Busy Beaver Machines and the Observant Otter Heuristic

James Harland

School of Computer Science and Information Technology
RMIT University
GPO Box 2476
Melbourne, 3001
Australia
james.harland@rmit.edu.au

Abstract

The busy beaver problem is to find the maximum number of non-zero characters that can be printed by an n-state Turing machine of a particular type. A critical step in the solution of this problem is to determine whether or not a given n-state Turing machine halts on a blank input. Given the enormous output sizes that can be produced by some small machines, it becomes critical to have appropriate methods for dealing with the exponential behaviour of both terminating and non-terminating machines. In this paper, we investigate a heuristic which can be used to greatly accelerate execution of this class of machines. This heuristic, which we call the *observant otter*, is based on the detection of patterns earlier in the execution trace. We describe our implementation of this method and report various experimental results based on it, including showing how it can be used to evaluate all known 'monster' machines, including some whose naive execution would take around $10^{36,534}$ steps.

1 Introduction

The *busy beaver* is a well-known example of a non-computable function. It was introduced by Rado [18] as a simple example of such functions, and is defined in terms of a particularly simple class of Turing machines [22]. This class of machines has a single tape, infinite in both directions, which is blank on input. Rado's original definition included only two tape symbols, 0 and 1 (or blank and non-blank, if preferred), although generalising this to an arbitrary number of symbols is straightforward. The machine is required to be deterministic, i.e. that for any state and input symbol there is exactly one transition. Each machine also includes a special state known as the *halt* state, from which there are no transitions. A machine is said to have n states when it has one halt state and n other states. The busy beaver function for n is then defined as the largest number of non-zero characters that can be printed by an n-state machine which halts. This function is often denoted as $\Sigma(n)$; in this paper we will use the more intuitive notation of $bb(n)$. The number of non-zeroes printed by the machine is known as its *productivity*.

This function can be shown to grow faster than any computable function[18]. Hence it is not only non-computable, it grows incredibly quickly. Accordingly, despite well over 40 year's worth of exponential increases in hardware capabilities in line with Moore's famous law[15], its value has only been established with certainty for $n \leq 4$ [13]. The values for $n = 1, 2, 3$ were established by Lin and Rado [9] in the 1960s and the value for $n = 4$ by Brady in the 1970s [3]. Larger values have proved more troublesome [11, 10, 8, 20], and the lower bounds for $n = 6$ are already spectacularly large [10]. There are some interesting analyses of the current champion machines for the $n = 5$ and $n = 6$ cases [17, 14], but due to the sheer size of the numbers involved, $bb(n)$ for $n \geq 7$ may never be known.

The current state of knowledge is given in the table below. We denote by $ff(n)$ (for *frantic frog*) the maximum number of state transitions performed by a terminating Turing machine with n states and 2 symbols.

n	$bb(n)$	$ff(n)$
1	1	1
2	4	6
3	6	21
4	13	107
5	≥ 4098	$\geq 47,176,870$
6	$\geq 3.51 * 10^{18267}$	$\geq 7.41 * 10^{36534}$

The numbers are also spectacular for machines with more than two symbols, as can be seen in the table below (where we denote the number of symbols by m).

n	m	$bb(n)$	$ff(n)$
2	3	9	38
2	4	≥ 2050	$\geq$ 3,932,964
3	3	$\geq$ 374,676,383	$\geq 1.12 * 10^{18}$
2	5	$\geq 1.7 * 10^{352}$	$\geq 1.9 * 10^{704}$
2	6	$\geq 1.9 * 10^{4933}$	$\geq 2.4 * 0^{9866}$
3	4	$\geq 3.7 * 0^{6518}$	$\geq 5.2 * 10^{13036}$
4	3	$\geq 1.383 * 10^{7036}$	$\geq 1.025 * 10^{14072}$

In order to determine what the maximum is for any given size, it is necessary to enumerate all possible Turing machines of that size, classify each of them as to whether they terminate on the blank input or not, and search all the terminating machines for the maximum. This makes it necessary to both find the final configuration for terminating machines (even though the number of steps needed might be very large), and to show that certain machines will never terminate.

One important aspect to note is that whilst the above numbers may appear large, the output patterns of the machines themselves are relatively simple. For example, the current 6-state 2-symbol champion, which prints out in the order of 10^{18628} characters terminates with the string below on the tape

$$1(110110)^X 11$$

where X is a number with 18,267 digits.

In this paper, we show how we can efficiently execute all of the machines mentioned above, and many others like them, by means of a simple heuristic that we call the *observant otter*.[1] This analyses the execution trace of the machine, and looks for some patterns in it. Once these are identified, this pattern can be used to predict resulting configurations of the machine, which saves a significant amount of execution time. In fact, for *leashed leviathans*[2] like the machines mentioned above, over 99% of the execution steps can be predicted in this way, which makes it possible to determine the final configurations of such machines in reasonable times.

It is also important to note that this process of predicting the machine's behaviour based on an execution trace is not just pivotal to showing that machines terminate, but also for showing that machines do not terminate. In [5] it is discussed how proving non-termination can be done by generating hypotheses from execution traces, and then showing that particular configuration patterns will recur infinitely. The process of showing that this will recur infinitely involves the same process of predicting execution patterns as is employed in the execution of terminating machines, and in fact the observant otter generalises all the techniques used in [5]. In this way, the observant otter can be thought of as a fundamental tool for the execution of machines of this kind.

