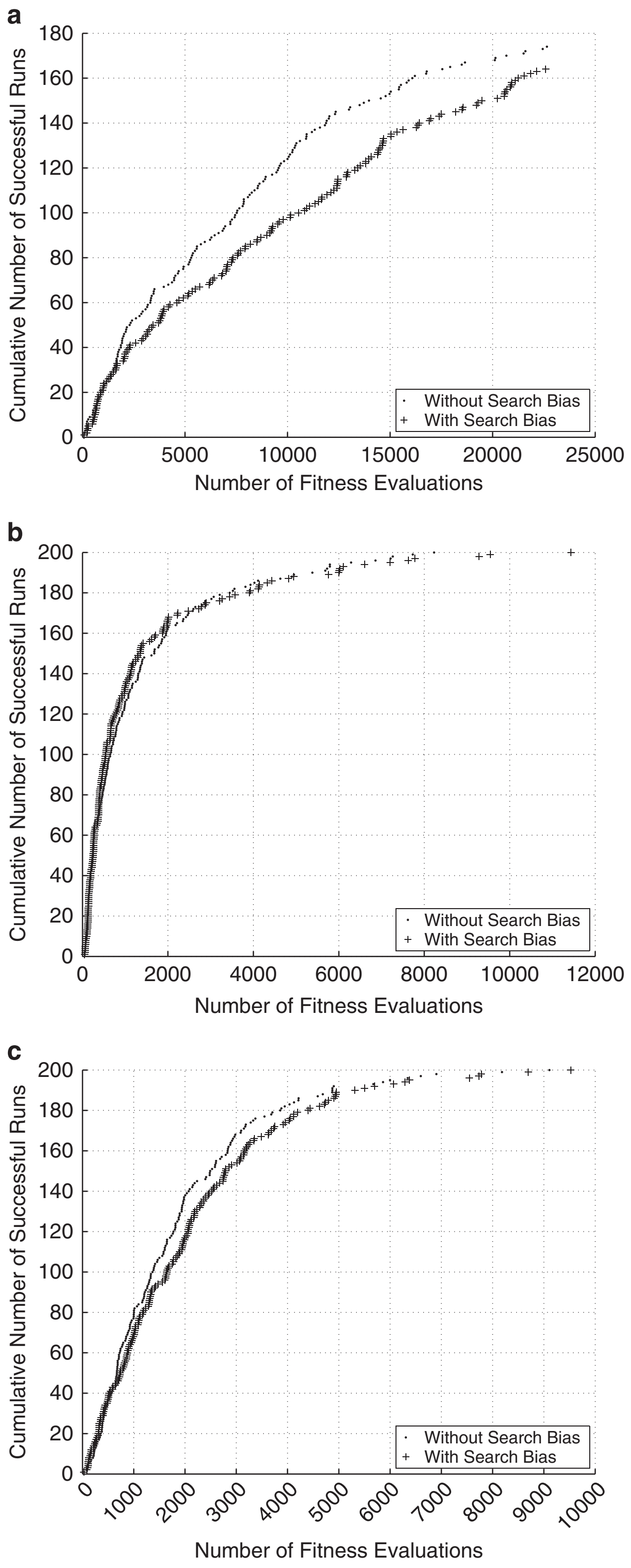

Figure 16 The results obtained by our local search methodology with, and without, the neighbourhood search bias. (a) Santa Fe ant trail; (b) Symbolic regression; (c) Even-5-parity.

a

```
if food_ahead then
  right()
else
  if food_ahead then
    right()
  else
    if food_ahead then
      right()
    else
      right()
    end if
  end if
end if
if food_ahead then
  move()
else
  left()
end if
move()
left()
if food_ahead then
  left()
else
  if food_ahead then
    move()
  else
    right()
  end if
end if
```

b

```
if food_ahead then
  right()
else
  right()







end if
if food_ahead then
  move()
else
  left()
end if
move()
left()
if food_ahead then
  left()
else
  if food_ahead then
    move()
  else
    right()
  end if
end if
```

c

```
if food_ahead then
  right()
else
  right()







end if
if food_ahead then
  move()
else
  left()
end if
move()
left()
if food_ahead then
  left()
else
  right()




end if
```

d

```
if food_ahead then
  right()
else
  right()







end if
if food_ahead then
  move()
else
  left()
end if
move()
left()
if food_ahead then
  move()
else
  right()




end if
```

Figure 17 Program 1 is too large to display. (a) Program 2 of 5; (b) Program 3 of 5; a large section of code is removed; (c) Program 4 of 5; a second section of code is removed; (d) Program 5 of 5; a 'left' command is replaced by 'move'.

