Brute Force is not Ignorance

Joseph Davidson and Greg Michaelson

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, Scotland
{jrd5/G.Michaelson}@hw.ac.uk

Abstract. Chaitin's notion of program elegance, that is of the smallest program to satisfy some specification, does not explicitly take account of the balance between a formal notation's expressive power and the richness of its semantics. To explore this wider space of elegance, we are investigating realisations of the Busy Beaver game (BBG) which involves finding programs of given size that produce maximal outputs.
Like elegance, BBG is undecidable. Canonically, BBG is represented using Turing machines, but there has been very little investigation into alternative formulations. Thus, we are exploring BBG in a number of classic models of computability, using empirical and analytic heuristics to find optimal BBG instances. Such experiments will be used as a basis for drawing comparisons between the expressive power of Turing machines and other models of computation with a view to gaining a deeper understanding of the expressive power of computer languages.
In this paper, we re-introduce the Random Access Stored Program machine (RASP) and build an analogue of the BBG for these machines. Though the BBG for any fixed precision RASP model is trivially computable, the expressivity of the model renders exhaustive search infeasible as we increase the size of our machine. Thus, we explore the space of BBG RASP machines using both brute force and genetic search methods.

Keywords: Elegance, Busy Beaver, Computability, Genetic Algorithms, Brute Force, RASP Machine

1 Motivation

Chaitin has extensively investigated the concept of elegant programs [3]. A program P, calculating a function f is said to be elegant if there is no other program which both calculates f and is smaller (by source code characters) than P.

Using Lisp, Chaitin has produced a contradictory program *elegantFinder* which is combined with the rules to some formal axiomatic system (FAS) FAS. Say we wish to find the shortest expression F for a function f. We invoke *elegantFinder* and FAS such that they enumerate all programs in size order and test them each to see if they calculate f and are therefore F.

Once *elegantFinder* has found F, it runs F in order to obtain its value and returns it. In doing this, we can see that *elegantFinder* also calculates f.

This becomes paradoxical when we consider the relative sizes of the programs involved. If F is smaller (in number of characters) than $|elegantFinder|+|FAS|$, then our elegant program is F. However, if F is larger (possibly calculating a very complex f), then because $elegantFinder$ runs F once it has been found, it turns out that $elegantFinder$ is actually our elegant program for f!

The problem of finding a our elegant program F is quite convincingly uncomputable, however it also raises some related questions about the expressive power of languages. Taking the size of interpreters/compilers into account, is a function in a high-level language like Lisp any smaller than in a lower level language like C, or even an Assembly-like language? Intuitively, we can see that lower level languages or models require simpler 'back end' infrastructure to be executed by the commonplace Von Neumann architecture, but is there a sweet spot in the ratio of program complexity to interpreter complexity where we can develop short programs and have them execute/compile quickly? And if there is, can we deduce or prove where that spot is and develop an 'elegant language' to take advantage?

Elegance is undecidable. However, we hypothesise that by realising the same computations in different models of computation, we can obtain heuristic information that allows us to draw useful comparisons of elegance. Thus, this paper discusses the implementation of the Busy Beaver game (BBG) in the Random Access Stored Program (RASP) architecture – which is similar to the RAM machine – and considers the size of the resulting search space and attempts to navigate it. The BBG itself is a problem in which every solution is elegant (we can phrase it as looking for the smallest machine to output a given integer), so producing this problem in both representations will help give us the experimental data that we need.

2 The Busy Beaver Game

The 'Busy beaver game', was formulated by Radó [8] in order to show an example of a simple undecidable problem. It is the problem of defining an instance of a Turing machine such that, when started on a blank tape it produces as much output as possible before halting. The Turing machine is the canonical model for exploring the busy beavers and it is what the majority of the hunters use due to its well defined operations and the relative simplicity of instance generation. When we limit our machines to a particular size of states – and symbols when Brady extended his machine in 1988 [1] – we create a competition between machine designers to find the 'champion machine' for that size. The 'champion machine' for a class is defined as the machine which either produces the most output or performs the highest number of shifts before halting, compared to other machines of that class.