We have implemented the observant otter heuristic, and added it to a simple execution engine based on the macro machines of Marxen and Buntrock [11]. We have tested it on 99 machines, including those mentioned above and various others.[3] We have found that the observant otter has been effective on all the 'monster' machines. We believe that this process has led to insights which will improve the design of the execution model for such machines.

[1] The alliterative nature of the term *busy beaver* has inspired many similar alliterative names like this one.

[2] This is another term inspired by the alliterative nature of the term 'busy beaver'. We will explain it in more detail later.

[3] All taken from Heiner Marxen's web page, `http://www.drb.insel.de/~heiner/BB/index.html`.

It should be noted that our philosophy is to do more than just find busy beaver machines and provide evidence of their optimality. Our aim is to study a number of interesting properties of given classes of machines. In order to do so, various tools are necessary, including methods to detect both termination and non-termination, and methods to search for all machines of a given size. This means that it is crucial to generate and store data systematically, such as a list of all machines of a given size, with or without classification. This will not only allow verification of empirical results by other researchers, but also for other properties of interest to be discovered and tested. For this reason, we believe that the process of discovery requires careful documentation, and it is this documentation that is the most important outcome of this work; results such as a specific value for the busy beaver, whilst interesting per se, are simply one aspect.

This paper is organised as follows. In Section 2 we give some basic definitions and background, and in Section 3 we describe the observant otter, including showing how it works on some examples. In Section 4 we describe our implementation and show our results, and in Section 5 we discuss our interpretation of them. Finally in Section 6 we present our conclusion and some areas of further work.

2 Definitions

2.1 Turing Machines

We use the following definition of a Turing machine [22].

Definition 1 *A Turing machine is a quadruple* $(Q \cup \{h\}, \Gamma, \delta, q_0)$ *where*

- h *is a distinguished state called a* halting state
- Γ *is the tape alphabet*
- δ *is a partial function from* $Q \times \Gamma$ *to* $Q \cup \{h\} \times \Gamma \times \{l, r\}$ *called the* transition function
- $q_0 \in Q$ *is a distinguished state called the* start state

Note that this is the so-called quintuple transition variation of Turing machines, in that a transition must specify for a given input state and input character, a new state, an output character and a direction for the tape in which to move. Hence a transition can be specified by a quintuple of the form

(State, Input, Output, Direction, NewState)

Some varieties of Turing machines allow only one of the latter two possibilities, i.e. either to write a new character on the tape or to move, and not both; for such machines, clearly only a tuple of 4 elements is required. Hence, given some notational convention for identifying the start state and halting state, a Turing machine can be characterised by the tuples which make up the definition of δ.

Many presentations of Turing machines assume that there is a single semi-infinite tape. In this paper, as is standard with busy beaver investigations, we assume that there is a single tape which is infinite in both directions.

Note also that there are no transitions for state h, and that as δ is a partial function, there is at most one transition for a given pair of a state and a character in the tape alphabet.

Definition 2 *We call a Turing machine M*

- ***normal*** *if there is exactly one tuple (S, I, O, D, N) of M for which $N = h$.*
- ***exhaustive*** *if δ is a total function from $Q \times \Gamma$ to $Q \cup \{h\} \times \Gamma \times \{l, r\}$, i.e. that $\forall q \in Q\ \forall \gamma \in \Gamma\ \exists q' \in Q \cup \{h\}, \gamma' \in \Gamma$ and $D \in \{l, r\}$ such that $\delta(q, \gamma) = \langle q', \gamma', D \rangle$.*

Note also that machines which are exhaustive but not normal are either guaranteed not to terminate (as there is no transition into the halting state and every combination of state and input symbol has a transition defined for it), or have multiple halting transitions, of which at most only one can ever be used, making the other halting transitions spurious. It is interesting to note that the Wolfram prize for finding minimal universal Turing machines given in 2007 used machines which are exhaustive but contained no halting transition.

We denote by an n-state Turing machine one in which $|Q| = n$. In other words, an n-state Turing machine has n "real" states and a halt state.

As we are interested in finding the maximum number of non-zeroes that can be printed by a terminating machine, we will only consider *maximising* machines, defined below.

Definition 3 *Let M be a Turing machine with n states and m symbols. M is a* ***maximising*** *machine if for every tuple (S, I, O, D, N) in M, whenever $N = h$, then O is 1.*

In a maximising machine, the final transition that is executed, i.e. the halting transition, can be guaranteed not to reduce the number of 1's (or more generally, non-blank symbols) on the tape. This seems an entirely desirable property in a search for busy beavers.

Henceforth we will restrict our attention to machines which are normal, exhaustive and maximising. In such a machine, the (unique) halting transition (S, I, O, D, N) always has O as 1, N as h, and we may arbitrarily assign D to be r.[4] We will denote the transitions of a machine in this way, so that we in general refer to the first item in the tuple (S, I, O, D, N) as S, the second as I etc.

We will find the following notion of *dimension* useful.

Definition 4 *Let M be a Turing machine with n states and m symbols (where $n, m \geq 2$). Then we say M has dimension $n \times m$.*

[4]This means that each machine we consider has a *sinister sibling* in which the direction of every transition is swapped, which will behave identically except for the direction of its movements.

Note that as we are interested in machines which are normal, exhaustive and maximising, we know that these machines will have exactly $n \times m$ transitions, one for each combination of state and tape symbol, and exactly one of these will be the halting transition.

The reason that this is of interest is that for every n-state m-symbol machine there is an m-state n-symbol machine and vice-versa. To see this, consider the mapping between transitions defined below.