This suggests that the bias which we attempted to include into the local search methodology is not performing the intended task of intelligently selecting which parts of the neighbourhood to explore. We believe that there are methods which would be more successful than this attempt, as many fitness evaluations are likely to be spent on strings which are very unlikely to improve the fitness of the original string.

There are also likely to be many changes which will not affect the functionality of the resulting program, and so such modifications are also wasteful. Further research should be put into methodologies to improve the detection of which locations on the string are likely to be beneficial.

7.4. Example of neighbourhood search

In this section we examine one example sequence of moves through a neighbourhood, to further clarify the local search process. Figure 17 shows a sequence of neighbourhood moves at the end of one of the 200 runs for the Santa Fe ant problem. The last neighbourhood move produces a string which generates an ideal program from the grammar. This program controls the artificial ant to visit all 89 food squares.

In Figure 17, the sequence of neighbourhood moves can be seen, in terms of the code produced by the integer strings that are being manipulated. This particular sequence of moves began with a randomly generated string which produced a program from the grammar that was much too large to fit on the page. Figure 17a shows the program generated by a superior neighbour of that large string. The large gaps in Figures 17b–17d serve to highlight the parts of the program that have changed due to the modification of an integer in the string.

8. Conclusion and future work

This paper has presented a methodology for automatic code generation based on local search of programs. This is in contrast to the majority of such systems which are based on evolutionary computation. The contribution of this paper is to describe an effective neighbourhood of a program, based on its integer string representation. We also describe an effective local search methodology to search that neighbourhood, based on iterated local search.

As the problems which we solve become more complex, then it is a possibility that finding good quality solutions to those problems may require ever more complex systems. Automatic generation of those systems (or components of those systems) may become an integral tool, because to design those complex systems by hand may require a prohibitive amount of time and resources.

Currently, the majority of automatic programming systems utilise evolutionary computation. By comparison, local search methodologies to automatically build systems have not yet received any significant research attention, and we see no reason why local search systems cannot be as effective as, or more effective than, evolutionary computation.

This paper generates some ideas for future research. The classic definition of a memetic algorithm is one which combines the ideas of evolutionary computation with local search, and so this is one area where the ideas of this research could complement existing evolutionary automatic programming approaches. Instead of only relying on the evolutionary algorithm, a local search could be applied to search the neighbourhoods of the best individuals at certain generations.

We have employed a local search methodology based on iterated local search, but other common metaheuristic techniques may be better suited to searching the space of programs. Our methodology begins again from a new randomly generated program when a local optimum appears to have been found, but it could be the case that a great deluge or simulated annealing acceptance criteria could escape local optima just as effectively.

Programs generated by a stochastic automated process generally contain redundant sections which do not contribute to the fitness of the program. This is due to the fact that removing or adding them does not change the functionality of the program, and therefore its fitness. A potential area of future research is to investigate effective methods to avoid wasting time in modifying such sections. This paper presents a method which avoids changes which are likely to *worsen* the fitness, but has no direct method to avoid changes which do not change the fitness.

As a long-term research goal, it would be useful to investigate whether there are any characteristics or representations in genetic programming, which affect the relative performance of local search and evolutionary search. This could form the basis of a theoretical understanding of when one methodology is more likely to perform better than the other.

In conclusion, this paper complements the significant amount of research into systems such as genetic programming and grammatical evolution. The results show that a local search methodology, in conjunction with a grammar, can be more efficient than evolutionary computation in some problem domains.

References

Allen S, Burke EK, Hyde MR and Kendall G (2009). Evolving reusable 3D packing heuristics with genetic programming. In: Raidl G *et al* (eds). *Proceedings of the ACM Genetic and Evolutionary Computation Conference (GECCO' 09)*. ACM: Montreal, Canada, pp 931–938.

Bader-El-Den MB and Poli R (2007). Generating SAT localsearch heuristics using a gp hyper-heuristic framework. In: Monmarché N, Talbi E-G, Collet P, Schoenauer M and Lutton, E (eds). *LNCS 4926. Proceedings of the 8th International Conference on Artificial Evolution*. Springer: Tours, France, pp 37–49.

Banzhaf W, Nordin P, Keller RE and Francone FD (1998). *Genetic Programming, An Introduction: On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann: San Francisco.