Radó formalises his game using n 'state' Turing machines and uses this as a method of classifying the machines. He defines a valid entry into the BB-n classification as a pair (M, s) where M is the Turing machine of n states and s is the exact number of steps that M has run for before halting.

There are two competitions in the BBG: The greatest number of shifts, and the greatest number of non blank symbols left on the tape. We characterise the shift number for a given machine as s, and the maximum shifts for a class as S. Likewise, the non-blank symbols for an individual machine is σ and for a class, we use Σ.

This method gives us an easy way to validate and score an entry, we simply run M for s steps and – if the machine has halted – record the number of non blank symbols on the tape. If the output is larger than the previous record, we call M our new champion.

After Brady extended the game to machines with more than two symbols [7], a new metric for classifying machines was used. Instead of Radós original BB-n classification, we now classify a BBG entrant as a machine BB(n, k) which is a Turing machine with n states and k symbols.

Present Landscape. At the time of writing, 4 classes of busy beaver machines have had confirmed S and Σ scores with machines to match: BB(1,2), BB(2,2) BB(3,2) and BB(4,2). Marxen and Buntrock [6] have established lower bounds for BB(5,2) at $S(5, 2) \geq 47,176,870$ and $\Sigma(5, 2) \geq 4098$.

The father and son team of Terry and Shawn Ligocki have made progress in exploring the space of machines with more than 2 symbols by using simulated annealing techniques to obtain high scoring machines. They currently hold the record for many of these classes.

Table 1, compiled by Pascal Michel [7] shows the current records for a few of the classes as of June 2012.

Table 1. Currently known lower bounds of the explored classes.

Date	**Discoverer(s)**	**Bounds**
1963	Radó, Lin	$S(2,2) = 6,\ \Sigma(2,2) = 4$ $S(3,2) = 21,\ \Sigma(3,2) = 6$
1964	Brady	$S(4,2) = 107,\ \Sigma(4,2) = 13$
February 1990	Marxen, Buntrock	$S(5,2) \geq 47,176,870,\ \Sigma(5,2) \geq 4098$
February 2005	T. and S. Ligocki	$S(2,4) \geq 40,737,\ \Sigma(2,4) \geq 3,932,964$
November 2007	T. and S. Ligocki	$S(3,3) \geq 119,112,334,170,342,540, \Sigma(3,3) \geq 374,676,383$ $S(2,5) > 1.9 \times 10^{704}$,$\Sigma(2,5) > 1.7 \times 10^{352}$
December 2007	T. and S. Ligocki	$S(3,4) > 5.2 \times 10^{13036}, \Sigma(3,4) > 3.7 \times 10^{6518}$
January 2008	T. and S. Ligocki	$S(4,3) > 1 \times 10^{14072}, \Sigma(4,3) > 1.3 \times 10^{7936}$ $S(2,6) > 2.4 \times 10^{9866}, \Sigma(2,6) > 1.9 \times 10^{4933}$
June 2010	Kropitz	$S(6,2) > 7.4 \times 10^{36534}, \Sigma(6,2) > 3.4 \times 10^{18267}$

3 The Random Access Stored Program Model

The Random Access Stored Program (RASP) machine model from Elgot and Robinson [4], is a computational model similar to the Random Access Memory

(RAM) machine but with the distinction that the internal variables of the state machine, as well as the program itself, is stored in memory along with the data.

This idea that the machine only has a single block of memory means that a program can potentially read and write to any address in the memory – which allows for programs which modify themselves while they are being executed. The

A RASP machine is arranged as an array M of size l, where each location (or register) in M can hold a single natural number and each register is referred to by a natural number address. The state machine uses the first 3 registers for specific tasks – although that doesn't prevent any external read/write operation to take place. The top 3 registers of M are defined to be: the program counter (PC), the instruction register (IR), and the accumulator (ACC). The PC contains the memory address of the current instruction, the IR is used for decoding the instruction, and the ACC is the register which the numerical instructions modify and which is tested by the conditional instruction.