Definition 5 *Let M be a Turing machine with n states and m symbols and let (S, I, O, D, N) be a transition T in M. We define the dual transition $dual(T)$ of T as follows:*

- *If $N = h$, then $dual(T) = (I, S, O, D, N)$*
- *otherwise, $dual(T) = (I, S, N, D, O)$.*

We define the dual machine $dual(M)$ of M as $\{dual(T) | T \in M\}$.

For example, the duals of the transitions $(a, 0, 1, r, b)$, $(b, 1, 1, l, c)$ and $(c, 0, 1, r, h)$ are $(0, a, b, r, 1)$, $(1, b, c, l, 1)$ and $(0, c, 1, r, h)$ respectively[5].

Hence for a machine M with n states and m symbols, $dual(M)$ has m states and n symbols. In particular, for every n-state 2-symbol machine M, there is a unique 2-state n-symbol machine M', and vice-versa. This shows that there are as many n-state m-symbol machines as there are m-state n-symbol machines, and hence searching through all such machines has the same complexity in each case.

While we do not discuss searching through lists of machines in this paper, it suggests that a systematic search of machines that have the same dimension is simpler than doing these independently. This seems particularly appropriate for machines of dimension 10 (i.e. machines with 5 states and 2 symbols, and 2 state and 5 symbols) and 12 (6x2, 4x3, 3x4, and 2x6 states and symbols respectively).

2.2 Macro Machines

As is well known, a configuration of a Turing machine is the current state of execution, containing the current tape contents, the current machine state and the position of the tape head. We will use $111\{b\}011$ to denote a configuration in which the Turing machine is in state b with the string 111011 on the tape and the tape head pointing at the 0.

Macro machines [11] is a well-known technique for optimising the execution of Turing machines. Essentially, the tape is considered to consist of blocks of characters of a given fixed length k, and the transitions for the machine are determined according to this structure. For example, if $k = 3$, then when the machine encounters say the input string 001 in state c, then it will execute naively until either the tape head moves outside these

[5]It may be more intuitive to re-label the dual transitions as $(a, 0, 1, r, b)$, $(b, 1, 2, l, b)$ and $(a, 2, 1, r, h)$.

three specific symbols, or it will eventually repeat an earlier configuration. In the latter case, we may classify the computation as non-terminating. Otherwise, we note the state in which the machine exits from the area containing just the three given symbols, and the symbols that now occupy these places, and consider this a new 'macro' transition for the machine. The only additional information that is necessary is to keep track of the direction from which the tape head entered the current string, and the direction that it exited from the string. This means that for a given string, there are two transitions: one for when it enters from the left, and another for when it enters from the right. This also means that we need to keep track of not just the current (macro) symbol, but the current orientation of the tape head (left or right). Extending the notation for configurations above, we will use $111\{b\}001\{l\}110$ to denote a configuration in which the machine is in state b with the string 111001110 on the tape, and the tape head pointing to the left hand end of the 001 string.

For the above example, if the initial symbols 001 are converted to 111 and the tape head leaves these three symbols in the state d, then we may consider this a macro transition from state b with input 001 to new state d with output 111. The details are described in Marxen and Buntrock's seminal paper [11], and there are various improvements described on Marxen's Busy Beaver web site.

A key aspect of macro machines is that having a fixed string length makes it easy to represent repetitive sequences of strings such as 001001001001 as $(001)^4$. This, together with the acceleration technique below, is what makes macro machines significantly more efficient than naive execution.

To see how this works, consider a macro machine with $k = 3$, and an execution trace in which the current state is b, the current string under the tape head is $(001)^{12}$, and the input direction is left (so that the tape head is pointing at the first 0 in the string $(001)^{12}$). If the output state is also b, the output string is 111 and the output direction is right (so that the tape head exits the string at the right-hand end), then it is clear that a machine configuration of the form

$$X\{b\}(001)^{12}\{l\}Y$$

will become

$$X(111)^{12}\{b\}Y_1\{l\}Y_2$$

where Y_1 is next element of Y (and Y_2 is the rest of Y).

If the output state is not b (i.e. it is different to the input state) or the output direction is not to the right, then we "decouple" one copy of 001 from the longer string $(001)^{12}$ and proceed. In the above example, if the output string is 110, the output state is c and the output direction is left, then the configuration

$$X\{b\}(001)^{12}\{l\}Y$$

will become

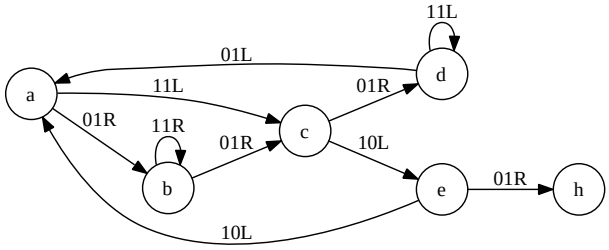

Figure 1: Current 5x2 champion

$$X_1\{c\}X_2\{r\}110(001)^{11}Y$$

where X_2 is the last element of X (and X_1 is the rest of X).

3 The Observant Otter

Consider the Turing machine in Figure 1. This is the 5x2 champion, which terminates after 47,176,870 steps with 4,098 ones on the tape.

During the execution of this machine, the following configurations occur.

