Investigations into Memory in Grammatical Evolution

Michael O'Neill

Dept. of Computer Science & Information Systems
University of Limerick
Ireland
Michael.ONeill@ul.ie

Conor Ryan

Dept. of Computer Science & Information Systems
University of Limerick
Ireland
Conor.Ryan@ul.ie

Abstract

We investigate the use of memory in Grammatical Evolution. Previous experiments with memory on the Caching Algorithm problem yielded disappointing results in terms of the exploitation of memory structures in solutions. In order to investigate the use of memory in GE further we need to identify a problem domain in which solutions to problems are dependent upon the use of memory structures. To this end, we report experiments on Teller's Tartarus world in which the use of memory is thought to be essential to the generation of a successfull solution.

1 Introduction

Grammatical Evolution (GE) [13] [11] [9] was previously applied to the real world problem of evolving a Caching Algorithm [10], with the evolved algorithms outperforming the human-designed Least Recently Used strategy. During the course of these experiments a memory facility was made available to the evolving programs, in the form of an indexed memory. The solutions generated by GE did not make use of the indexed memory, instead exploiting the principle of locality of reference, by evolving a solution that selected victims by cycling through the cache memory.

The use of memory in evolutionary automatic programming is widely accepted as being beneficial to the evolution of code to more complex problems, e.g., [14] [8] [7] [15] [1] [3] [5]. This comes as no surprise to programers who routinely use various memory structures such as stacks, queues, registers, etc. Indeed Koza et al., have identified internal storage as one of the necessary attributes for an automatic programming system [7]. As such, we wish make memory structures available to GE, which can be easily achieved through the grammar.

An overview of the Cache Algorithm experiments follows. The application of GE to Teller's Tartarus problem is then described in which an indexed memory is made available.

2 Caching Algorithm's

A cache memory is a small, fast memory, ideally into which the *address space* of a running program is held in order to improve efficiency of the program's execution. Caches operate on the notion of *locality of reference*; that is, if a word in memory has recently been accessed, it is likely to be required again soon. The executing program generates a stream of *requests* of main memory locations to access. A recording of this request stream is referred to as a *trace*. The cache has a fixed number of *lines*, each of which can hold a word. When a *request* is met from the cache memory this is called a *hit*, otherwise a *miss* occurs. In the event of a miss, the requested memory location is copied into the cache to the first empty *line* available, otherwise a caching algorithm determines which line of the cache will be replaced (the *victim*). In order that the caching algorithm can make an educated decision as to which line will become the victim, typically some management information is available, e.g. an ordered list of lines.

For the purposes of these experiments, we adhered as closely as possible to the procedures outlined in [12], in order to facilitate a comparison of the systems. Thus, the same trace files, a similar grammar (no default or multiple rules were used) with the same terminal set, an integer management array $info[\ ]$ (one element per line), and the same fitness function were used. This indexed memory has the same size as the Cache.

The raw fitness of a caching algorithm is therefore given as "$number\ of\ runs - number\ of\ misses$" where a run is a consecutive series of identical requests. The terminal operators and their respective functions are given in Table 1.

Terminal Operator	Function
$write\_x(i, v)$:	sets info[i] to v
$read\_x(i)$:	returns info[i]
$small\_x(i, v)$:	index of smallest element of info[]
$large\_x(i, v)$:	index of largest element of info[]
$random\_x(i, v)$:	index of random element of info[]
$counter()$:	successive values 0, 1, 2 etc
$div(x, y)$:	if y==0 then 1 else x / y
$rem(x, y)$:	if y==0 then 1 else x % y

Table 1: Available terminal operators.

2.1 Results

Similar to Paterson & Livesley [12], we employ three trace files. One file is used for training GE, while the resulting algorithms are tested using the other two. The training file `ken2.00200` contains around 400,000 requests with about 250,000 distinct addresses. Paterson & Livesley reduced memory requirements by eliminating the lower order 10 bits of each addresses, thus reducing the alphabet size to just 500, i.e. 500 distinct addresses as opposed to 250,000 originally. We use the files in their original state, and show comparisons with Paterson & Livesley's algorithm below. Paterson & Livesley found that the speed of the cache simulator frustrated his attempts to run many experiments, and we suffered the same experience. To alleviate this, we use a population of just 50 individuals, compared to his 500. For each of the experiments that follow, ten runs were conducted, and the best individual at the end of twenty generations was analysed.

