HALTING PREDICTION ON BUSY BEAVER TYPE TURING MACHINES BASED ON INFORMATION ENTROPY

(BUSY BEAVER TÜRÜ TURING MAKİNALARINDA BİLGİ ENTROPİSİNE DAYALI SONLANMA ÖNGÖRÜSÜ)

by

Hakan AYRAL, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

in the

INSTITUTE OF SCIENCE AND ENGINEERING

of

GALATASARAY UNIVERSITY

May 2008

HALTING PREDICTION ON BUSY BEAVER TYPE TURING MACHINES BASED ON INFORMATION ENTROPY

(BUSY BEAVER TÜRÜ TURING MAKİNALARINDA BİLGİ ENTROPİSİNE DAYALI SONLANMA ÖNGÖRÜSÜ)

by

Hakan AYRAL, B.S.

Thesis

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Date of Submission : 16 May 2008

Date of Defense Examination : 2 June 2008

Acknowledgements

I would like to express my sincere gratitude to my supervisor Assoc. Prof. A. Muhammed Uludağ, not only for his academic support he provided during the preparation of this thesis but also on many other fields of mathematics. He scientifically inspired me not only during the course of preparation of this thesis, but throughout all years of my graduate studies, and extended my perspective on many scientific subjects.

I also would like to express my deepest appreciation and gratitude to my family for their infinite support in every possible way.

Hakan Ayral

15 May 2008

Table of Contents

List of Figures

List of Tables

List of Code Listings

Abstract

In this thesis we mainly propose a new asymptotically complete halting predictor for Turing Machines of Busy Beaver type which is defined by T.Rado in 1962. Also we propose an efficiency measure to benchmark different halting predictors and finally we propose a topological representation for space of valid Turing Machines as a metric space with a Manhattan like distance metric allowing us to define a neighborhood between Turing Machines.

Our predictor uses the ratio of tape space explored to cycles taken during a point in simulation history as a measure of information density for that moment of simulation, which allows us to predict the unconstructability of a counting process in terms of number of cycles occurred till that point. We show that a halting Busy Beaver Turing Machine has to have the ability to keep track of its temporal position at each point of its simulation; and we construct a non-halting predictor using mentioned information density measure to show inability to track temporal position.

Our method predicts non-halting of Busy Beaver Turing Machines by incurring negligible computational overhead to the regular simulation, while obtaining results very early on simulation; even for complicated machine configurations where conventional automated non-halting proving is ineffective or unfeasible.

Résumé

Dans cette thèse nous proposons principalement une nouvelle prédicteur d’arrêt asymptotiquement complet pour machines de Turing de type Busy Beaver qui est défini par T. Rado en 1962. Aussi nous proposons une mesure d'efficacité pour différents prédicteurs d’arrêt et enfin, nous proposons une représentation topologique pour l'espace des machines Turing comme un espace métrique avec une métrique de distance Manhattan.

Notre prédicteur utilise le ratio de la bande exploré aux cycles prises au cours d'un point de simulation, comme une mesure de densité d'information pour ce moment de la simulation; ce qui nous permet de prédire impossibilité de construire un processus compteur en termes de nombre de cycles. Nous montrons que une Busy Beaver machine Turing doit avoir la capacité de garder la trace de sa position temporelle à chaque point de sa simulation, et nous construisons notre prédicteur de non-arrêt utilisant densité d’information pour démontrer l'impossibilité de suivre la position temporelle.

Özet

Bu tezde Busy Beaver türü Turing Makinaları için asimptotik olarak tam bir sonlanma öngörüsü öneriyoruz. Aynı zamanda farklı sonlanma öngörüsü sistemlerini karşılaştırabilmek için bir verimlilik ölçümü sunuyoruz ve son olarak geçerli Turing Makinası tanımlarını topolojik anlamda temsil edebilecek Manhattan uzaklık fonksyonu türevi bir uzaklık fonksyonuna sahip metrik bir uzay ve bu uzaydaki komşulukları tanımlıyoruz.

Sonlanma öngörüsü sistemimiz, benzetim geçmişindeki bir noktada Turing makina teybinin ziyaret edilmiş kısım uzunluğunun, teyp başlığının kaymalarına oranını, simülasyon geçmişinde o nokta için bilgi yoğunluğunun bir ölçümü olarak kullanarak simülasyon geçmişinin o anı için bir sayma sürecinin üretilip üretilemeyeceğini ispatlamaya dayanmaktadır. Busy Beaver Turing Makinalarının simülasyon geçmişlerinin her noktasında zamansal konumunu takip edebiliyor olması gerektiğini göstererek, önerdiğimiz bilgi yoğunluğu ölçütünü zamansal konum takibinin mümkünlüğünü test ederek takibin imkansızlığı halini sonlanmama öngörüsü kanıtı olarak kullanıyoruz. Normal makina benzetimine çok az bir hesapsal yük ekleyerek verimli bir erken sonlanma/sonlanmama öngörüsüne ulaşabiliyoruz.

1. Introduction

In 1962 T.Rado introduced the "busy beaver problem" in his paper “On non-computable functions” [1] , defined as follows. Let M be a Turing machine with n states (plus an anonymous halting state) and two symbols that is conventionally assumed as 1 and 0; no blank symbol is used and tape is assumed to be filled with symbol 0 at the beginning. At each step M has to write a symbol to the tape, move the machine head one symbol to the right or left, and change state. To be a valid Busy Beaver machine, M must eventually halt when started on an empty two-way infinite tape. Machine’s score according to the problem is the number of 1's left on the tape when M halts. Thus M tries to write as many 1's on the tape as it can, but it must halt. Rado defines his infamous ∑ function as ∑(n) being the maximum possible score for a valid n-state entry.

The theoretical interest in this competition arises from the fact that, although ∑(n) is simply the maximum of a finite set, the ∑ function itself is not computable.[1] Furthermore, ∑ is eventually greater than any given computable functions. In fact ∑ is very valuable to construct a specific non-computable function

1.1. Quadruple vs. quintuple definition of TM

A Turing Machine can be defined by a sextuple *(Q,P,G,d,s,f)* [2], where :

- Q is a finite set of states
- P is an alphabet of input symbols
- G is an alphabet of tape symbols
- δ is the transition function
- s in Q is the start state

[1] Neither the maximum shift function S is computable; both ∑ and S grows faster than any computable function; but it is possible to compute the values of ∑ and S for very small n values.

• f in Q is the final state.

The original definition proposed by Rado [1] for Busy Beavers, considered deterministic 5-tuple TMs with $n+1$ states (n states and an anonymous halting state). In each transition TM writes a symbol to the tape and moves the head left or right. This is the quintuple definition of TM where the state transition function δ for this definition has the following form:

δ: $Q\times\Gamma \rightarrow Q\times\Gamma\times\{L,R\}$ where $\Gamma \in P$, $P = G$ and $s,f \in Q$

There is also the quadruple definition of the TM having the following state transition function δ:

δ: $Q\times\Gamma \rightarrow Q\times\{\Gamma\cup\{L,R\}\}$

The Busy Beaver problem defined on Rado's original paper uses the quintuple definition instead of quadruple definition; and through out this paper all definitions and statements are made assuming a quintuple definition. Busy Beaver problem using quadruple machine definition is also investigated in the literature.