We can define the RASP machine to be either bounded or unbounded depending on how concrete we desire the instance of the model to be. An unbounded RASP machine is of an arbitrary size and each register can hold a natural number of unbounded size. The bounded RASP is specified in n bits, the size of the machine is set as 2^n registers numbered 0 to $2^n - 1$, each register can also hold a natural number x in the range $0 \leq x < 2^n$.

An equivalent definition is as a pair (N, K), where N is the number of registers in the machine and K is the maximum integer that can be expressed. This allows us to easily define RASP machines with unbounded registers which can hold finite integers (∞, K), or finite registers which themselves are unbounded (N, ∞). For the rest of this paper – unless otherwise stated – RASP "n-bit machines" will be of the form $(2^n, 2^n - 1)$

In the bounded RASP, over and underflow of a register is not treated as a special case. For example, when the machine executes the final instruction in the memory, the next increment of the PC (currently set to $2^n - 1$) will overflow it to 0 and the running of the machine continues as normal.

RASP Instructions and Programs. A RASP machine program is a sequence of natural numbers of length $\leq 2^n - 3$. When the machine is instantiated, the top 3 registers are set to (3 0 0), which is a zeroed IR and ACC and the PC is pointing to the first line of the program.

The execution of instructions by the machine follows the standard fetch-decode-execute cycle:

1. Copy into the IR, the instruction contained in the memory location pointed to by the contents of the PC.
2. Read the IR and decode which instruction the contents refer to.
   (a) If the instruction is not recognised – halt.
   (b) If the instruction takes a parameter – increment the PC and repeat step 1 to obtain it.
3. Execute the instruction.
4. Increment the PC, if the PC becomes equal to K – overflow to 0.

The instructions that can be executed by the machine are listed below.

- HALT – Halt execution.
- INC – Increment the accumulator.
- DEC – Decrement the accumulator.
- LOAD c – Load the value c into the accumulator.
- STO m – Store the value of the accumulator in register m.
- JGZ m – Set the PC to the value m if the accumulator is greater than zero.
- OUT – print the character '1'.
- CPY m – Copy the contents of register m into the accumulator.

LOAD, STO, JGZ, and CPY also require an operand which is assumed to be located in the proceeding memory location. In the case of these instructions, another fetch is performed in order to retrieve the operand. If the RASP machine attempts to interpret an integer which isn't an instruction, it halts as if it has executed the HALT instruction.

The state machine for the RASP machine uses a map to match an opcode to one of the above instructions. This allows us to produce arbitrary injective mappings between instructions and the natural numbers. The instruction set mapping is defined as a map $M : \mathbb{N} \mapsto \mathbb{INS}$ where
$\mathbb{INS} = \{HALT, INC, DEC, LOAD, STO, OUTJGZ, CPY\}$.

4 RASP as a Busy Beaver

In a RASP machine, the command 'OUT' can be thought of as printing a symbol to an attached screen, or writing to a write-only output tape. This idea invites us to draw comparisons between the σ and s functions for RASP and TMs. With a RASP machine R, we define $\sigma(R)$ to be the number of times 'OUT' has been executed and $s(R)$ as the number of fetch-decode-execute cycles that have been performed in total. In addition to this, the number of bits we specify for the bounded RASP machine defines natural classes of machines for the competition.

We define an entry into the BBG RASP (BB-$R(n)$) competition, as a pair (P, IS) where P is the program of size n and IS is the instruction set mapping.

Observations on RASP Machines. In using this model to play the BBG, we can make some observances on the nature of the machines which provide some insight into how easy searching for the champion machines in a class will be.