Step	**Configuration**
12393	$001111\{b\}(001)^{66}100$
12480	$001(111)^{6}\{b\}(001)^{63}100$
12657	$001(111)^{11}\{b\}(001)^{60}100$

Clearly there is a pattern here, which is that a configuration of the form

$$001(111)^{X}\{b\}(001)^{Y}100$$

becomes

$$001(111)^{X+5}\{b\}(001)^{Y-3}100$$

provided that $Y > 3$. Hence the role of the observant otter is to notice this pattern, and apply it. This is done by looking through the execution trace for matching 'shapes', and then checking to see if there is a pattern of this form. If we detect three instances of a pattern like this, then we use the three instances to predict the final configuration in this sequence. In the above example, we can see that eventually the machine will reach the configuration below.

$$001(111)^{106}\{b\}(001)^{3}100$$

Note that we stop with the last non-zero value of Y in the pattern. This is because there are some instances in which going down to zero causes incorrect behaviour. Whilst this is a conservative choice, it does not appear to be a very costly one. It is an item of ongoing research to determine criteria for when it is 'safe' to go to the zero case and when it is not.

Similar patterns, with increasingly large numbers, recur through the execution trace. For example, the following three configurations occur in the 5x2 champion.

Step	Configuration
281384	$001111\{b\}(001)^{316}100$
281471	$001(111)^{6}\{b\}(001)^{313}100$
281648	$001(111)^{11}\{b\}(001)^{310}100$

We can use this sequence to predict a configuration of

$$001(111)^{526}\{b\}001100$$

This means that we add an extra execution process to the two above for macro machines. This is to search through the execution trace for patterns that match the current one, and if one is found, to calculate the resulting configuration. Otherwise, if no such pattern applies, then we proceed as in the two cases above. In some ways, the observant otter may be thought of as a natural counterpart to the process of 'decoupling', i.e. the second step of the macro machine execution described above. This is because this process essentially changes a string of the form S^n to one of the form SS^{n-1}, and hence generally decrements n. As in the above example for the 5x2 champion, if the process repeated in a predictable pattern, then the observant otter will act to 'complete' this process. Put another way, it is this process of 'decompressing' S^n to SS^{n-1} (which we shall refer to as the *stretching stork* operation) that produces potential configurations for the observant otter to detect.

In general, then, if the current configuration is

$$X_1^{i_1} \dots X_n^{i_n}\{S\}Y^j\{D\}Z_1^{k_1} \dots Z_m^{k_m}$$

where the X_i, Y and the Z_k are strings of characters of a fixed length, S is the current state and D the current direction, the observant otter searches through the execution trace of the machine for two earlier occurrences of configurations of the same 'shape' and for which there is a *regression*, i.e., two configurations of the form

$$X_1^{o_1} \dots X_n^{o_n}\{S\}Y^p\{D\}Z_1^{q_1} \dots Z_m^{q_m}$$

and

$$X_1^{r_1} \dots X_n^{r_n}\{S\}Y^s\{D\}Z_1^{t_1} \dots Z_m^{t_m}$$

such that at least one of the following three conditions holds:

- for some $1 \leq u \leq n$ and some $a > 0$ we have $o_u = i_u - a$ and $r_u = i_u - 2a$
- for some $a > 0$ we have $p = j - a$ and $s = j - 2a$
- for some $1 \leq v \leq m$ and some $a > 0$ we have $q_v = k_v - a$ and $t_v = k_v - 2a$

In other words, we have the same sequence of 'macro' characters $X_1 \dots X_n\{S\}Y\{D\}Z_1 \dots Z_m$ in all three configurations, and the index of at least one of these configurations is regressing as execution proceeds.

One issue with this approach is that searching through the history of the execution trace has quadratic complexity, and hence has the potential to significantly slow down execution. We address this problem by setting an upper limit on the number of previous steps that are stored. In the results reported below, a 'window' of 150 steps was sufficient for all cases tested, including those of the highest productivity. This shows that the patterns detected by the otter are remarkably local, in that it is sufficient to look at most 150 steps in the past to find patterns. Increasing this value may find more patterns, but it comes with a noticeable performance penalty. Whilst we are interested in ensuring that the computation is efficient enough to be feasible, finding a maximally efficient method is outside the scope of this paper. If performance were to become a critical issue, then standard techniques such as hashing or balanced binary trees may be appropriate for performing this search.

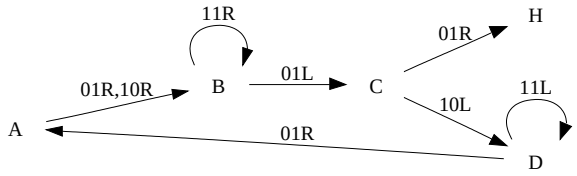

Figure 2: Non-terminating machine

A further issue is to calculate the number of steps between the current configuration and the predicted one. When the difference between the three occurrences of the pattern is constant, this is straightforward. However, there are cases in which the difference increases, and so some further care is required. This is not an essentially difficult problem, and the details are not relevant to the scope of this paper, and so they are omitted.

One point that should be noted is that the observant otter heuristic is also crucial to the success of non-termination proofs. As discussed in [5], a key aspect of many non-termination proofs is to determine a pattern from the execution trace, and then show that this pattern will recur infinitely often. This is usually done by showing that a pattern is growing in size, such as observing the sequence of configurations below,

$$11\{c\}11$$
$$(11)^2\{c\}11$$
$$(11)^3\{c\}11$$

and inferring from them that a configuration of the form $(11)^n\{c\}11$ will eventually result in one of the form $(11)^{n+1}\{c\}11$ for any $n \geq 1$. This process is certainly similar to the observant otter, but seems a little distinct from it (not the least because the count is always increasing here). However, the observant otter patterns often crop up when attempting to show that configurations such as $(11)^n\{c\}11$ eventually result in the configuration $(11)^{n+1}\{c\}11$. For example, consider the machine in Figure 2.