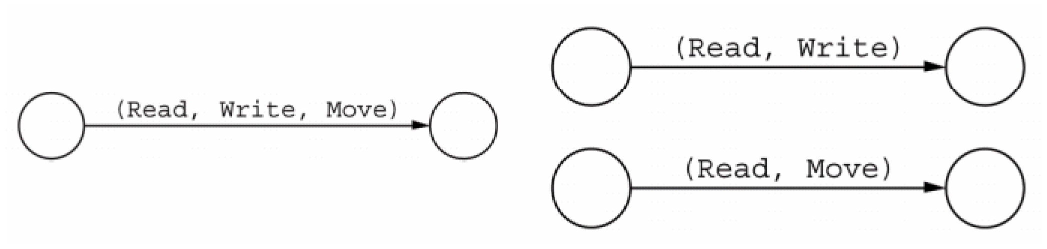

fig. 1.1 – Transitions with Quintuple vs. Quadruple Definition

According the quintuple definition the total number of valid n-state ($n+1$ states including anonymous halting state) m-symbol TMs is $(m.(n+1).2)^{n.m}$. According to this the space of valid Turing Machines with 5-states and 2-symbols consists of $(2.6.2)^{2.5} = 24^{10}$ TM instances, including isomorphic machines with equivalent behavior and machines with no connectivity between starting and halting states. The isomorphic machines and machines with disconnected or sub-optimal topologies can be omitted during enumeration (before emulation) with a technique named *tree normal form* (commonly abbreviated as TNF) to obtain a complete and effective mapping to a solution sub-space of the problem. TNF enumeration is extensively investigated on the further sections.

1.2. Halting Problem

Halting problem is a decision problem about properties of computer programs given a fixed Turing-complete model of computation. The problem is to decide, given a program and input pair on a chosen computational model, whether this system will eventually halt. No resource limitation of memory or execution time on the program's execution is assumed so system’s execution can take arbitrarily long time, and use arbitrarily much storage space, till halting. The problem is simply about whether a particular program will ever halt on a given input.

The popularity of the halting problem in literature comes from its undecidable nature. Lack of a computable function that correctly determines whether a program halts or not is easy to prove by contraction, and has extensively been referenced on literature.

Decision problems are commonly represented by the set of objects having the property defined in question. The halting set

H := { (p, i) | program p halts if run with input i}

represents the halting problem.

Set *H* is recursively enumerable, so there is at least one computable function *F* that lists all pairs *(p,i)* that belong to *H.* This computable function simulates all programs on all inputs in parallel similarly to a multithreaded computer program and indicates whenever one of the programs being simulated halts.

There are many equivalent formulations of the halting problem; any set whose Turing degree[2] is the same as that of the halting problem can be thought of as such a formulation.

1.3. Undecidability of Halting Problem

Halting problem is provably undecidable. Undecidability of it is frequently proven with a diagonalization proof. The following common proof from the literature shows that there is no total computable function deciding whether an arbitrary program *p* halts on arbitrary input *i*; thus the following function *halt* is not computable:

$$\boxed{halt(p,i) = \mathbf{1} \text{ iff program } p \text{ halts when run on input } i; \ \mathbf{0} \text{ other wise}}$$

Here program *p* refers to the p^{th} program from the enumeration of all valid programs of a specific Turing-complete computation model.

If we can show that every totally computable function having two arguments is different from the necessary function *halt,* the undecidability of halting problem becomes established. Let *f* be an arbitrary totally computable function with two arguments, we construct the following partial function *g* which is also computable:

$$\boxed{g(i) = \mathbf{0} \text{ iff } f(i,i) = 0; \ \textbf{undefined} \text{ other wise}}$$

[2] The Turing degree or degree of unsolvability of a set of natural numbers measures the level of algorithmic unsolvability of the set. The concept of Turing degree is fundamental in computability theory, where sets of natural numbers are often regarded as decision problems; the Turing degree of a set tells how difficult it is to solve the decision problem associated with the set.

- As *g* is partially computable, there exists at least one program *p´* that gets assigned to it, in the chosen Turing-complete model of computation (i.e. program *e* computes function *g*).

Definition of *g* imposes one of the following cases to hold:

- $(g(e)=0 \wedge f(e,e)=0) \rightarrow halt(e,e) = 1$ (because program *e* halts on input *e*)
- $(g(e)$ is undefined $\wedge f(e,e) \neq 0) \rightarrow halt(e,e) = 0$ (because program *e* does not halt on input *e)*

In either case, *f* cannot be the same function as *halt*. Because *f* is an arbitrary totally computable function having two arguments and all such functions must differ from function *halt*.

The diagonalization proof above can also be constructed as a two-dimensional array with one column and one row for each natural number. Where value of *f* for *(i,j)* resides at column *i*, row *j*. As *f* is a totally computable function, any element of the array can be calculated with *f*. The construction of the function *g* can be visualized as the main diagonal of this array. If the array has *0* at position *(i,i)*, then *g(i)* is *0*, otherwise, *g(i)* is undefined. The contradiction comes from the fact that there exists a column *e* of the array corresponding to *g* itself. If *f* was our halting function *halt*, there should be a *1* at position *(e,e) iff g(e)* is defined, but *g* is constructed such that *g(e)* is defined *iff* there is *0* at position *(e,e)*.

1.4. Consequences of Undecidability of Halting Problem

Importance of the halting problem is due to the fact that it is one of the first problems proven to be undecidable. Turing's undecidability proof is sent to press in May 1936, while Church's proof of the undecidability of a problem in his lambda calculus had already been published as of April 1936. Later many other similar problems have been described. The typical method of proving a problem to be undecidable is by using the technique of reduction; by transforming instances of an undecidable problem into instances of a new problem, if a solution to a new problem would be found it could be used to decide the undecidable problem. As it’s known that there is no method to decide the former problem, no method can decide the new problem either.

A consequence of the halting problem's undecidability is that there can’t be a general algorithm that decides whether a given statement about natural numbers is true; because the proposition stating that a certain algorithm will halt given a certain input can be converted into an equivalent statement about natural numbers. Assuming we had an algorithm that could solve every statement about natural numbers, it could solve the latter statement; but this would determine whether the former program halts which is impossible, since the halting problem is proven to be undecidable.

A second consequence of the undecidability of the halting problem is Rice's theorem which states that the truth of any non-trivial statement about the function that is defined by an algorithm is undecidable. The decision problem "will algorithm A halt for the input 0" is already undecidable. This theorem holds for the function defined by the algorithm and not the algorithm itself. It is possible to decide if an algorithm will halt within a reasonable number of steps, but this is not a statement about the function that is defined by the algorithm.

Gregory Chaitin defined a halting probability, represented by the symbol Ω, a type of real number that represents the probability that a randomly produced program halts. Real numbers of this type have the same Turing degree as the halting problem. It is a transcendental number which can be defined, but cannot be computed completely. It can be proven that there is no algorithm producing the digits of Ω, although first digits of it can be calculated to a precision for simple cases.

Although Turing's proof shows that there can’t be any general method or algorithm to determine whether an algorithm halts, individual instances of halting problem is susceptible to attack. For a specific algorithm, it can often be shown that it must halt for any input, and in fact software analysts do that as part of correctness proofs; but each proof has to be developed specifically for a specific algorithm; there isn't an automated, general way to determine whether an algorithm implemented as a Turing machine halts. However, there are some heuristics that can be used in an automated fashion to attempt to construct a proof, which succeed frequently on typical programs. This field of research is known as automated termination analysis.