Theorem 1. *The halting problem for the bounded RASP machine is decidable.*

Proof. Consider a bounded n-bit RASP machine M. We define the state of M to be the entire memory at a particular time, and each fetch-decode-execute cycle as a transition from one state to another. Since there is only a finite range of values for a finite number of memory locations, we can calculate the maximum number of possible states for any given machine $numStates(n) = N^N$.

Because each fetch-decode-execute cycle performs a transition between states $S \to S'$ we can run the machine for $numStates(n)$ cycles before concluding that for some state X which is entered during execution of the machine, there exists a transitive closure over a relation R such that XR^+X. From which we can conclude that M will never halt. □

In practice, we rarely need to run a machine for $numStates(n)$ steps before we can work out if it halts or not. It suffices to store each visited state as it is encountered and check the store for the new state after every state transition, if we encounter the same state twice, a loop has occurred. Different instruction set mappings behave differently; because RASP programs are simultaneously programs and data – the machine can attempt to execute any part of the memory as if it were all instructions, and it can read/write to any part of the memory as if it were all data.

This means that the magnitude of the value which represents certain instructions can affect the s or σ value of the machines. As an example, consider tables 2 and 3.

The program in table 2 is the best program that can be found for the specified instruction set. It has scores of $S(3) \geq 33$ and $\Sigma(3) \geq 16$. Table 3 is the champion machine for $BB - R(3)$ which was found through brute force searching of the entire space of program/instruction set combinations and it presents with scores of $S(3) = 82$ and $\Sigma(3) = 47$.

Table 2. The best 3-bit machine with the instruction set $\{0 \mapsto HALT, 1 \mapsto INC, 2 \mapsto DEC, 3 \mapsto LOAD, 4 \mapsto STO, 5 \mapsto OUT, 6 \mapsto JGZ, 7 \mapsto CPY\}$

Memory Address(es)	**Integer**	**Instruction**	**Label**
0	3		:PC
1	0		:IR
2	0		:ACC
3	1	INC	:start
4	5	OUT	
5	5	OUT	
6	6	JGZ	
7	3	LOAD	

The number of individual machines arising from these program/instruction set combinations grows very quickly. The number of unique instruction sets for a given n-bit machine is

$$\prod_{N-8<i\leq N} i \tag{1}$$

To calculate the number of programs, we recognise that the IR and ACC are initialised to 0 and that the PC points to the first command in the program in memory address 3. Since these are constant in the initial state of every program, we need to calculate the number of possible base N numbers of length $N - 3$.

If we consider each register to contain a digit of the base N number, the number of possible machines with a starting state of (3 0 0) comes to $(2^n)^{2^n-3}$.

Table 3. The best 3-bit machine with the instruction set $\{0 \mapsto OUT, 1 \mapsto LOAD, 2 \mapsto DEC, 3 \mapsto INC, 4 \mapsto CPY, 5 \mapsto STO, 6 \mapsto HALT, 7 \mapsto JGZ\}$

Memory Address(es)	**Integer**	**Instruction**	**Label**
0	3		:PC
1	0		:IR
2	0		:ACC
3	3	INC	:start
4	3	INC	
5	0	OUT	
6	0	OUT	
7	3	INC	

Like the number of possible instruction sets, this number grows quickly. Indeed, the space of machines becomes rapidly intractable for all but the smallest machines ($n < 4$). Combining these two values demonstrates the intractability of brute forcing a result. As an example, for 4 bit machines

$$\begin{aligned}(2^4)^{2^4-3} \times \prod_{8<i\leq 16} i &= 4,503,599,627,370,496 \times 518,918,400 \\ &= 2,337,000,712,875,693,991,526,400 \end{aligned} \tag{2}$$

5 Searching for the Best

A parallelised version of the brute force algorithm was implemented to find the champion machine for 3-bit machines which is shown in table 3. The brute force algorithm generates all the possible instruction sets and dispatches even chunks of these to the workers. Once a worker has determined their best machine, it is returned to the master which determines the very best and returns it. This works for finding the best 3-bit machine however is hopelessly inadequate in tackling the sheer volume of cases which arises from using more than 3 bits. A genetic algorithm (GA) [5] approach was therefore employed so that we can use an informed search to explore the space of machines.