Small Cachesize The first experiment consists of using a cache size of just 20, clearly, it is very difficult to exploit any locality of reference with such a small size. The best individual generated by GE was

```
GE 1 :  victim = counter() - CACHESIZE;
```

This curious looking individual simply cycles through the cache when searching for a victim, selecting it without consideration for how recently it was used. There is, however, an implicit temporal quality about the manner in which victims are selected. A word is guaranteed to remain in the cache for at least $CACHESIZE$ requests, longer if it (or other cache members) is requested. This outperforms the human designed Least Recently Used algorithm (LRU) by 15.7%, and, by virtue of its simplicity, is far less expensive to use.

Large Cachesize A cachesize of 200 lines was used in this instance, and one of the best individuals produced was again GE 1. However, in this case the number of misses was far less, outperforming LRU by 46.2%.

Using info [ ] Given the level of disinterest exhibited by GE to the use of the info[] indexed memory, we ran an experiment that forced it to write into some element before choosing its victim. The grammar in this case varied from the one outlined above in the case of the production rules for the non-terminals `stmts` and `stmt`. Both of these rules were replaced with the following:

```
(1)   <stmts> :: = write_x(<expr>,<expr>);
                   victim=<expr>;
```

The best of run individual generated by this system was:

```
GE 2 :
write_x( CACHESIZE + counter(),
         CACHESIZE + counter() )

victim = CACHESIZE + counter();
```

Rather predictably, this individual simply ignores what it writes into the memory, and promptly chooses its victim using the same method as in the other experiments. In fact, the ubiquity of the beautifully simple $CACHESIZE + counter()$ algorithm is demonstrated by virtue of its appearance in no less than three parts of this algorithm.

Comparison of Algorithms A comparison between the LRU algorithm and those which had been evolved was carried out by rating their performance on the other two trace files, and the number of misses obtained in each case can be seen in Table 2.

Paterson & Livesley were dismayed to report that the algorithms they produced were very poor at generalising to unseen data, and that they suffered greatly from overfitting. The algorithms produced by GE don't suffer from this problem, and perform extremely well on the new data.

One factor that might contribute to the large difference in performance was Paterson & Livesley's decision to simplify the problem - the problem as posed to GE was possibly too large to spot any patterns in.

The solution generated by GE was so general that when applied to a larger cache size, a huge increase in performance was observed. When the cache size was increased by an order of magnitude for LRU, the accompanying increase in performance shows that the associated cost of such an increase would probably not pay off.

Simplified Stream Data Another set of experiments were conducted to determine if using the simplified data stream as adopted in [12] would have a detrimental effect on the generalisation capabilities of solutions found by GE. Similarly to Paterson & Livesley we removed the low-order 10 bits from each address in the training stream file, thus reducing the alphabet size to 500 as opposed to 250,000 in the original stream.

Experiments with the small cachesize, large cachesize and forcing the solutions to write to the info[] memory were conducted, and in each case GE re-discovered the solutions found in the original experiments. Simplifying the training data set thus had no effect on the generalisation capabilities of GE. Again, when the info[] array use was forced the solutions did not utilise this memory as part of the solution.

2.2 Discussion

From the results described above it is disappointing and clear that GE failed to use the allocated memory structure available. Instead GE exploited the problem environment to evolve an elegant solution that did not require the use of memory. We now seek to find a problem in which the use of memory is essential to generating a solution.

Algorithm (Cachesize)	ken2.00100 (Misses)	ken2.00200 (Misses)	Average of % Improvement over LRU
LRU (20)	374,596	380,041	-
LRU (200)	367,104	373,935	-
GE1 (20)	300,569	318,444	17.97
GE1 (200)	106,067	82,856	74.51
GE2 (20)	300,569	318,445	17.97
GE2 (200)	106,068	82,855	74.51

Table 2: Algorithm performance comparison.