1.5. Proof of Non-computability of ∑(n)

The non-computability of ∑(n) is proven by contradiction. The proof is as follows:

1. $\forall n\ \sum(n+1) > \sum(n)$ (Simple to establish by replacing the halting state with an intermediate state leading to halting state on all inputs)

2. Lets suppose a Turing Machine A, on input 1^n halts with $1^{\Sigma(n)}$ on its tape. Let s_A denote the number of states of A.

3. Lets suppose another Turing Machine B that writes 1^k on its tape and then enters A's starting state; thus B halts with $1^{\Sigma(k)}$ on its tape.

4. It is evident that B can be constructed using $\sqrt{k} + s_A$ states, which is less than k, for k sufficiently large.

5. Therefore B must produce fewer than ∑(k) 1's

Contradiction

2. Historical Survey of Busy Beavers

Parts of the information presented on this section is derived from the website of Pascal Michel [3] where he keeps track of current record holders of Sigma and Omega for different values of state count and tape alphabet size; he also provides peer reviewing of machines for the new record contesters by independently simulating their proposed machine configurations, which is extremely resource expensive as the necessary simulation steps lately became as high as on the order of 10^{14072} as in the case of 4-state, 3-symbol Turing Machine of T. and S. Ligocki proposed on January 2008.

2.1. Current Best Candidates

The tables below present the evolution of the lower bounds and values for ∑(N) in the quintuple variant of the problem. The exact value of ∑ is known for up to 4 state TMs.

table 2.1 – Busy beaver best candidates quintuple variant

n	∑(n)	Omega(n)	Authors, Date
1	1	1	Lin and Rado, 1962
2	4	6	Lin and Rado, 1962
3	6	21	Lin and Rado,1965
4	13	107	Brady,1975
5	≥ 501	≥ 134467	U.Schult, 1983
5	≥ 1915	≥ 2133492	G.Uhing,1984
5	≥ 4098	≥ 47176870	Marxen and Buntrock [3] [4], 1990
6	≥ 136612	≥ 13122572797	Marxen and Buntrock [4], 1990
6	≥ 95524079	$\geq 8690333381690951$0	Marxen, 2002
6	$\geq 6.427499 \times 10^{462}$	$\geq 6.196913 \times 10^{925}$	Marxen, 2002
6	$\geq 1.29149 \times 10^{865}$	$\geq 3.00233 \times 10^{1730}$	Marxen, 2002
6	$\geq 2.5 \times 10^{881}$	$\geq 8.9 \times 10^{1762}$	Terry and Shawn Ligocki, 2007
6	$\geq 4.6 \times 10^{1439}$	$\geq 2.5 \times 10^{2879}$	Terry and Shawn Ligocki, 2007

For the quadruple variant the known best candidates are the following:

table 2.2 - Busy beaver best candidates quadruple variant

n	∑(n)	Omega(n)	Authors, Date
1	1	1	Trivial
2	2	3	Trivial
3	3	7	Trivial
4	5	6	Unknown
5	≥ 11	≥ 52	Unknown
6	≥ 21	≥ 125	Cris Nielsen, 1996
6	≥ 25	≥ 256	Machado and Pereira, 1999
7	≥ 37	≥ 253	Lally, Reineke and Weader, 1997
7	≥ 196	≥ 13683	Machado and Pereira, 2002
8	≥ 86	≥ 1511	Norman, Chick e Marcella, 1996
8	≥ 672	≥ 198340	Machado and Pereira, 2002

[3] In 1990 Heiner Marxen took about 240 processor hours to obtain ∑*(5)* ≥ *4098* with a 33 Mhz Clipper CPU.

2.2. Historical evolution of relations

- Rado (1962) defined S(n) and ∑(n), and showed that they are non-computable functions [1]. He proved that

  $S(n) < (n+1)\,\Sigma(5n) \times 2^{\Sigma(5n)}$

- Julstrom (1992) proved that

  $S(n) < \Sigma(20n)$

- Wang and Xu (1995) proved that

  $S(n) < \Sigma(10n)$

- Yang, Ding and Xu (1997) proved that

  $S(n) < \Sigma(8n)$

  and that there is a constant c such that

  $S(n) < \Sigma(3n+c)$

- Ben-Amram, Julstrom and Zwick (1996) proved that

  $S(n) < \Sigma(3n+6)$ and $S(n) < (2n-1)\,\Sigma(3n+3)$

- Ben-Amram and Petersen (2002) proved that there is a constant c such that

  $S(n) < \Sigma(n + 8n / \log_2 n + c)$

2.3. Chronological Summary

table 2.3 – Chronological summary

1963	Rado, Lin	**$S(2,2) = 6$, $\sum(2,2) = 4$** **$S(3,2) = 21$, $\sum(3,2) = 6$**
1964	Brady	(4,2)-TM: $s = 107$, $\sum = 13$
1964	Green	(5,2)-TM: $\sum = 17$ (6,2)-TM: $\sum = 35$
1972	Lynn	(5,2)-TM: $s = 435$, $\sum = 22$ (6,2)-TM: $s = 522$, $\sum = 42$
1974	Lynn	(5,2)-TM: $s = 7{,}707$, $\sum = 112$
1974	Brady	**$S(4,2) = 107$, $\sum(4,2) = 13$**
1983	Brady [5]	(6,2)-TM: $s = 13{,}488$, $\sum = 117$
January 1983	Schult	(5,2)-TM: $s = 134{,}467$, $\sum = 501$ (6,2)-TM: $\sum = 2{,}075$
December 1984	Uhing	(5,2)-TM: $s = 2{,}133{,}492$, $\sum = 1{,}915$
February 1986	Uhing	(5,2)-TM: $s = 2{,}358{,}064$
1988	Brady	**(2,3)-TM: $s = 38$, $\sum = 9$** (2,4)-TM: $s = 7{,}195$, $\sum = 90$
February 1990	Marxen, Buntrock	**(5,2)-TM: $s = 47{,}176{,}870$, $\sum = 4{,}098$** (6,2)-TM: $s = 13{,}122{,}572{,}797$, $\sum = 136{,}612$
September 1997	Marxen, Buntrock	(6,2)-TM: $s = 8{,}690{,}333{,}381{,}690{,}951$ $\sum = 95{,}524{,}079$
August 2000	Marxen, Buntrock	(6,2)-TM: $s > 5.3 \times 10^{42}$, $\sum > 2.5 \times 10^{21}$
October 2000	Marxen, Buntrock	(6,2)-TM: $s > 6.1 \times 10^{925}$, $\sum > 6.4 \times 10^{462}$
March 2001	Marxen, Buntrock	(6,2)-TM: $s > 3.0 \times 10^{1730}$, $\sum > 1.2 \times 10^{865}$
October 2004	Michel	(3,3)-TM: $s = 40{,}737$, $\sum = 208$
November 2004	Brady	(3,3)-TM: $s = 29{,}403{,}894$, $\sum = 5{,}600$
December 2004	Brady	(3,3)-TM: $s = 92{,}649{,}163$, $\sum = 13{,}949$
February 2005	T. and S. Ligocki	**(2,4)-TM: $s = 3{,}932{,}964$, $\sum = 2{,}050$** (2,5)-TM: $s = 16{,}268{,}767$, $\sum = 4{,}099$ (2,6)-TM: $s = 98{,}364{,}599$, $\sum = 10{,}574$
April 2005	T. and S. Ligocki	(4,3)-TM: $s = 250{,}096{,}776$, $\sum = 15{,}008$ (3,4)-TM: $s = 262{,}759{,}288$, $\sum = 17{,}323$ (2,5)-TM: $s = 148{,}304{,}214$, $\sum = 11{,}120$ (2,6)-TM: $s = 493{,}600{,}387$, $\sum = 15{,}828$
July 2005	Souris	(3,3)-TM: $s = 544{,}884{,}219$, $\sum = 36{,}089$
August 2005	Lafitte, Papazian	(3,3)-TM: $s = 4{,}939{,}345{,}068$, $\sum = 107{,}900$ (2,5)-TM: $s = 8{,}619{,}024{,}596$, $\sum = 90{,}604$
September 2005	Lafitte, Papazian	(3,3)-TM: $s = 987{,}522{,}842{,}126$, $\sum = 1{,}525{,}688$ (2,5)-TM: $\sum = 97{,}104$
October 2005	Lafitte, Papazian	(2,5)-TM: $s = 233{,}431{,}192{,}481$, $\sum = 458{,}357$ (2,5)-TM: $s = 912{,}594{,}733{,}606$, $\sum = 1{,}957{,}771$

