Single Node Genetic Programming on Problems with Side Effects

David Jackson

Dept. of Computer Science, University of Liverpool
Liverpool L69 3BX, United Kingdom
djackson@liverpool.ac.uk

Abstract. Single Node Genetic Programming (SNGP) offers a new approach to GP in which every member of the population consists of just a single program node. Operands are formed from other members of the population, and evolution is driven by a hill-climbing approach using a single reversible operator. When the functions being used in the problem are free from side effects, it is possible to make use of a form of dynamic programming, which provides huge efficiency gains. In this research we turn our attention to the use of SNGP when the solution of problems relies on the presence of side effects. We demonstrate that SNGP can still be superior to conventional GP, and examine the role of evolutionary strategies in achieving this.

1 Introduction

Single Node Genetic Programming (SNGP) [1] is a newly-introduced form of GP in which each individual in the population consists of just a single program node drawn from the terminal or function set of the problem we are attempting to solve. The operands for a node are other members of the population. As such, it could be argued that the population forms one large graph; however, we do not treat it as such during evolution. In addition to the attributes encoding connections with other members, each individual has data structures recording the outputs it produces when evaluated; more importantly, it has its own distinct fitness value. It therefore makes much more sense to view the population as a set of graphs, with each individual holding the root node of an expression or program to be evaluated.

This approach is vastly different from other forms of GP, including those in which alternatives to conventional tree-based structures are employed. Most systems treat individuals as distinct, separate structures (although some hierarchical approaches have made use of limited forms of interconnectedness between members). Even when the values of internal graph nodes become important (such as in Oltean's Multi-Expression Programming [2,3]), the population is still constructed as a set of multi-node graphs, each unconnected to the others.

Structural considerations are not the only differences between SNGP and other forms of GP, however. In conventional GP and most other variants, the evolutionary operators are reproduction by cloning, recombination via subtree or segment crossover, and mutation. By contrast SNGP has only one evolutionary operator, and its effects are reversible. SNGP uses a form of hill-climbing in which evolutionary changes which do not lead to improvements are undone.

In the initial experiments we have performed, SNGP has demonstrated substantial improvements over conventional GP in terms of solution rates, execution times, and sizes of solutions obtained. One of the reasons its performance is so good is that it is able to exploit a form of dynamic programming in which the outputs obtained during the evaluation of each node are recorded as an attribute of that node. This means that any function requiring the values of its operand sub-graphs merely has to fetch those values directly from the vectors in which they are stored, without the necessity for further node evaluations.

An approach such as this works well for problems in which the nodes are purely functional and do not have side effects, i.e. it is the outputs of the function that are of interest, and these outputs depend only on the inputs supplied. Examples of such problems in GP are symbolic regression and various Boolean problems such as even-parity.

It is a different matter when problems rely on behavioural side effects. Such problems make use of nodes which modify state or have some form of interaction with an external world. This usually means that behaviour is reliant not just on current inputs but also on previous history. A good example of this type of problem in the context of GP is the Santa Fe artificial ant problem [4], in which the aim is to evolve a program that guides an agent along a trail of ‘food’ particles. In this problem, function outputs are of no relevance. Instead, the fitness of a program is ascertained via the side effects of those functions on a model of the ant’s world. Because of this, the type of dynamic programming previously used in SNGP cannot be employed. Note that this does not necessarily imply that SNGP is not capable of solving such problems, merely that its efficiency will be hampered in doing so, since full evaluation is required of the tree rooted at each individual.

In this paper, then, we investigate the extent to which SNGP is a suitable system for the evolutionary solution of problems with side effects. As we shall see, this work entails re-examination of the evolutionary strategy used to drive SNGP.

2 Related Work

SNGP is, of course, not the first approach to deviate from Koza-style GP [4], in which programs are stored as tree structures and evolutionary operators work by swapping subtrees or replacing them with new, randomly-generated subtrees. In linear GP [5], for example, programs are simply sequences of individual instructions; and whereas tree-based GP takes a functional view of programs, in which calculations are passed up a tree as it is evaluated, linear GP is more akin to conventional imperative programming, with intermediate and final results being stored in registers of memory variables.