There are a myriad of methods with which to explore the space, however we have observed that the search space tends to be very volatile. A neighbour solution which differs by a single instruction or a slight change to the instruction set mappings can produce very different results. As such (meta)heuristic methods of searching the space – for instance, local search – were though to be too sensitive to these changes. A more global search method was decided upon as genetic algorithms. Simulated annealing – as employed by the Ligockis – would also have been an acceptable alternative and experimentation with that method .

Overview of the Algorithm. The algorithm initialises a random pool of n-bit machines fitting the constraints for RASP machines set out above and proceeds

to determine their fitness. The fitness function is the number of OUT commands executed, but if the machine is shown not to halt (by using the loop detection strategy mentioned after the above decidability proof) then the fitness of the machine is defined to be 0.

Once the machines have all been executed, roulette selection [5] is performed to select a group of machines to populate the next generation. The machines not selected are removed from the pool, and the selected ones are preserved and randomly crossed in order to produce new machines to fill the pool.

Crossing involves first picking two parents, then a method of reproduction – crossing program only, instruction set only, or both. In the case of not crossing both, a 'dominant' parent is randomly selected which is used to fill in the non-crossed attribute of the machine by directly copying it into the machine.

The crossing itself is done by picking a point on the program or instruction set – which are our chromosomes represented as a vector in memory – and combining the left side of the chromosome from parent A (which is arbitrarily chosen) with the right side of the chromosome from parent B. In the case of the instruction set, if there is a duplicate entry such that the map is no longer injective then substitute integers are randomly generated for the duplicates until the map is injective again.

There is also a random chance of mutation to the instruction set or program which will pick two values in the program or instruction set of the child machine and swap their positions.

Parallelisation of the algorithm was through means of an 'isolated island' [5] approach where each processor in the computation has their own pool and returns their best found machine to the master processor which sorts them and returns the overall best of the computation.

For the 4 and 5 bit machines, the generation of high scoring machines was helped along by seeding the pool with a previously found high scoring machine at the pool initialisation stage. It became a good strategy to seed the current champion so that the algorithm can attempt to improve on it.

Current Results. The parallel brute force algorithm calculating the best 3-bit machine was benchmarked on a lightly loaded dual Intel Xeon E5506 machine running at 2.13GHz. This setup provides 8 cores which were apportioned as 7 worker processors and a master.

The times (seconds) taken across 3 runs were 1740, 1743, and 1678 for an average runtime of 1719 seconds. Each of these machines were executed for an average of 5 fetch-execute cycles each. This is due to the loop detection system. It appears that most RASP machines either halt or enter an infinite loop early – which is similar to the findings of [2] whereby most programs either halt quickly or loop forever. From a random sampling of 8,402,100,000 4 bit-machines, our average number of fetch execution cycles is 1.8 which translates into around 305,045 machines per second.

This is a very rough approximation as the sample size is tiny compared to the space. Given our calculated number of 4 bit machines in (2), we can see that

this would take a long time - assuming 256 workers, it will take approximately 948 million years - which underlines our need for informed search methods.

Table 4. Current records for numbers of shifts and outputs.

Bits	Instruction Set	Results	Comments
3	{6,3,2,1,5,0,7,4}	$S(3) = 82, \Sigma(3) = 47$	Exact values found through brute force searching.
4	{2,6,7,5,1,3,4,0}	$S(4) \geq 2668, \Sigma(4) \geq 1483$	Genetic, Pool: 100000, Generations: 1000, Islands: 32
5	{7,1,0,2,4,3,6,5}	$S(5) \geq 7540865, \Sigma(5) \geq 5242881$	Genetic, Pool: 100000, Generations: 500, Islands: 32
5	{4,7,1,3,5,2,6,0}	$S(5) \geq 235652, \Sigma(5) \geq 163846$	This machine was constructed by hand using 3 nested loops.