December 2005	Lafitte, Papazian	(2,5)-TM: s = 924,180,005,181
April 2006	Lafitte, Papazian	(3,3)-TM: s = 4,144,465,135,614, $\sum$ = 2,950,149
May 2006	Lafitte, Papazian	(2,5)-TM: s = 3,793,261,759,791, $\sum$ = 2,576,467
June 2006	Lafitte, Papazian	(2,5)-TM: s = 14,103,258,269,249, $\sum$ = 4,848,239
July 2006	Lafitte, Papazian	(2,5)-TM: s = 26,375,397,569,930
August 2006	T. and S. Ligocki	(3,3)-TM: s = 4,345,166,620,336,565 $\sum$ = 95,524,079 (2,5)-TM: $s > 7.0 \times 10^{21}$, $\sum$ = 172,312,766,455
September 2007	T. and S. Ligocki	(3,4)-TM: $s > 5.7 \times 10^{52}$, $\sum > 2.4 \times 10^{26}$ (2,6)-TM: $s > 2.3 \times 10^{54}$, $\sum > 1.9 \times 10^{27}$
October 2007	T. and S. Ligocki	(4,3)-TM: $s > 1.5 \times 10^{1426}$, $\sum > 1.1 \times 10^{713}$ (3,4)-TM: $s > 4.3 \times 10^{281}$, $\sum > 6.0 \times 10^{140}$ (3,4)-TM: $s > 7.6 \times 10^{868}$, $\sum > 4.6 \times 10^{434}$ (3,4)-TM: $s > 3.1 \times 10^{1256}$, $\sum > 2.1 \times 10^{628}$ (2,5)-TM: $s > 5.2 \times 10^{61}$, $\sum > 9.3 \times 10^{30}$ (2,5)-TM: $s > 1.6 \times 10^{211}$, $\sum > 5.2 \times 10^{105}$
November 2007	T. and S. Ligocki	(6,2)-TM: $s > 8.9 \times 10^{1762}$, $\sum > 2.5 \times 10^{881}$ **(3,3)-TM: s = 119,112,334,170,342,540** **$\sum$ = 374,676,383** (4,3)-TM: $s > 7.7 \times 10^{1618}$, $\sum > 1.6 \times 10^{809}$ (4,3)-TM: $s > 3.7 \times 10^{1973}$, $\sum > 1.6 \times 10^{986}$ (4,3)-TM: $s > 3.9 \times 10^{7721}$, $\sum > 4.0 \times 10^{3860}$ (4,3)-TM: $s > 3.9 \times 10^{9122}$, $\sum > 2.5 \times 10^{4561}$ (3,4)-TM: $s > 8.4 \times 10^{2601}$, $\sum > 1.7 \times 10^{1301}$ (3,4)-TM: $s > 3.4 \times 10^{4710}$, $\sum > 1.4 \times 10^{2355}$ (3,4)-TM: $s > 5.9 \times 10^{4744}$, $\sum > 2.2 \times 10^{2372}$ **(2,5)-TM: $s > 1.9 \times 10^{704}$, $\sum > 1.7 \times 10^{352}$** (2,6)-TM: $s > 4.9 \times 10^{1643}$, $\sum > 8.6 \times 10^{821}$ (2,6)-TM: $s > 2.5 \times 10^{9863}$, $\sum > 6.9 \times 10^{4931}$
December 2007	T. and S. Ligocki	**(6,2)-TM: $s > 2.5 \times 10^{2879}$, $\sum > 4.6 \times 10^{1439}$** (4,3)-TM: $s > 7.9 \times 10^{9863}$, $\sum > 8.9 \times 10^{4931}$ (4,3)-TM: $s > 5.3 \times 10^{12068}$, $\sum > 4.2 \times 10^{6034}$ **(3,4)-TM: $s > 5.2 \times 10^{13036}$, $\sum > 3.7 \times 10^{6518}$**
January 2008	T. and S. Ligocki	**(4,3)-TM: $s > 1.0 \times 10^{14072}$, $\sum > 1.3 \times 10^{7036}$** **(2,6)-TM: $s > 2.4 \times 10^{9866}$, $\sum > 1.9 \times 10^{4933}$**

2.4. Summary Tables

S(n state,m symbol)

table 2.4 – S(2-6,2-6)

6 symbols	$> 2.4 \times 10^{9866}$				
5 symbols	$> 1.9 \times 10^{704}$	?			
4 symbols	≥ 3,932,964	$> 5.2 \times 10^{13036}$	?		
3 symbols	≥ 38	$> 1.1 \times 10^{17}$	$> 1.0 \times 10^{14072}$	?	
2 symbols	**6**	**21**	**107**	≥ 47,176,870	$> 2.5 \times 10^{2879}$
	2 states	***3 states***	***4 states***	***5 states***	***6 states***

∑(n state,m symbol)

table 2.5 – ∑(2-6,2-6)

6 symbols	$> 1.9 \times 10^{4933}$				
5 symbols	$> 1.7 \times 10^{352}$	?			
4 symbols	≥ 2,050	$> 3.7 \times 10^{6518}$	?		
3 symbols	≥ 9	≥ 374,676,383	$> 1.3 \times 10^{7036}$	?	
2 symbols	**4**	**6**	**13**	≥ 4098	$> 4.6 \times 10^{1439}$
	2 states	***3 states***	***4 states***	***5 states***	***6 states***