A tree is merely one form of a graph, and so it is perhaps not surprising that it is not the only such graph structure that has been tried for GP. One of the first systems to explore this was PADO (Parallel Algorithm Discovery and Orchestration) [6]. PADO makes use of stack memory and indexed memory, and a graph may contain action nodes and branch-decision nodes. The system was used to evolve parallel programs for classifying images.

Taking inspiration from the parallel processing performed in neural networks, Poli's PDGP (Parallel Distributed GP) [7] uses a grid representation to hold graph-structured programs. Individuals are still subject to (suitably modified) crossover and mutation, but programs are more compact than tree-based equivalents, and offer opportunities for concurrent execution. A similar grid-based approach is employed in Cartesian Genetic Programming (CGP) [8], in which the number of rows and columns, and the amount of feed-forward, are all parameters to the system. Originally developed to evolve digital logic designs, the approach made use exclusively of mutation to generate new candidates which took part in a (1+λ) evolutionary strategy, but more recent research has explored the advantages of a new crossover operator [9]. In the GRAPE (GRAph structured Program Evolution) approach [10], graphs contain arbitrarily directed links, and both calculations and node sequencing are determined by a separate data set.

Other researchers have taken conventional tree-based or linear GP and augmented them with additional structures. In linear-tree GP [11], each node of a tree consists of a linear program and a branching node which determines the next node in the tree to be executed. The idea was later extended to more general graph structures [12]. In the MIOST system [13], program trees may contain additional links both to provide more sophisticated interaction between nodes and also to allow multiple outputs from individuals.

In Multi-Expression Programming (MEP) [2,3], each individual has a structure similar to that of single-row CGP, with each node of the graph having links to operands further back in the graph. The main difference is that execution results are computed not only for a program graph as a whole, but also for each of its sub-graphs. The overall fitness of the individual is defined to be the fitness of the best sub-expression. Mutation and crossover are the primary evolutionary operators. As we shall see, an SNGP population can be viewed as being analogous to a single MEP individual, although the mechanics of evolution are very different.

3 The SNGP Model

An SNGP population is a set of N members

$$M = \{m_0, m_1, \ldots, m_{N-1}\}.$$

Each member is a tuple of the form:

$$m_i = < u_i, r_i, S_i, P_i, O_i >$$

where:

$u_i \in \{T \cup F\}$ is a single graph node taken from either the function set F or the terminal set T of the problem;

r_i is the rating of fitness for the individual;

S_i is a set of successors of this node;

P_i is a set of predecessors of the node;

O_i is a vector of outputs generated when this node is evaluated.

During initialisation, the population is partitioned in such a way that:

$u_i \in T$ if $i < TNUM$

$u_i \in F$ otherwise

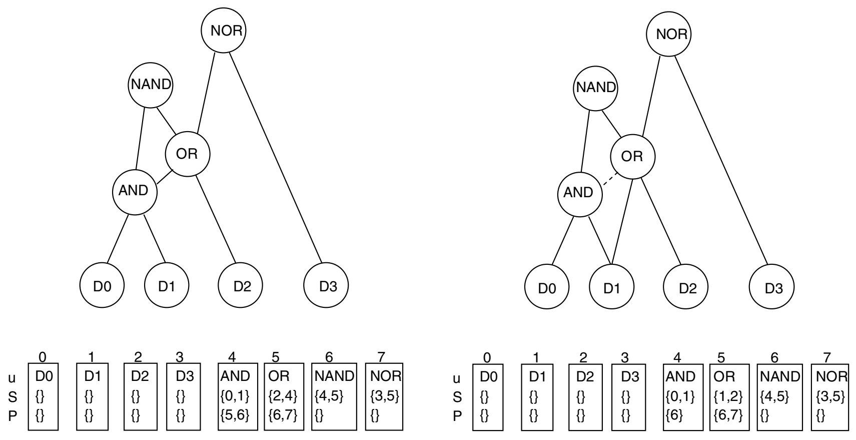
where *TNUM* is the number of terminals in the terminal set. Moreover, for any u_i, u_j such that $i, j < TNUM$ and $i \neq j$, we have $u_i \neq u_j$.