Baxter J (1981). Local optima avoidance in depot location. *Journal of the Operational Research Society* **32**(9): 815–819.

Burke EK, Hart E, Kendall G, Newall J, Ross P and Schulenburg S (2003a). Hyper-heuristics: An emerging direction in modern search technology. In: Glover F and Kochenberger G (eds). *Handbook of Meta-Heuristics*. Kluwer: Boston, MA, pp 457–474.

Burke EK, Kendall G and Soubeiga E (2003b). A tabu-search hyperheuristic for timetabling and rostering. *Journal of Heuristics* **9**(6): 451–470.

Burke EK, Hyde MR and Kendall G (2006). Evolving bin packing heuristics with genetic programming. In: Runarsson T, Beyer HG, Burke E, Merelo-Guervos JJ, Whitley D and Yao X (eds) *LNCS 4193, Proceedings of the 9th International Conference on*

Parallel Problem Solving from Nature (PPSN'06). Springer: Reykjavik, Iceland, pp 860–869.

Burke EK, Hyde MR, Kendall G and Woodward J (2007a). Automatic heuristic generation with genetic programming: Evolving a jack-of-all-trades or a master of one. In: Thierens D *et al* (eds). *Proceedings of the 9th ACM Genetic and Evolutionary Computation Conference (GECCO'07)*. ACM: London, England, UK, pp 1559–1565.

Burke EK, Hyde MR, Kendall G and Woodward J (2007b). The scalability of evolved on line bin packing heuristics. In: Srinivasan D and Wang L (eds). *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'07)*. IEEE: Singapore, pp 2530–2537.

Burke EK, Hyde M, Kendall G, Ochoa G, Ozcan E and Woodward J (2009). Exploring hyper-heuristic methodologies with genetic programming. In: Mumford C and Jain L (eds). *Computational Intelligence: Collaboration, Fusion and Emergence*. Springer: New York, pp 177–201.

Burke EK, Hyde M and Kendall G (2010a). Providing a memory mechanism to enhance the evolutionary design of heuristics. In: Sobrevilla P, Aranda J and Xambo S (eds). *Proceedings of the IEEE World Congress on Computational Intelligence (WCCI'10)*. IEEE: Barcelona, Spain, pp 3883–3890.

Burke EK, Hyde M, Kendall G, Ochoa G, Ozcan E and Woodward J (2010b). A classification of hyper-heuristics approaches. In: Gendreau M and Potvin J-Y (eds). *Handbook of Meta-Heuristics 2nd Edition*. Springer: New York, pp 449–468.

Burke EK, Hyde MR, Kendall G and Woodward J (2010c). A genetic programming hyper-heuristic approach for evolving 2-D strip packing heuristics. *IEEE Transactions on Evolutionary Computation* **14**(6): 942–958.

Dempsey I, O'Neill M and Brabazon A (2009). *Foundations in Grammatical Evolution for Dynamic Environments*. Springer: New York.

Diosan L and Oltean M (2009). Evolutionary design of evolutionary algorithms. *Genetic Programming and Evolvable Machines* **10**(3): 263–306.

Fukunaga AS (2008). Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation (MIT Press)* **16**(1): 31–61.

Geiger CD, Uzsoy R and Aytug H (2006). Rapid modeling and discovery of priority dispatching rules: An autonomous learning approach. *Journal of Scheduling* **9**(1): 7–34.

Ho NB and Tay JC (2005). Evolving dispatching rules for solving the flexible job-shop problem. In: Corne D *et al* (eds). *Proceedings of the IEEE Congress on Evolutionary Computation (CEC'05)*. IEEE: Edinburgh, UK, pp 2848–2855.

Juels A and Wattenberg M (1996). Stochastic hillclimbing as a baseline mathod for evaluating genetic algorithms. In: Touretzky DS, Mozer MC and Hasselmo ME (eds). *Advances in Neural Information Processing Systems*. MIT Press: Denver, CO, Vol. 8, pp 430–436.

Kenyon C (1996). Best-fit bin-packing with random order. In: Tardos E (ed). *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics Philadelphia, pp 359–364.

Koza JR (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press: Boston, MA.

Koza JR and Poli R (2005). Genetic programming. In: Burke EK and Kendall G (eds) *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Kluwer: Boston, MA, pp 127–164.

Langdon WB and Poli R (1998). Why ants are hard. In Koza JR *et al* (eds). *Genetic Programming 1998: Proceedings of the Third Annual Conference*. Morgan Kaufmann, University of Wisconsin, Madison: Wisconsin, USA, pp 193–201.