An analysis of the execution trace shows that configurations of the form $(11)^n\{c\}11$ occur. Hence we attempt to show that this configuration

will eventually lead to $(11)^{n+1}\{c\}11$. In doing so, a necessary intermediate step is to go from the configuration $1^n01^k\{c\}1^m$ to $1^{n+1}01^{k-2}\{c\}1^{m+1}$, which is precisely an observant otter step. Hence, we need the observant otter to determine that the machine will eventually reach the configuration $1^{n+l}01^j\{c\}1^{m+l}$ where $l = k \;//\; 2$ and $j = k \; mod \; 2$, and from there to $(11)^{n+2}\{c\}11$, at which point we can conclude that we have successfully shown that this machine does not terminate. A similar method has been described by Hertel [6].

Hence the observant otter seems to be fundamental to the execution of these machines, whether terminating or not.

4 Implementation and Results

We have developed an implementation of both a naive interpreter and one based on macro machines. We then added the observant otter heuristic to the macro machine implementation. This implementation is deliberately simple; there is probably a large amount of scope for improvement, particularly in performance times. Our aim with this implementation was not to break any 'land speed' records, so to speak, but to show that a simple approach together with an appropriate heuristic (in this case, the observant otter) is sufficient to evaluate machines with the largest known productivities to date. Our implementation is around 2,000 lines of SWI-Prolog [21].

The table in Figures 3 and 4 shows all 99 machines that we have evaluated. This list, together with all the code developed, is also available at `http://www.cs.rmit.edu.au/~jah/busybeaver`. We have included machines of various dimensions, including 3x3, 6x2, 3x4, 4x3 and 2x6. The results were obtained on a desktop with an Intel i7 with 3.4GHz processors and 3.2 GB of RAM running Windows XP.

We have included the time taken to execute (in seconds). This is useful to know, but the most important point to note is that these can be achieved on commodity hardware with the longest time taking around 45 minutes, and all but four machines taking less than 10 minutes each. In other words, the entire sequence of tests can be done within a few hours at most.

In the table in Figures 3 and 4, an entry of the form **(N)** denotes a (known) number with **N** digits. We have adopted the convention of only explicitly writing out numbers of up to 15 digits in length. The **No.** column is our identifier for each machine. The **Ones** column is the number of non-zero characters on the tape in the final configuration. The **Hops** column is the number of steps needed to reach this configuration (if executed naively). The **Steps** column is the number of steps executed in our interpreter. The **Otters** column is the number of otter patterns detected during execution. The **Otter %** column is the percentage of the execution steps that were predicted by the observant otter heuristic.

5 Discussion

As can be seen from the table in Figures 3 and 4, for the machines of very high productivity, the observant otter has proved to be very effective, with at least 99% of the execution steps being predicted by the observant otter. It also shows some potentially surprising results, in that the machines with the largest productivities weren't necessarily the ones which required the largest number of observant otter occurrences. For example, machine 19, the 5x2 champion, required 11 occurrences of the observant otter, but machine 21 required 74. This suggests that the number of such occurrences is a more intuitive measure of the difficulty of execution than the sheer size of the final configuration or the number of steps required to compute it.

It is also worth noting that the ratio of **Hops** to **Steps** in the above table is very high, especially for the machines of largest productivity. Clearly the **Steps** value increases as the productivity does, but at an exponentially lower rate than **Hops**.

A key decision to make in the implementation is to determine how large the 'history window' should be. After a little experimentation, it was found that 150 works for all the above cases, but making it smaller tended to make the performance deteriorate significantly on some of the larger machines. This shows that the otter patterns are remarkably local, in that despite the large number of steps involved, only a small fraction of the most recent ones are needed to detect patterns (or at least, the patterns detected in this way are sufficient to efficiently execute the machine). As mentioned above, it is possible to improve the efficiency of this procedure by using data structures such as a hash table rather than the simple list implementation used here.

A further potential improvement to this process is to store the otter patterns as they are found, rather than computing them 'afresh' each time. For example, the 11 otter occurrences in machine 19 (the 5x2 champion) are basically different instances of two particular otter patterns. However, storing and applying the correct pattern is not altogether straightforward, particularly as the number of steps in each pattern needs to be calculated differently. This is an item of future work.

A related aspect is that of finding the 'earliest possible' otter pattern. To return to the 5x2 champion yet again, the first otter detected by our process is the one below.

Step	**Configuration**
448	$001111\{b\}(001)^{12}\{l\}100$
535	$001(111)^{6}\{b\}(001)^{9}\{l\}100$
712	$001(111)^{11}\{b\}(001)^{6}\{l\}100$

However, there are some earlier patterns that the macro machine structure makes it difficult to detect. This is because the choice of $k = 3$ is generally best for this machine (due to strings such as $(001)^{522}$ occurring in the execution trace), but this has the effect of representing a string of 18 1's as $(111)^6$, rather than the more compact and intuitive form 1^{18}. In order to find the earliest possible