In other words, the first *TNUM* members of the population are initialised to represent the members of the terminal set, with each terminal appearing exactly once. All other members contain nodes drawn from the function set. These are allocated at random, and so may be replicated in the population.

For a population member which represents a function, the operands of that function are drawn from other members of the population. The successor set of the node is a list of the population members acting as operands, represented by their position in the population. We make the restriction that for each $s \in S_i$ we have $0 \leq s < i$, i.e. the operands of a function must be 'lower down' in the population (towards position zero).

Similarly, the predecessors of an individual are those population members for which the individual is used directly as an operand, i.e. they take us to the next higher expression level. This means that for each $p \in P_i$ we have $i < p < N$.

Note that for terminal nodes the successor sets are empty. Moreover, as these nodes cannot change during evolution (see later), their predecessor sets are not needed and are also left empty.



(a) (b)

Fig. 1. SNGP graph structure and effects of the *smut* operator

Fig. 1(a) shows how a population of just 8 members might be initialised, together with the corresponding graph. The first four positions in the population are occupied by terminals, the remainder by functions. For ease of explanation the functions shown here are all different, although in reality functions could be replicated, and certainly will be with larger population sizes. Note that the AND node and the OR node both have two predecessors, i.e. they appear as immediate operands of two other function nodes. This form of reuse is characteristic of SNGP programs, and therefore differs from conventional tree-based GP.

The graph shown here contains eight different expressions, one per node. The simplest expressions are the single-node terminals: D0, D1, D2 and D3. The other expressions are those rooted at the remaining nodes:

AND(D0, D1)
OR(AND(D0, D1), D2)
NAND(AND(D0, D1), OR(AND(D0, D1), D2))
NOR(OR(AND(D0, D1), D2), D3)

It can be seen that, even with only eight nodes, a range of reasonably complex expressions can be encoded. This complexity can rise dramatically when hundreds of nodes are used.

If the type of problem is one in which functions and terminals do not have side effects, then we can employ a form of dynamic programming which substantially enhances the efficiency of SNGP. During initialisation, each terminal is evaluated across all test input cases, and the outputs generated are stored in O_i. These outputs are used to calculate the fitness values r_i. As initialisation continues, and each randomly selected function is inserted into the population, outputs and fitnesses continue to be computed, but making use of the values already stored for the operands forming the successor set. In this way, the fitness calculation for an individual is highly efficient, involving the application of only one operator or function per test case. Of course, when side effects are present, as they are in the problems studied later in this paper, the use of such a mechanism is ruled out, and every node contained in a graph must be evaluated fully.

In SNGP there is only one evolutionary operator, called *smut* (successor mutate). The way that *smut* works is that a member of the population is chosen at random, and then one of its operands (i.e. a member of its successor set) is modified to refer to a different member of the population (but still lower down in the position order). Figure 1(b) shows how this operator is applied. Here, the first operand of the OR node is being changed from population member number 4 (the AND node) to member number 1 (the terminal D1). Hence, the successor set of node 5 must be changed to reflect this, and node 5 must therefore be deleted from the predecessor set of the AND node. In this example, the new operand is a terminal, and so nothing more needs to be done to the graph structure; when the new operand is a function, its predecessor set must also be updated to add in the new parent.

A modification such as this means that the individual which has been changed must be re-evaluated to determine its new outputs and fitness rating. In our example, the expression OR(D1, D2) must be computed for all test cases. However, this will also have an effect on individuals higher up in the population. Exactly which individuals are affected is determined by the predecessor sets. In Figure 1(b), the predecessors of the OR node are the NAND node and the NOR node, and so these must be re-executed. In larger graphs, it may be necessary to continue this chain of execution by pursuing the predecessor references until all affected individuals have been re-assessed.

The order in which evaluations proceed up the population can have a great impact on efficiency. In Figure 1(a), a change to the operands of the AND node might cause the immediate predecessors NAND and OR to be evaluated next. Then, because the OR outputs have changed, the NAND node might be invoked once again. In general, there may be many unnecessary evaluations that take place before the population eventually settles to its final values. To circumvent this, we implement a mechanism in which the predecessor sets are followed to build an ordered ‘update list’ of all affected individuals. We then execute each member of the list in turn, from the lowest to the highest position in the population, thus ensuring that no function is invoked more than once.