3 Tartarus

Tartarus was adopted as described in [14], that is a 6x6 grid, with 6 boxes, where a wall exists at each of the four grid edges. An agent exists in this grid with the goal of pushing all boxes onto the egdes of the grid, scoring 1 point for each box that lies against a wall, and 2 points for boxes that lie in a corner. The agent has three possible actions, MoveForward, TurnLeft, and TurnRight, and eight input sensors (one for each neighbouring grid). Each sensor can take on one of three values indicating the existence of a clear grid cell, a box, or a wall. In order to prevent the agent from exhaustively moving around the world, a time limit of 80 steps is imposed, where each action takes one time step to execute. When the agent first appears in the world it is placed in a random clear grid cell, facing in a random direction, the boxes are also placed randomly for each fitness case.

Two grammars were adopted; the first which explicity allows the use of an indexed memory of size 20, the second removes the ability to use the indexed memory.

Grammar with Memory

```
code : line | code line
line : op ; | memop ; | cndtn
op : c->moveForward() | c->turnLeft() | c->turnRight()
memop : read ( address ) | write ( address , val )
cndtn : if ( test ) { code } else { code }
test : val relbi val
val : read ( address ) | sensor | int | address
sensor : misense ( ( int % 8 ) , c )
int : intexpr | 1 | 2 | 3 | 4 | 5 | 10
intexpr : int matbi int
matbi : + | - | *
relbi : > | < | >= | <=
address : ( int % MEMORY_SIZE )
```

Grammar without Memory

```
code : line | code line
line : op ; | cndtn
op : c->moveForward() | c->turnLeft() | c->turnRight()
cndtn : if ( test ) { code } else { code }
test : val relbi val
val : sensor | int
sensor : misense ( ( int % 8 ) , c )
int : intexpr | 1 | 2 | 3 | 4 | 5 | 10
intexpr : int matbi int
matbi : + | - | *
relbi : > | < | >= | <=
```

4 Results

We ran 30 runs for each grammar on the Tartarus problem where the fitness function took an average of 40 randomly generated fitness cases (a new randow set was generated at each generation).

Graphs depiciting the performance of GE on Tartarus can be seen in Figure 1. It can be seen here, and has been verified using statistical analysis (using a ttest and re-sampling ttest), that the difference between the performances with and without the availablility of the indexed memory structure is insignificant.

The results were surprising in that the individuals that had the ability to use the indexed memory in the generation of a solution at no stage exploited the potential benefits that this conferred. Instead it appears that evolution was able to determine the size of the world implicitly and exploit this by the number of move operations that are called in sequence such that the better solutions were able to push the blocks far enough so that they would be lying against a wall even if the agent started from the far side of the world. These types of solutions evolved in the case of both grammars analysed.

In [15] it has been previously found that it is possible to solve Tartarus using implicit state in a fashion similar to the solutions evolved here, i.e., agents performed a *dance*, through a fixed sequence of actions. [2] exploited this fact in a novel representation using If-Statement-Action sequences, called ISAc lists, to evolve higher scoring agents.

5 Conclusions & Future Work

Experiments on the evolution of Caching algorithms, while yielding interesting results, failed to make use of the available memory structure in successful solutions. In this paper we tackled a problem for which memory is thought to be essential to finding successful solutions, in order to investigate the use of memory in Grammatical Evolution. Curiously, results demonstrate that solutions to the problem were found that did not use the indexed memory structure available. Instead solutions exploited the size of the environment to dictate how to sequence the number of move instructions so that a block that exists on the same axis as the agent will always be forced against a wall.