No.	**Dim.**	**Ones**	**Hops**	**Steps**	**Otters**	**Otter %**	**Time** (seconds)
1	2x4	84	6445	284	2	57	0.094000
2	2x4	90	7195	287	2	64	0.078000
3	2x4	90	7195	287	2	64	0.078000
4	2x4	2050	3932964	737	8	99	0.203000
5	2x4	2050	3932964	737	8	99	0.204000
6	3x3	31	2315619	824187	0	0	18.093000
7	3x3	5600	29403894	1050	16	> 99	0.266000
8	3x3	13939	92495804	441	7	> 99	0.094000
9	3x3	2050	3932964	737	8	99	0.203000
10	3x3	36080	310196798	828	17	> 99	0.172000
11	3x3	13802	99981302	641	7	> 99	0.172000
12	3x3	9486	84327815	974	11	> 99	0.265000
13	3x3	11501	56122828	651	10	> 99	0.204000
14	3x3	9486	84327831	982	11	> 99	0.265000
15	3x3	17338	127529109	787	17	> 99	0.219000
16	3x3	12077	69440986	421	7	> 99	0.093000
17	3x3	95524079	(16 digits)	1149	17	> 99	0.282000
18	3x3	374676383	(18 digits)	4520	60	> 99	1.063000
19	5x2	4098	47176870	526	11	> 99	0.172000
20	5x2	4098	11798826	510	10	> 99	0.109000
21	5x2	4097	23554764	1456	74	> 99	0.593000
22	5x2	4097	11798796	498	10	> 99	0.109000
23	5x2	4096	11804910	550	9	99	0.125000
24	5x2	4096	11804896	544	9	99	0.141000
25	5x2	1915	2133492	393	4	97	0.078000
26	5x2	1471	2358064	931	41	98	0.234000
27	5x2	501	134467	290	6	90	0.047000
28	2x5	90604	8619024596	1561	9	> 99	0.437000
29	2x5	64665	4561535055	1283	10	> 99	0.328000
30	2x5	97104	7543673517	1362	17	> 99	0.313000
31	2x5	458357	233431192481	3640	84	> 99	0.875000
32	2x5	668420	469121946086	1879	12	> 99	0.531000
33	2x5	1957771	912594733606	852	12	> 99	0.187000
34	2x5	1137477	924180005181	1559	47	> 99	0.422000
35	2x5	90604	8619024596	970	10	> 99	0.296000
36	2x5	4848239	14103258269249	916	32	> 99	0.329000
37	2x5	143	26375397569930	??	0	0	??
38	2x5	4099	15754273	958	10	> 99	0.250000
39	2x5	3685	16268767	765	13	> 99	0.219000
40	2x5	11120	148304214	947	15	> 99	0.266000
41	2x5	36543045	417310842648366	1685	27	> 99	0.421000
42	2x5	114668733	(16 digits)	961	18	> 99	0.250000
43	2x5	398005342	(17 digits)	648	19	> 99	0.157000
44	2x5	620906587	(17 digits)	919	20	> 99	0.235000
45	2x5	1194050967	(18 digits)	815	20	> 99	0.218000
46	2x5	1194050967	(18 digits)	823	20	> 99	0.219000
47	2x5	172312766455	(22 digits)	862	26	> 99	0.235000
48	2x5	(31 digits)	(62 digits)	8587	168	> 99	2.640000
49	2x5	(106 digits)	(212 digits)	24367	593	> 99	7.047000
50	2x5	(106 digits)	(212 digits)	26169	593	> 99	7.375000
51	2x5	(353 digits)	(705 digits)	81738	1993	> 99	25.532000
52	6x2	136612	13122572797	1498	15	> 99	0.344000
53	6x2	95524079	(16 digits)	1514	16	> 99	0.328000
54	6x2	17485734	95547257425490	1929	25	> 99	0.438000
55	6x2	36109969	(16 digits)	2256	83	> 99	0.546000
56	6x2	36109970	758650111948072	1000	14	> 99	0.203000
57	6x2	5234513991	(20 digits)	5817	215	> 99	1.547000
58	6x2	33299939444	(21 digits)	1734	21	> 99	0.390000
59	6x2	11974457230330	(26 digits)	2155	27	> 99	0.485000
60	6x2	40740206640846	(28 digits)	12634	408	> 99	3.343000

Figure 3: Observant Otter results (part 1)

No.	Dim.	Ones	Hops	Steps	Otters	Otter %	Time (seconds)
61	6x2	(20 digits)	(38 digits)	4197	153	> 99	1.094000
62	6x2	(22 digits)	(43 digits)	33228	1075	> 99	8.547000
63	6x2	(48 digits)	(96 digits)	20493	643	> 99	5.312000
64	6x2	(48 digits)	(96 digits)	19203	641	> 99	4.969000
65	6x2	(50 digits)	(99 digits)	28242	380	> 99	6.922000
66	6x2	(61 digits)	(120 digits)	12321	332	> 99	2.937000
67	6x2	(50 digits)	(100 digits)	29547	389	> 99	7.016000
68	6x2	(463 digits)	(926 digits)	265475	12330	> 99	108.500000
69	6x2	(866 digits)	(1731 digits)	244643	14334	> 99	87.688000
70	6x2	(882 digits)	(1763 digits)	288760	2211	> 99	79.890000
71	6x2	(1440 digits)	(2880 digits)	406270	8167	> 99	128.016000
72	6x2	(10567 digits)	(21133 digits)	1927565	99697	> 99	876.375000
73	6x2	(18268 digits)	(36535 digits)	6195809	30345	> 99	2866.063000
74	3x4	17323	262759288	379583	17778	47	112.172000
75	3x4	(27 digits)	(53 digits)	9187	358	> 99	3.469000
76	3x4	(141 digits)	(282 digits)	50694	801	> 99	14.375000
77	3x4	(435 digits)	(869 digits)	103962	3493	> 99	36.281000
78	3x4	(629 digits)	(1257 digits)	168263	1488	> 99	54.890000
79	3x4	(1302 digits)	(2602 digits)	96341	2163	> 99	28.813000
80	3x4	(2356 digits)	(4711 digits)	3178530	107045	> 99	926.156000
81	3x4	(2373 digits)	(4745 digits)	944360	13465	> 99	240.828000
82	3x4	(6519 digits)	(13037 digits)	731258	13667	> 99	223.125000
83	4x3	15008	250096775	1594	27	> 99	0.547000
84	4x3	(714 digits)	(1427 digits)	375816	20051	> 99	105.797000
85	4x3	(810 digits)	(1619 digits)	924926	34272	> 99	265.188000
86	4x3	(987 digits)	(1974 digits)	497189	7887	> 99	140.687000
87	4x3	(3861 digits)	(7722 digits)	1759075	44810	> 99	513.157000
88	4x3	(4562 digits)	(9123 digits)	1511192	25943	> 99	467.562000
89	4x3	(4932 digits)	(9864 digits)	1524523	57368	> 99	514.047000
90	4x3	(6035 digits)	(12069 digits)	1319358	34262	> 99	354.391000
91	4x3	(7037 digits)	(14073 digits)	1695880	25255	> 99	492.453000
92	4x3	(7037 digits)	(14073 digits)	1695879	25255	> 99	522.531000
93	2x6	10574	94842383	948	11	> 99	0.531000
94	2x6	10249	98364599	712	13	> 99	0.250000
95	2x6	15828	493600387	1263	15	> 99	0.422000
96	2x6	(28 digits)	(55 digits)	5059	149	> 99	1.844000
97	2x6	(822 digits)	(1644 digits)	139554	2733	> 99	47.203000
98	2x6	(4932 digits)	(9864 digits)	1083249	49142	> 99	323.719000
99	2x6	(4934 digits)	(9867 digits)	770686	16394	> 99	245.657000