2.5. Busy Beaver State Topology Samples

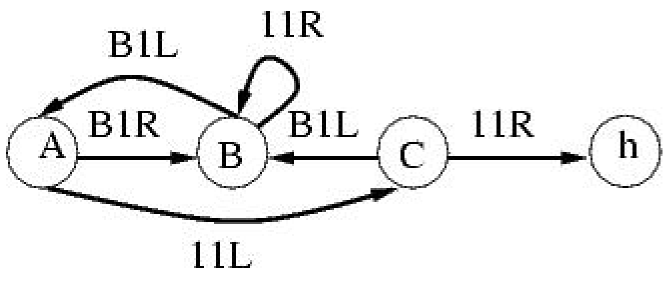

fig. 2.1 – 3-State Busy Beaver - Lin and Rado,1965

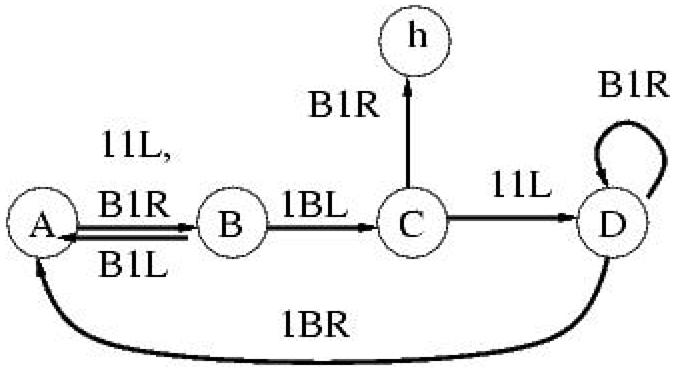

fig. 2.2 – 4-State Busy Beaver – Brady 1975

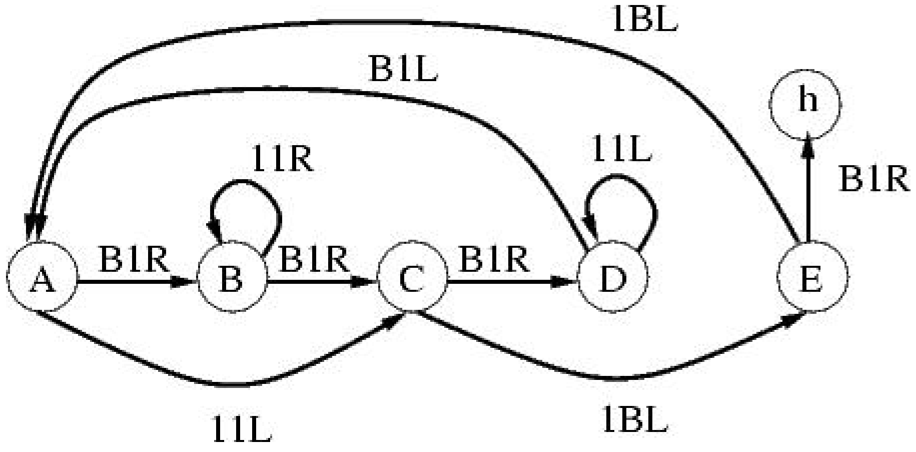

fig. 2.3 – 5-State Busy Beaver Candidate – Marxen 1990

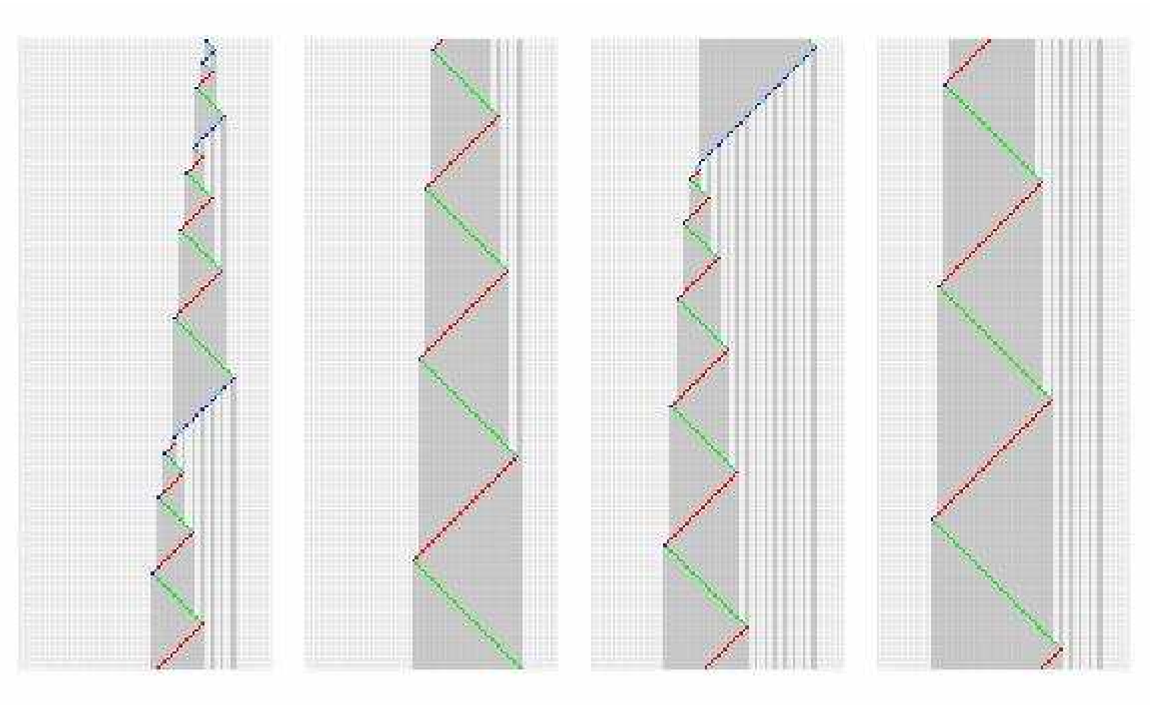

fig. 2.4 – Tape trace of first 800 steps of the TM in fig. 2.3

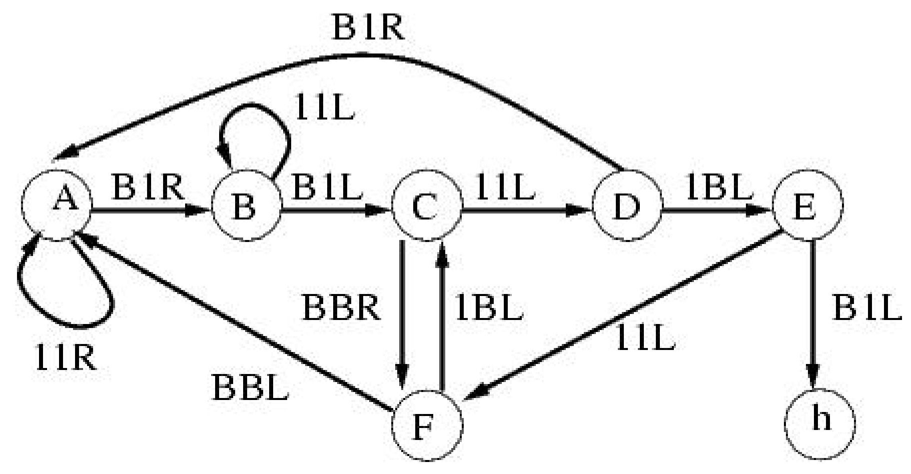

fig. 2.5 – 6-State Busy Beaver Candidate

3. Search Space

3.1. Size

As mentioned the space of Turing machines for n states and m tape symbols consists of $(2m.(n+1))^{n.m}$ valid definition of Turing machines. The space size grows exponentially in function of state and symbol count but can be pruned to a much lower size using the previously mentioned Tree Normal Form which still grows exponentially. The following table shows the number of valid Turing Machines definitions and number of TMs enumerated by TNF for several values of n and m. As there are different ways to implement the TNF enumerator with different optimizations included, the values on the following table is specific to our implementation detailed in later sections.