We are left at the point where we started in this investigation, we must identify other problems in which the use of memory is essential to finding successfull solutions. To this end we are currently looking at the problems tackled by Bill Langdon in [8], in particular our current efforts are focusing on the Bal-

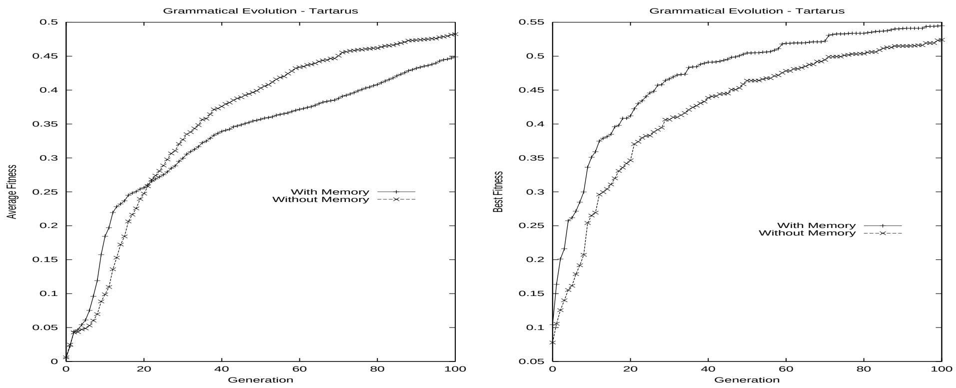

Figure 1: A plot of the average and best fitness at each generation with and without the availability of memory on the Tartarus problem. It can be seen that there is no significant difference between performances.

anced Bracket, and Dyck language problems. An additional problem suggested in [15], Jacobi's T-Maze [6], is also under consideration.

Acknowledgement

The authors thank Adrian Trenamen for the use of his Tartarus code, and for many inspiring discussions.

References

[1] Peter J. Angeline. An alternative to indexed memory for evolving programs with explicit state representations. In *GP 1997: Proc. of the Second Annual Conference*, pp. 423–430, Stanford University, CA, USA, July 1997. Morgan Kaufmann.

[2] Dan Ashlock and Mark Joenks. ISAc lists, A different representation for program induction. In *GP 1998: Proc. of the Third Annual Conference*, pp. 3–10, University of Wisconsin, Madison, Wisconsin, USA, July 1998. Morgan Kaufmann.

[3] Forrest H Bennett III. A multi-skilled robot that recognizes and responds to different problem environments. In *GP 1997: Proc. of the Second Annual Conference*, pp. 44–51, Stanford University, CA, USA, July 1997. Morgan Kaufmann.

[4] Scott Brave. The evolution of memory and mental models using genetic programming. In *GP 1996: Proc. of the First Annual Conference*, pp. 261–266, Stanford University, CA, USA, July 1996. MIT Press.

[5] Daniel Howard, Simon C. Roberts, and Conor Ryan. Evolution of an object detection ant for image analysis. In *GEECO 2001 Late Breaking Papers*, pp. 168–175, San Francisco, California, USA, July 2001.

[6] Nick Jacobi. Evolutionary robotics and the radical-envelope-of-noise hypothesis. *Adaptive Behaviour*, 6:131–174, 1997.

[7] John R. Koza, David Andre, Forrest H Bennett III, and Martin Keane. *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman, 1999.

[8] William B. Langdon. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!*, volume 1 of *Genetic Programming*. Kluwer, 1998.

[9] M. O'Neill. *Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution*. PhD thesis, University of Limerick, 2001.

[10] M. O'Neill and C. Ryan. Automatic generation of caching algorithms. In *Proc. of EUROGEN99*, pp. 127–134, Jyväskylä, Finland, 1999. University of Jyväskylä.

[11] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4), 2001.

[12] Norman Paterson and Mike Livesey. Evolving caching algorithms in C by genetic programming. In *GP 1997: Proc. of the Second Annual Conference*, pp. 262–267, Stanford University, CA, USA, July 1997. Morgan Kaufmann.

[13] Conor Ryan, J. J. Collins, and Michael O Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proc. of EuroGP'98*, volume 1391 of *LNCS*, pp. 83–95, Paris, April 1998. Springer-Verlag.

[14] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pp. 199–219. MIT Press, 1994.

[15] Adrian Trenaman. Concurrent genetic programming, tartarus and dancing agents. In *Proc. of EuroGP'99*, volume 1598 of *LNCS*, pp. 270–282, Goteborg, Sweden, May 1999. Springer-Verlag.