Figure 4: Observant Otter results (part 2)

occurrences of patterns, it thus seems necessary to develop an adaptive compression of configurations, so that we can represent the above sequence of configurations as below.

Step	Configuration
448	$1^4\{b\}(001)^{12}\{l\}1$
535	$1^{19}\{b\}(001)^{9}\{l\}1$
712	$1^{34}\{b\}(001)^{6}\{l\}1$

This finer granularity will enable 'earlier' otter occurrences to be detected. This is also an item of future work.

It also seems worth remarking that often the size of the otter steps predicted seems to increase exponentially as the computation goes on. For example, for the 5x2 champion, in a total of 48 million steps, there are a total of 11 otter applications, with the number of predicted steps for each one as in the table below.

Otter number	Steps predicted
1	267
2	2,235
3	6,840
4	20,463
5	62,895
6	176,040
7	500,271
8	1,387,287
9	3,878,739
10	10,830,672
11	30,144,672

Note that the final otter step alone predicts 30,144,672 steps, or around 60% of the total number of steps. Clearly there is an exponential growth in the number of predicted steps here, yet the overall computation still terminates. This property of terminating exponential growth is the inspiration for the name *leashed leviathans*; there

is an exponential computation happening here, but it reaches a limit at some point. This behaviour is reminiscent of the famous $3n + 1$ sequence. For now, we note that the observant otter may be thought of as a means of coping with this behaviour.

It is also worth noting that for the machine of very high productivity, the number of Hops taken seems to be almost always the square of the number of non-zeroes in the final configuration, and that this property seems remarkably consistent, despite the very large numbers involved. There are more sophisticated analyses of busy beaver champions than we have given here (such as those by Michel [13]); here we remark that it seems intriguing to contemplate the connection between the patterns found by the observant otter and this property, particularly as there seems to be a correlation between the number of otter occurrences and adherence to this property.

As mentioned above, in the above results we always applied the otter in such a way that the predicted configuration was always 'one less' than the maximum possible. In other words, given a pattern such as X^nY^m leading to $X^{n+2}Y^{m-1}$, our calculation always predicted the final state as $X^{n+2(m-1)}Y$ rather than the potentially more efficient choice of X^{n+2m}. Whilst this approach worked in many cases, there are some in which this approach leads to erroneous results. Determining precise criteria under which one can 'safely' use the zero case is also an item of future work. This seems particularly important for the patterns occurring in a non-termination context; when used on terminating machines, our practice is technically less efficient, but only results in a very small increase in the time taken to execute the machine.

It should also be noted that the observant otter is not a panacea; in fact there were two machines where the otter was of no use at all (numbers 6 and 37), and one where its use was significantly less significant than many others (number 74). The first two of these are remarkable for their non-adherence to the above mentioned 'square law', in that these machines produce 31 non-zeroes in 2,315,619 steps and 143 non-zeroes in 26,375,397,569,930 steps respectively. These machines do not produce configurations with large exponents (e.g. containing strings like $(001)^{123194244242}$), and in fact do not have any patterns recognised by the observant otter. Machine 74 is a hybrid case, in that almost half the steps were predicted by the observant otter, but the other half were not. These three machines are examples of what we call *wandering wastrels*; these are machines that have very low productivities compared to the number of steps needed to come to the final configuration. No doubt there are other varieties of extreme machine behaviour 'out there', so that when dealing with a systematic analysis of a large number of machines, it will be necessary to have a number of different techniques in addition to the observant otter. However, we believe that the results of this paper have shown that the observant otter is a crucial component of the techniques required.