table 3.1 – Number of valid TMs

State	Symbol	All Valid TMs	TNF Enumerated TMs	Ratio
3	2	16.777.216	16.656	0,09928%
4	2	25.600.000.000	2.902.620	0,01134%
5	2	63.403.380.965.376	671.859.240	0,00106%
6	2	232.218.265.089.212.000	?	?
2	3	34.012.224	11.340	0,00033%
3	3	2.641.807.540.224	181.656.744	0,00007%
4	3	531.441.000.000.000.000	?	?
5	3	221.073.919.720.733.000.000.000	?	?
2	4	110.075.314.176	4.555.488	0,00414%
3	4	1.152.921.504.606.850.000	?	?
4	4	42.949.672.960.000.000.000.000.000	?	?

3.2. Smoothness

Like most dynamical and chaotic systems Turing Machines are very sensitive to initial conditions, like the contents of the tape at the beginning or the transition function of the Turing Machine. No smooth gradient exists between the outputs of similar Turing machines. Of course in order to talk about similarity, we first need to define a topology with a distance function and neighborhoods for Turing Machines.

3.3. Topology

Here we propose an alternate topologic representation of sets of Turing machines belonging to the same class of tape alphabet size and state count. Such a set of valid Turing machines for a given state number and alphabet size can be represented as a topological metric space by defining a neighborhood function where neighborhood system consists of Turing Machines with their definitions differing from the center machine of neighborhood open ball by only a single state transition, a single shift direction or a single alphabet character to write. This definition of neighborhood is the most intuitive in terms of both mathematical topology and computational hill climbing type evolutionary algorithms. Also an intuitive distance metric for a such topology would be a metric distance similar to Manhattan Distance for spaces with more than 3 dimensions.

A Space of Turing machines having *n* states and *m* tape alphabet symbols can be represented as follows : Let's assume each possible transition of the form [$n,m,2$] (state to go, symbol to write, one of two possible directions for head shift) is represented as a point in a three dimensional discreet space; and a TM is represented as a vector consisting of $n \times m$ points (a transition for each possible symbol read at each possible state). As each point would have $m+n-1$ neighbors[4], therefore a vector representing a particular TM would have m^2n+n^2m-nm neighbor[5] vectors (neighbor TMs). Assuming such neighborhood, the difference of resulting tape configurations for Turing Machines of a neighborhood is observed to be far from being smooth. A single change on machine definition commonly changes either the loop behavior of TM or the tape output significantly.

This last observation defines one of the significant properties of the TM space in context of the choice of how to explore the solution space. It is known that from the range of evolutionary algorithms genetic algorithms are better suited to optimization of functions with continuous surfaces with smooth transitions; on the other hand hill

[4] *$(n-1)+(m-1)+1$ (change transition to another state, or to write another symbol or to other direction)*
[5] *$(n \times m)(m+n-1)$ (sum of possible transition changes for each transition of machine)*

climbing doesn’t necessarily impose such a constraint while benefiting from it if present.

4. Exploring Search Space

4.1. Exhaustive Enumeration

Exhaustive enumeration of the search space is, while being complete neither efficient nor feasible, as the number of possible TMs increases exponentially both for the number of states and for the number of symbols. In context of Busy Beaver problem this kind of full enumeration is only defendable for cases like comparative study, and classification of redundancy classes with their sizes.

4.2. Random Sampling

Random sampling of Turing Machines belonging to a set is one of the enumeration techniques we have implemented; but obviously as this kind of enumeration is not complete, it only serves for statistics gathering purposes to fine-tune other algorithms with obtained statistical properties of machine space.

Still random sampling can be implemented with different levels of complexity. First of all assigning random values adhering to respective ranges for each variable of a machine definition does neither necessarily nor likely produce a possibly halting Turing Machine. Most randomly generated machines suffer from disconnected state transition topologies and other sub-optimal machine configurations.

To prevent this we implemented a smarter version of random machine sampler code which checks state connectivity and makes sure that each state and transition is used and unique. Another technique for smart random sampler is to force the first transition to write a “1” symbol and shift to a specific direction. This technique prevents the enumeration of mirror machines which exhibit the exact same behavior mirrored according to starting point on tape.

4.3. Evolutionary approaches

In order to explore subsets of all possible TMs in context of Busy Beaver problem, many evolutionary approaches have been proposed throughout the literature [6] [11]. Genetic Algorithm is the most prevalent of the evolutionary computation methods; to employ GA on a problem first a suitable encoding for the possible solutions of the problem must be chosen. In the case of the busy beaver problem the state transition function of a TM is the proposed solution for which the encoding should be built upon.[6] GA relies on strings of specific alphabets where genetic operators like mutation and cross-over gets applied. The alphabet of the GA representation for a TM should not be confused with the symbols used for the TM actually the most natural way to encode the state transition function of a TM with n states and m symbols is to use a string of n by m characters from an alphabet. The Alphabet of the GA can be defined as characters consisting of all possible triples of the form [next state × shift direction × TM symbol to write] which is very similar to the points defined on space topology section. For TMs with 5-state (6 with anonymous halting state), 2-symbol a natural encoding for GA have an alphabet consisting of all the 24 possible triples; and each encoding defining an individual is a string (also called chromosome in GA jargon) of 10 characters from that string.

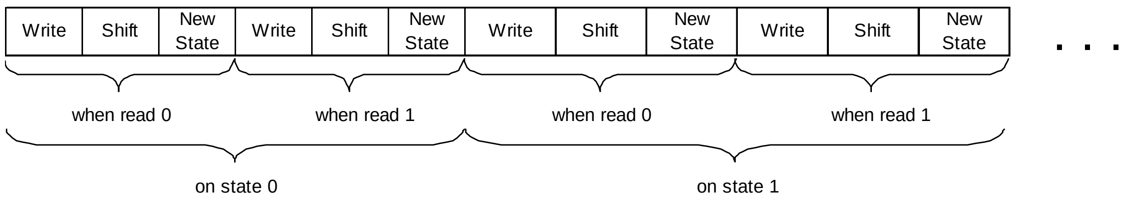

fig. 4.1 – A possible genetic encoding for Turing Machine Setup

Genetic Algorithms try to evolve better performing codes or machines at each generation according to a fitness function which is the heuristic for picking the better

[6] Inherently the number states and symbols of the Busy Beaver problem is necessary to define a possible solution but we assume the computation is performed for a specific BB(state,symbol) problem thus these numbers are assumed to be constant which don't require an encoding on GA population.

samples. In the case of Busy Beaver Problem the fitness function can be defined as the number of 1s left written on tape if a TM halts before timing out according to a specified hard coded shift limit. A further extension to fitness function can be to employ a weighted sum of the shifts taken before halt to favor more productive TMs in terms of numbers of 1s written for number of shifts taken.

Employing Genetic Algorithms on executable codes or configurations of execution machines is conventionally called Genetic Programming. Like all evolutionary computation optimizations Genetic Algorithms and Genetic Programming is not deterministic or complete.