Løkketangen A and Olsson R (2010). Generating metaheuristic optimization code using ADATE. *Journal of Heuristics* **16**(6): 911–930.

Lourenco HR, Martin O and Stutzle T (2003). Iterated local search. In: Glover F and Kochenberger G (eds). *Handbook of Meta-Heuristics*. Kluwer: Boston, MA, pp 321–353.

Martin O, Otto SW and Felten EW (1992). Large-step markov chains for the tsp incorporating local search heuristics. *Operations Research Letters* **11**(4): 219–224.

McKay RI, Hoai NX, Whigham PA, Shan Y and O'Neill M. (2010). Grammar-based genetic programming: A survey. *Genetic Programming and Evolvable Machines* **11**(3–4): 365–396.

Olsson R (1995). Inductive functional programming using incremental program transformation. *Artificial Intelligence* **74**(1): 55–83.

Oltean M. (2005). Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation* **13**(3): 387–410.

O'Neill M and Brabazon A (2006). Grammatical swarm: The generation of programs by social programming. *Natural Computing* **5**(4): 443–462.

O'Neill M and Ryan C (2001). Grammatical evolution. *IEEE Transactions on Evolutionary Computation* **5**(4): 349–358.

O'Neill M and Ryan C (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers: Norwell, MA.

Poli R, Langdon WB and McPhee NF (2008). A field guide to genetic programming. lulu.com, freely available at http://www.gp-field-guide.org.uk.

Poli R, Chio CD and Langdon WB (2005a). Exploring extended particle swarms: A genetic programming approach. In: Beyer H-G *et al* (eds). *Proceedings of the Genetic and Evolutionary Computation Conference*. ACM: Washington DC, USA, pp 169–176.

Poli R, Langdon WB and Holland O (2005b). Extending particle swarm optimisation via genetic programming. In: Keijzer M, Tettamanzi A, Collet P, van Hemert J and Tomassini M (eds). *Proceedings of the 8th European Conference on Genetic Programming*. Springer: Lausanne, Switzerland, pp 291–300.

Ross P (2005). Hyper-heuristics. In: Burke EK and Kendall G (eds). *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer: New York, pp 529–556.

Ross P, Schulenburg S, Marin-Blazquez JG and Hart E (2002). Hyper heuristics: Learning to combine simple heuristics in bin packing problems. In: Langdon WB *et al* (eds). *Proceedings of the Genetic and Evolutionary Computation Conference 2002 (GECCO '02)*. Morgan Kaufmann: New York, NY, pp 942–948.

Ross P, Marin-Blazquez JG, Schulenburg S and Hart E (2003). Learning a procedure that can solve hard bin-packing problems: A new GA-based approach to hyperheurstics. In: Cantú-Paz E (ed). *Proceedings of the Genetic and Evolutionary Computation Conference 2003 (GECCO '03)*. Springer-Verlag: Chicago, Illinois, pp 1295–1306.

Tay JC and Ho NB (2008). Evolving dispatching rules using genetic programming for solving multi-objective flexible job-shop problems. *Computers and Industrial Engineering* **54**(3): 453–473.

Terashima-Marin H, Flores-Alvarez EJ and Ross P (2005). Hyper-heuristics and classifier systems for solving 2D-regular cutting stock problems. In: Beyer HG (ed). *Proceeedings of the ACM Genetic and Evolutionary Computation Conference (GECCO'05)*. ACM: Washington, D.C. USA, pp 637–643.

Terashima-Marin H, Farias Zarate CJ, Ross P and Valenzuela-Rendon M (2006). A GA-based method to produce generalized hyper-heuristics for the 2D-regular cutting stock problem. In: Cattolico M (ed). *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'06)*. ACM: Seattle, WA, USA, pp 591–598.

Terashima-Marin H, Farias Zarate CJ, Ross P and Valenzuela-Rendon M (2007). Comparing two models to generate hyper-heuristics for the 2D-regular bin-packing problem. In: Thierens D *et al* (eds). *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*. ACM: London, England, UK, pp 2182–2189.

Terashima-Marin H, Ross P, Farias Zarate CJ, Lopez-Camacho and Valenzuela-Rendon (2010). Generalized hyperheuristics for solving 2D regular and irregular packing problems. *Annals of Operations Research* **179**(1): 369–392.

Received February 2011;
accepted October 2012 after two revisions