Evolution of an SNGP population is driven using a hill-climbing approach. Whenever the smut operator is applied, the fitnesses of the affected individuals are re-assessed. If fitness has not improved, then the modifications made by smut are reversed. To make this more efficient, the old outputs (if outputs are being recorded) and fitness values of each member of the update list are recorded by *smut*, so that they can be put back in place if necessary by a single *restore* operation.

This begs the question of how we determine whether fitness has improved. In the original version of SNGP, this was ascertained by considering fitnesses across the population as a whole. In this strategy, the aim is to drive down the aggregate fitness of the population (and therefore the average fitness). More formally, and assuming that lower fitness values are better, the aim is to minimize Σr_i. One of the things called into question during the research described here is whether that is always the best strategy.

4 Experimentation

For the purposes of evaluation in situations where the dynamic programming approach previously employed is not possible, we have chosen three problems which rely on side effects during program execution.

The first of these problems is the Santa Fe artificial ant problem [4], which is commonly used in assessing the effectiveness of GP algorithms and is known to be difficult to solve [14]. The second test problem we have used is that of navigating a maze. Although less well-known than the ant problem, it has been used as the subject for research on introns in several studies [15-17]. In our third problem, the aim is to evolve programs which are capable of parsing arithmetic and logical expressions. The output of a successful parser is the postfix (Reverse Polish) form of each expression, but the need to manipulate a stack during execution means that functions must rely on side effects to achieve this aim. Full details of this problem can be found elsewhere [18].

The problem parameters as they apply to the use of standard GP in all these problems is given in Table 1. For SNGP there are really only two parameters. The first is the population size (number of nodes), which we have arbitrarily set to 50, although later we will discuss the effects of altering this. The second parameter is the ‘length’ of a run, which we will refer to as *L*. SNGP does not have generations as such; we can think instead in terms of the number of evolutionary operations performed. Since standard GP with a population size of 500 running over 50 generations creates 25,000 individuals via crossover or reproduction, we will set the upper limit on the number of *smut* applications to 25,000.

Table 1. GP system parameters common to all experiments

Population size	500
Initialisation method	Ramped half-and-half
Evolutionary process	Steady state
Selection	5-candidate tournament
No. generations	51 generational equivalents (initial+50)
No. runs	100
Prob. crossover	0.9
Mutation	None
Prob. internal node used as crossover point	0.9

As mentioned in an earlier section, the criterion used in the original version of SNGP to decide whether to reverse the effects of an evolutionary operation is based on the average fitness of the population. If the average fitness (or, to be more precise, the aggregate fitness) worsens, the operation is undone. Henceforth, we will use the notation SNGP/A to refer to the approach when it makes use of an evolutionary strategy based on *Average* fitness. An alternative strategy is to use best fitness, rather than average fitness, as our criterion: if the best fitness in the population worsens, reverse the operation. The term SNGP/B will be used to refer to SNGP when it makes use of a strategy based on *Best* fitness.

Table 2. Comparisons of SNGP with standard GP on example problems

Problem	**System**	**Soln. rate (%)**	**Effort (evals/soln) (x 10^6)**	**Time 100 runs (secs)**	**Av. soln size**	**Max. soln size**	**Min. soln size**
Ant	GP	9	308	50	51	199	25
	SNGP/A	16	2014	298	14	24	7
	SNGP/B	56	260	131	12	21	7
Maze	GP	47	8	7	1987	5620	722
	SNGP/A	98	18	18	33	45	10
	SNGP/B	60	15	9	24	37	9
Parse	GP	32	415	101	399	1154	58
	SNGP/A	45	4406	1073	19	30	9
	SNGP/B	83	355	159	19	34	11

Table 2 compares SNGP against standard GP for our problems. In relation to performance, three measures are used: solution rate, computational effort, and execution time. The solution rate is simply the percentage of full solutions obtained over 100 runs of the problem. For computational effort it would be possible to count fitness evaluations, but since the sizes of SNGP programs and the way in which they are executed differs enormously from standard GP, we chose a different measure that is more reflective of the effort involved. We count the total number of program node evaluations over all runs and divide this by the number of solutions obtained, thus giving a notion of effort per solution. Finally, timings are given for execution of 100 runs. These were taken on a PC with an Intel Core i7 quad-core processor running at 2.8GHz. Both the SNGP and GP systems were written in C and compiled using Microsoft Visual Studio as single-threaded processes running under identical load conditions.