4.4. Hill climbing

Hill climbing is a greedy, local search based, optimization technique. Hill climbing can be used on problems where multiple solutions with different performance or fitness exist. Hill climbing implementations start with random solutions and sequentially make small changes to the solution and keeping only the improved ones. At some point the algorithm arrives at a solution where no improvement can be seen on any solution neighbor to that one on the solution space thus the algorithm terminates. Hill climbing is not guaranteed to reach the optimal solution but ideally the final solution is close to optimal solution for most of the time, as this is the characteristic of greedy algorithms.

Many NP-Complete problems can be tackled with hill climbing easily when sub-optimal solutions are also acceptable. Hill climbing operates on a discreet space of solutions; the continuous counterpart of hill climbing is called gradient ascend/descent. Like genetic algorithms search space topology is important for hill climbing; most importantly the connectedness and smoothness of it. Assuming solution space consists of discrete solutions with a neighborhood; search space can naturally be represented as a graph with vertices where edges represent the distance metric or similarity of each solution. Hill climbing will explore the graph vertex to vertex by monotonically increasing (or decreasing) the fitness function $f(v)$ where v denotes the visited vertex/solution. As intended, the space topology for TMs that we proposed on the previous section fits conveniently to the type of discreet space required by Hill Climbing.

Hill climbing is very similar to genetic algorithm with no cross-over operator. An elitist genetic algorithm running with mutation operator only has an operating mode between hill climbing and beam search[7].

[7] An optimized version of best-first search

5. Implementation

Our base Turing Machine emulator’s implementation is pretty much straight forward. We have implemented the emulator both as native code using Borland Delphi Compiler to compile Object Pascal source code to native x86 ASM code; and as managed code using Microsoft C# compiler to compile C# source code to .Net 2.0 MSIL code.

The term managed code means executable code that runs under the management of a virtual machine, unlike native (unmanaged) code which is executed directly on the processor. The benefits of managed code include automatic memory garbage collection, strong type enforcement, advance range and bounds checking, isolation between application domains and similar security guarantees at a cost of minimal speed overhead.

The most common meaning of the term is the Microsoft’s description of programs that execute under the management of Microsoft’s CLR (Common Language Runtime) virtual machine of .NET Framework. Microsoft's main programming languages for creating managed code are C# and Visual Basic .NET. There are open source alternatives to Microsoft’s CLR such as MONO project and GNU Portable .Net which aims instruction level compatibility with MSIL. The Java programming language also creates managed code which is called bytecode in Java terminology and executed by the Java Virtual Machine which is part of Java Runtime Environment (JRE).

A common misconception about managed code is whether it is executed by an interpreter. Although managed code requires a set of runtime libraries and a runtime engine, neither Java nor Microsoft’s managed languages are interpreted languages (although interpreters exist for them); they are both JIT (Just-in time) compiled to native code thus exploiting the system architecture and computer state to the furthest extent. This is dismisses another widespread misconception about managed code being too slow. After JIT compilation (performed on runtime) managed code is indiscernible from native code for processor; when JIT’s benefit of machine architecture exploiting compilation and most importantly benefit of re-JIT compilation during runtime based on code hot spots detected with live run trace statistics allows the compilation process to adapt most perfectly to architecture and run time behavior. Most of time those advantages of JIT compilation either compensates the overhead or even make the code performance surpass statically compiled native code.

5.1. Data Types

Both managed and native code represent *shift direction*s with enumeration data types, while *alphabet symbol*s are represented using unsigned short integer (byte) data type in order to make machine tapes (one dimensional array of *symbol*s) as space efficient as possible, and state numbers are represented using integer data types.

The transitions and states have been implemented as separate inner classes on managed code to exploit pass by reference method call semantics, because managed code only allows pointers to be used inside of code sections marked with “unsafe code” attribute. This way while branching on Tree Normal Form enumerator, new machine prototype branches use the same references to instances of transitions or states if no modification to the values is needed.

On native code, transitions are heap allocated composite data structures and accessed using typed pointers, while states are implemented as arrays of transitions with symbol read used as index.

For native code a copy of a transition or a state (which is simply a collection of transitions with a unique identifier) can be generated with a memory move system call which is very efficient in terms of processor cycles as only the size of structure is dumbly copied from original variable to the newly allocated heap memory. On the other hand for the managed code creating a copy of a non-primitive, user defined, nested data structure, which is a class containing arrays of other classes, has to implement system defined *ICloneable* interface to hand tune the granularity of copy semantics, in order to work around possible errors caused by shallow copy, deep copy and memberwise copy which is common on complex structures with circular references. In order to mark a class as cloneable with *[Cloneable]* attribute either all the member fields should be of a cloneable type or the implementation of the first *ICloneable* interface should deal with proper copy generation of incompatible members recursively.

Same considerations mentioned above apply for the *[Serializable]* attribute where the persistence of a class to a stream (i.e. a file stream) is handled by *ISerializable* interface, which should be implemented; all member fields of a class should be serializable too in order to mark that class as serializable.

During the implementation of Tree Normal Form enumerator a need for a stack arised to hold “to be processed” nodes of tree where each node is a Turing Machine Prototype Class instance, which is basically a partially defined Turing Machine waiting to be incrementally defined in all possible ways with child branches. In native code the stack implementation is straight forward since a regular stack implemented on system libraries can hold typed pointers to our data types. On the other hand for managed code implementation, either a regular stack for *System.Object* type which is the ultimate ancestor to all other classes can be used, or a specific typed stack could be implemented. We decided to go with the latter because a regular stack required type casting from *System.Object* to our own class type for each pop operation from the stack which has a great impact on performance.

Our typed stack for managed code is implemented with the help of *generics* introduced with the second version of .Net framework. This way a list, queue, stack or any class can be generalized to all forms of user classes and data types without the performance loss due to object type casting and without the need to have different implementations for different types which in turn reduces memory footprint. Generics are very similar to templates in C++ and Java, where a class can be defined with declaration of some of the types postponed till instantiations of that class.

```
public class turingMachine : ICloneable
{
    public const int MAX_STATE = 5,
                     ALPHABET_SIZE = 2,


    public enum scrollDirections
    {
        left,
        right
    }


    public class State : ICloneable
    {
        public byte[] write = new byte[ALPHABET_SIZE];
        public State[] nextState = new State[ALPHABET_SIZE];
        public scrollDirections[] scrollDirection = new scrollDirections[ALPHABET_SIZE];
        public byte flag = 0;
    }


    public State[] states = new State[MAX_STATE + 1];
    public byte[] tape = new byte[MAX_TAPE];


    public State currentState;

    public int headPos = CENTER;
    public int cycle = 0;
    public int minx = CENTER;
    public int maxx = CENTER;
```