6 Conclusions and Further Work

We have seen how the observant otter heuristic, even when implemented in a simple manner, makes it possible to efficiently evaluate machines of very high productivity. We have also seen how this heuristic seems to be a natural extension of the implementation of macro machines, and is also appropriate for evaluating machines in general, terminating or otherwise. It still remains an open issue as to how we may determine the earliest possible application of the observant otter. As mentioned above, this is related to the design of macro machines, and to solve this problem will require a more flexible machine architecture. In particular, rather than pick some values in advance, and see which ones work best (which is the current approach), it would be better to run a (naive) execution for say 1,000 steps, examine the trace for patterns, and to determine the most appropriate representation. In addition, rather than use a fixed block size, it would be better to adapt the representation dynamically as the machine executes. Determining the right way to do this is an item of future work. It should be noted that this adaptive compression may also be helpful for the wandering wastrel machines.

Further work is also needed in order to determine precise conditions under which the predicted configuration can be safely calculated with a value of 0 for the final value of the decrementing variable. As mentioned above, for terminating machines, there is not a great loss of efficiency in being conservative and always avoiding this case. For non-terminating machines, it may be critical to be more precise.

Another avenue of further work is to be able to store patterns, and hence be able to reuse them without having to 'rediscover' patterns. We refer to this process as the *ossified ocelot* heuristic, which will be the subject of a future paper. Whilst it seems an intuitively natural thing to do, a key problem is to know exactly what to store, i.e. which parts of a given configuration are strictly necessary to the pattern, and which are irrelevant. It seems also that the number of 'different' otter patterns necessary for the evaluation of the machine is an even more intuitive measure of the complexity of the machine than the number of otter applications used during evaluation.

Acknowledgements

The author would like to thank Austin Wood, Alex Holkner, Andy Kitchen, and the referees of this paper for helpful comments and discussions related to this work.

References

[1] George Boolos and Richard Jeffrey, *Computability and Logic*, 2nd edition, Cambridge University Press, 1980.

[2] Allen Brady, Busy Beaver Problem of Tibor Rado, `http://www.cse.unr.edu/~al/BusyBeaver.html`

[3] Allen Brady, *The Determination of the value of Rado's noncomputable function* $\Sigma(k)$ *for four-state Turing machines*, Mathematics of Computation 40(162): 647-665, 1983.

[4] James Harland, *The Busy Beaver, the Placid Platypus and Other Crazy Creatures*, Proceedings of Computing: the Australasian Theory Symposium (CATS'06), Hobart, January, 2006. Published as Volume 51 - Theory of Computation 2006 of the ACS Conferences in Research and Practice in Information Technology (CRPIT) series.

[5] James Harland, *Analysis of Busy Beaver Machines via Induction Proofs*, Proceedings of Computing: the Australasian Theory Symposium (CATS'07) 71-78, Ballarat, January, 2007. Published as Volume 65 - Theory of Computation 2007 of the ACS Conferences in Research and Practice in Information Technology (CRPIT) series.

[6] Joachim Hertel, *Computing the Uncomputable Rado Sigma Function: An Automated Symbolic Induction Prover for Nonhalting Turing Machines*, Mathematica Journal 11:2, 2009.

[7] Alex Holkner, *Acceleration Techniques for Busy Beaver Candidates*, in Gad Abraham and Benjamin I.P. Rubenstein (eds.), *Proceedings of the Second Australian Undergraduate Students' Computing Conference* 75-80, December, 2004. ISBN 0-975-71730-8. Available from `http://www.cs.berkeley.edu/~benr/publications/auscc04`.

[8] Owen Kellett, *A Multi-Faceted Attack on the Busy Beaver Problem*, Master's Thesis, Rensselaer Polytechnic Institute, August, 2005.

[9] Shen Lin and Tibor Rado, *Computer Studies of Turing Machine Problems*, Journal of the Association for Computing Machinery 12(2):196-212, April, 1965.

[10] Heiner Marxen, Busy Beaver web page, `http://www.drb.insel.de/~heiner/BB/index.html`.

[11] Heiner Marxen and Jürgen Buntrock, *Attacking the Busy Beaver 5*, Bulletin of the EATCS 40:247-251, February 1990.

[12] Pascal Michel, *Busy beaver competition and Collatz-like problems*, Archive for Mathematical Logic 32 (5) 1993, 351-367.

[13] Pascal Michel, *The Busy Beaver Competition: a historical survey*, ArXiv e-prints 0906.3749, June, 2009. Available from `http://adsabs.harvard.edu/abs/2009arXiv0906.3749M`.

[14] Pascal Michel, Behavior of Busy Beavers, `http://www.logique.jussieu.fr/~michel/beh.html`.

[15] Gordon Moore, *Cramming More Components onto Integrated Circuits*, Electronics 38(8), April 19, 1965.

[16] Doran Moppert, *Busy Beavers in the Lambda Calculus*, M.App.Sci(IT) Thesis, School of Computer Science and IT, RMIT University, Melbourne, July 2008.

[17] R. Munafo, Large Numbers – Notes, `http://home.earthlink.net/~mrob/pub/math/ln-notes1.html`.

[18] Tibor Rado, *On non-computable functions*, Bell System Technical Journal 41: 877-884, 1962.

[19] Kyle Ross, *Use of Optimisation Techniques in Determining Values for the Quadruplorum Variants of Rado's Busy Beaver Function*, Masters thesis, Rensselaer Polytechnic Institute, 2003.

[20] Georgi Georgiev, Busy Beaver Prover, `http://skelet.ludost.net/bb/index.html`.

[21] SWI Prolog WWW Site, `http://www.swi-prolog.org`.

[22] Thomas Sudkamp, *Languages and Machines: An Introduction to the Theory of Computer Science*, (3rd ed.), Addison Wesley, 2005.

[23] Austin Wood, *Techniques for Evaluating Busy Beaver Machines*, Honours thesis, School of Computer Science and IT, RMIT University, Melbourne, October, 2008.