The first thing to note is that, at least in terms of solution rate, both forms of SNGP perform very much better than conventional GP. In comparing SNGP strategies, it would seem that, for two of our problems (artificial ant and parsing), strategy B is the one to opt for. Even though this strategy takes longer to execute 100 runs than standard GP, its much higher solution rate means that less effort is required per solution. Strategy A, on the other hand, requires an inordinate amount of effort to find its solutions, entailing lengthy run times. For the maze navigation problem the situation is perhaps not as clear-cut. Strategy A has a much higher solution rate, and in fact is able to discover solutions on almost every run. However, it requires slightly more effort per solution to achieve this, and requires double the execution time for the runs. The choice of a winner here rests on whether one prefers lots of solutions, or fewer solutions in a faster time.

An important point to make here is that the performance differences between SNGP and standard GP have been verified as statistically significant. This has been done by recording the fitness values of the best programs found in each run (whether forming a solution or not), and then performing a t-test on these data with p=0.05.

Turning to solution sizes, there is no contest. SNGP clearly outperforms standard GP for all three problems, with little to choose from between the two evolutionary strategies. In the case of the maze and regression problems, the solutions found by SNGP are many times smaller than those found by standard GP.

The compactness of the SNGP programs merits further discussion. Since any SNGP program is built only from the nodes contained in the population members, it cannot be larger than the population size. In the experiments described here, this means an upper bound of 50 nodes in any program. That said, the graph-like nature of these programs allows code re-use that is not present in conventional GP trees and which would otherwise require many more nodes to implement. For example, one 40-node SNGP solution to the maze problem would require 4953 nodes if written out as a tree-based GP expression.

Key to the solution sizes obtained is the size of the population, which by definition in SNGP acts as a constraint. In the experiments above, the population size N was set arbitrarily at 50. However, this is not necessarily an optimum for each problem. In contrast to standard GP, one of the advantages of SNGP is that it can find solutions even when N is set very low. Usually this means that fewer solutions will be found, but they will be smaller programs, found in a shorter time. Hence, via the single parameter N, one can tune the system to select an appropriate balance of solution count, program size and execution time. Sometimes, however, this tuning can lead to surprising results. For example, setting N to just 20 for the ant problem using strategy B leads to seven times as many solutions as standard GP, and at a third of the computational cost per solution.

5 Conclusions

Single Node Genetic Programming (SNGP) is a new approach to GP in which each individual in the population consists of just a single program node. Where a node requires operands, these are drawn from other members of the population. Evolution is driven by a hill-climbing mechanism that uses a single reversible operator.

The research described in this paper began as a test of how well SNGP could cope with problems in which the functions and terminals used to build programs rely on side effects for their behaviour. In previous experiments, side effects were not present, and so it was possible to make use of a form of dynamic programming in which the outputs of subtrees could be cached for use as operands by higher-level functions, thereby leading to enormous efficiency gains. Without the opportunity to make use of this mechanism, it was known that efficiency would suffer, but it was hoped that SNGP would be at least competitive with conventional GP systems.

In the event, our experimentation showed that SNGP can be superior in terms of solution-finding performance, computational effort, and solution size, with the caveat that is influenced heavily by the choice of evolutionary strategy. We investigated two strategies: one which promotes average fitness across the population, and the other concentrating on best fitness amongst individuals. In a sense, the first strategy could be described as altruistic, with the other being more selfish. In previous work, the altruistic strategy was generally found to lead to better results. In the work described in this paper, however, the selfish appears better, at least for two of the three problems. What we do not know is why this difference should exist. We hope to carry out further investigations to provide some insight, and in particular to assist us in deciding on a strategy to employ based purely on an *a priori* description of a problem.