The experiment confirms that the number of machines grows very rapidly with the size of the machine word The experiment also confirms that the largest BBG solution found grows very rapidly with the size of machine word.

The best 5 bit RASP found by the GA (5,242,881) is 50 times better than a good seed machine with 3 nested loops constructed by hand (163,846). From inspection, the evolved machine makes considerable use of self-modification, with programming constructs which are very hard to characterise succinctly.

6 Conclusions, Reflections and Further Work

We have introduced our RASP model variant and ported the BBG over to it. From the current champions, we can see that they heavily leverage the ability of RASP machines to self-modify in order to produce high shift and output counts.

The halting problem for the bounded RASP machine has been shown to be decidable, but since both the size of the machine and the range of natural numbers expressible by the machine double with each successive class; the space of possible champions explodes. This renders exhaustive searching of all the candidates too time consuming to consider so we produced a genetic algorithm to perform an informed search for the champions.

The resulting GA has found and improved on candidates for the champion machines in the 4 and 5 bit classes. However the current GA tends to be weak with respect to the diversity of the gene pool. Thus, the current best machine tends to dominate the pool within a few generations which leads to the GA only attempting variations of this machine and mostly finding highly local optima.

Adapting the reproduction and mutation mechanics can allow for more diversity (i.e. reproduction eliminates breeding parents after replenishing the pool). The isolated island approach also unduly constrained the search procedure. Using a 'true' island approach, diversity can be introduced through migration of a few best scoring machines to a limited number of adjacent processes.

Aside from improvements to the GA, finding optimal BBG solutions for RASP machines can benefit from methods of classifying high scoring or non-halting machines. There may be distinguishing patterns of machines which can augment a brute force approach by allowing us to discount large numbers of machines as being suboptimal or non-halting. The observation that a large number of non-halting machines are shown to quickly enter loops implies there is a pattern which we can discern and use to our advantage.

We now have BBG expressed in TMs and RASP machines, and parallel GA machinery to explore both spaces. We have also implemented a Universal TM and a RASP interpreter as both TMs and RASP machines. We next plan to construct compilers from TM to RASP and from RASP to TM in both TM and RASP. This will enable us to explore the comparative elegance of TM and RASP programs, and Chaitin's LISP programs, taking into account all layers of language realisation down to the formal semantics in a common notation.

We would also like to use GA techniques to look for paradigmatic programming patterns in high scoring RASP machines.

Acknowledgements. We would like to thank our colleague Phil Trinder for his constructive reflections on this research, and to the 4 anonymous reviewers whose comments have improved the quality of this paper.

Joe Davidson is pleased to acknowledge the support of the School of Mathematical and Computer Science, Heriot-Watt University for his PhD study.

References

1. Allen H. Brady. The busy beaver game and the meaning of life. In Rolf Herken, editor, *The universal Turing machine (2nd ed.)*, pages 237–254. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1995.
2. Cristian S. Calude and Michael Stay. Most programs stop quickly or never halt. *CoRR*, abs/cs/0610153, 2006.
3. Gregory J. Chaitin. *The Limits of Mathematics : A Course on Information Theory and the Limits of Formal Reasoning (Discrete Mathematics and Theoretical Computer Science)*. Springer, October 2002.
4. Calvin C. Elgot and Abraham Robinson. Random-access stored-program machines, an approach to programming languages. *J. ACM*, 11(4):365–399, 1964.
5. David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.
6. H. Marxen and J. Buntrock. Attacking Busy Beaver 5. *Bulletin of the European Association for Theoretical Computer Science*, 40, 1990.
7. Pascal Michel. The busy beaver competition: a historical survey, 2012.
8. T. Rado. On non-computable functions. *The Bell System Technical Journal*, 41(3):877–884, 1962.