Other research lined up for the future includes an investigation into the dynamics of SNGP, to discover how it is able to find solutions so readily with such small populations. We also wish to explore ways to exploit the parallelism inherent in both the SNGP system and in the programs it evolves. And in the same way that we have explored alternative evolutionary strategies here, we want to evaluate the effects of using different operators, initialisation procedures and algorithms.

References

1. Jackson, D.: A New, Node-Focused Model for Genetic Programming. In: Moraglio, A., Silva, S., Krawiec, K., Machado, P., Cotta, C. (eds.) EuroGP 2012. LNCS, vol. 7244, pp. 49–60. Springer, Heidelberg (2012)
2. Oltean, M.: Evolving Digital Circuits using Multi-Expression Programming. In: Zebulum, R.S., et al. (eds.) Proc. 2004 NASA/DoD Conf. on Evolvable Hardware, Seattle, USA, pp. 87–97 (2004)
3. Oltean, M.: Solving Even-Parity Problems using Multi-Expression Programming. In: Chen, C., et al. (eds.) Proc. 7th Joint Conf. on Information Sciences, North Carolina, USA, vol. 1, pp. 295–298 (2003)
4. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)

5. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer, Heidelberg (2007)
6. Teller, A., Veloso, M.: PADO: Learning Tree Structured Algorithms for Orchestration into an Object Recognition System. Technical Report CS-95-101, Department of Computer Science, Carnegie-Mellon University, USA (1995)
7. Poli, R.: Parallel Distributed Genetic Programming. In: Corne, D., et al. (eds.) New Ideas in Optimization, pp. 779–805. McGraw-Hill Ltd., UK (1999)
8. Miller, J.F., Thomson, P.: Cartesian Genetic Programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000)
9. Clegg, J., Walker, J.A., Miller, J.F.: A New Crossover Technique for Cartesian Genetic Programming. In: Thierens, D., et al. (eds.) Proc. Genetic and Evolutionary Computing Conf (GECCO 2007), London, England, UK, pp. 1580–1587 (2007)
10. Shirakawa, S., Ogino, S., Nagao, T.: Graph Structured Program Evolution. In: Thierens, D., et al. (eds.) Proc. Genetic and Evolutionary Computing Conf (GECCO 2007), London, England, UK, pp. 1686–1693 (2007)
11. Kantschik, W., Banzhaf, W.: Linear-Tree GP and Its Comparison with Other GP Structures. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tetamanzi, A.G.B., Langdon, W.B. (eds.) EuroGP 2001. LNCS, vol. 2038, pp. 302–312. Springer, Heidelberg (2001)
12. Kantschik, W., Banzhaf, W.: Linear-Graph GP - A New GP Structure. In: Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B. (eds.) EuroGP 2002. LNCS, vol. 2278, pp. 83–92. Springer, Heidelberg (2002)
13. Galvan-Lopez, E.: Efficient Graph-Based Genetic Programming Representation with Multiple Outputs. International Journal of Automation and Computing 5(1), 81–89 (2008)
14. Langdon, W.B., Poli, R.: Why Ants are Hard. In: Koza, J.R., et al. (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference, pp. 193–201. Morgan Kaufmann (1998)
15. Jackson, D.: Dormant Program Nodes and the Efficiency of Genetic Programming. In: Beyer, H.-G., et al. (eds.) Proc. Genetic and Evolutionary Computing Conf (GECCO 2005), Washington DC, pp. 1745–1751. ACM Press, New York (2005)
16. Langdon, W.B., Soule, T., Poli, R., Foster, J.: The Evolution of Size and Shape. In: Spector, L., et al. (eds.) Advances in Genetic Programming, vol. 3, pp. 163–190. MIT Press, Cambridge (1999)
17. Soule, T.: Code Growth in Genetic Programming. PhD Thesis, University of Idaho (1998)
18. Jackson, D.: Parsing and Translation of Expressions by Genetic Programming. In: Beyer, H.-G., O'Reilly, U.-M. (eds.) Proc. Genetic and Evolutionary Computation Conf (GECCO), Washington, DC, pp. 1681–1688. ACM Press, New York (2005)