listing 1 – C# source of main fields defined at the beginning of emulator class

```
unit turingMachineClassU;

interface

const

      MAX_STEP = 2000; //step limiter for turing machines
      MAX_TAPE = 2000;

      ALPHABET_SIZE       = 2;
      MAX_STATE           = 5;

      MAX_BLOCKS          = 2*ALPHABET_SIZE*MAX_STATE;

type

     TScrollDirection = (scLeft,scRight);
     THaltingStatus   = (HUndecided,HHalt,HDisconnectedStates,HStateExhaustion);

     TSymbol = Byte;


     Pblock = ^TBlock; //pointer to TBlock
     TBlock = record
               write           : TSymbol;
               nextStateNo     : integer;
               scrollDirection : TScrollDirection;
              end;


     PState = ^TState; //pointer to TState
     TState = record
               write           : array [0..ALPHABET_SIZE-1] of TSymbol;
               nextStateNo     : array [0..ALPHABET_SIZE-1] of integer;
               scrollDirection : array [0..ALPHABET_SIZE-1] of TScrollDirection;
              end;


     TturingMachine = class
      public
       states : array [0..MAX_STATE-1] of TState;
       tape   : packed array [-MAX_STEP-1..MAX_STEP+1] of TSymbol;

       headPos        : integer;
       currentStateNo : integer;

       cycle          : integer;
       minx,maxx      : integer;

       haltingReason  : THaltingStatus;

       procedure randomizeTM;
       procedure reinitialize(rnd : boolean = true);

       procedure processStep;
       function  run : integer;

       procedure DrawTrace;
     end;
```

listing 2 – Object Pascal source of main fields defined at the beginning of emulator class

5.2. Emulator

The implementation of Turing Machine emulator is pretty similar in native and managed code. In both cases the *run()* method of Turing Machine class call the *processStep()* method continuously in a loop till the simulated machine reaches the halt state which is checked from the *haltingStatus* enumeration field or some cycle limit is exceeded.

processStep() method implements a single step of Turing Machine. When called it first reads the tape symbol under the tape head by accessing the tape array with head position field as index; then from the current state instance a reference to the respective transition instance is retrieved using symbol which is read. Using this reference to current transition, first the symbol to be written to tape is written to tape array; then the head position field is either incremented or decremented according to shift direction indicated by the current transition reference; and finally current state field is changed with the new state to be transitioned to by changing the reference on *currentState* field with the new one as indicated on current transition instance. Before returning *processStep()* increments a counter field called *cycle* in order to keep track of the number of steps of execution. Many similar counters are incremented or decremented both in *processStep()* and *run()* in order to gather statistical data of the execution trace; most important of those are the following two indexes. In order to be able to track the tape space explored till this shift, when the head position field is changed, two indexes showing the left and right bounds of explored section of the tape is updated if a new cell is explored from left or right.

```
public void processStep() {

    int symRead = tape[headPos];
    byte symWrite = currentState.write[symRead];

    tape[headPos] = symWrite;                                              //1.write
    if (currentState.scrollDirection[symRead] == scrollDirections.left)    //2.scroll tape
    {
        headPos--;
        if (headPos < minx) minx = headPos;
    }
    else
    {
        headPos++;
        if (headPos > maxx) maxx = headPos;
    }
    currentState = currentState.nextState[symRead];                        //3.change state
    if (currentState == states[MAX_STATE]) haltingReason = haltingStatus.halt;

    cycle++;
}


public haltingStatus run()
{

    while ((haltingReason == haltingStatus.undecided) && (cycle < MAX_STEP))
    {
        processStep();
    }

    return haltingReason;
}
```

listing 3 – C# source of Turing Machine class' emulator methods

```
// --------------------------     PROCESS STEP    --------------------------------
procedure TturingMachine.processStep;
var curSta    : PState;
    symRead   : TSymbol;
begin
 curSta := @states[currentStateNo];
 symRead := tape[headPos];

 tape[headPos] := curSta.write[symRead];                  //1.write

 if curSta.scrollDirection[symRead] = scLeft then begin   //2.scroll Tape
  dec(headPos);
  if headPos < minx then minx := headpos;
 end else begin
  inc(headPos);
  if headPos > maxx then maxx := headpos;
 end;

 currentStateNo := curSta.nextStateNo[symRead];           //3.change State

 inc(cycle);
end;


// --------------------------------    RUN   ------------------------------------
function TturingMachine.run : integer;
begin
 result := 0; //Undecided
 haltingReason := HUndecided;

 //        HALT STATE REACHED               TIME OUT                SPACE OUT
 while (currentStateNo < MAX_STATE) and (cycle < MAX_STEP) and ((maxx-minx) < MAX_TAPE) do
begin
  processStep;
 end;

 if (cycle <> MAX_STEP) and (tapeLen <> MAX_TAPE) then haltingReason := HHalt;
end;
```

listing 4 – Object Pascal source of Turing Machine class' emulator methods

5.3. Enumerator

Our enumerator code is based on the previously mentioned Tree Normal Form methodology; since enumerating other valid configurations has no pragmatic value in our case. As previously mentioned, TNF is based on emulation of partially defined Turing Machine definitions till an undefined transition is needed, then it recursively branches to incrementally defined machines having that transition defined in all possible ways. That's why an emulation loop is embedded inside the enumeration loop. In order to prevent recursive branching interfere with emulation loop, a recursion removal technique (similar to mentioned one on simple connectivity check) is applied on the enumerator code. To implement the recursion removal, the enumeration tree traversal is replaced with an enumeration loop. First a partially defined machine is "pop"ed from a stack of machines, second it is emulated till an undefined transition is about to be taken, then all possible transitions for that case are generated based on number of already visited states and number of undefined transitions left. Finally new Turing machine prototypes are generated by incrementally defining the first machine for each possible transition. While the newly generated machines get pushed to stack of machines the original machine is discarded as all possible new machines for its lineage has been derived and pushed to stack for further investigation.

Actual implementation of enumerator contains many further optimizations like not building a transition to halting state until only a single undefined transition is left and forcing the last transition to be a transition to halting state with writing 1 to tape; because any transitions to halting state defined earlier would produce a sub-optimal machine topology where halting transition doesn't benefit from the states to be defined or would have multiple transitions to halting state.

During the actual coding of the TNF enumerator, we noticed that fields of previously defined transitions which are represented as instances of *transition class* don't change within a as the traversal of machine space tree gets deeper. Therefore as an optimization we clone the new incrementally defined machines as shallow copies (not copying the objects in fields too, but by passing the same references hold on fields to copied objects field) of parent machine prototype, except the machine tape field because the machine tape works bi-directionally and its contents can change within that branch as deeper nodes are explored. A shallow copy of machine topology let the new machines skip instantiating and cloning the previously defined transitions, and only use a reference to them which is more efficient, both processing time wise and memory foot-print/fragmentation wise.

At the beginning of TNF enumeration there are $(n \times m)\text{-}1$ undefined transitions[8] so at each deepening level of tree traversal a new transition gets defined; but also at each level a not-yet-reached state of the machine topology becomes reachable too. As a result two concurrent branching factor limit applies to recursive traversal which can be expressed as $2 \times m \times min(tree\ depth + 1,n)$ [9] where n denotes the number of states and m denotes the number tape alphabet symbols. As the number of transitions is always greater than the number of states, the first $n\text{-}1$ levels [10] of tree has a branching factor of $2 \times m \times (tree\ depth + 1)$ and after that branching factor stabilizes as $2 \times m \times n$ till maximum depth, which is the undefined transition count mentioned at the beginning.

As the enumerator also contains an intrinsic emulator to keep track of the partial simulations of machine prototypes, when a machine reaches the halting state during its simulation the respective counter variables are incremented and a callback function is called for further statistics gathering if one is provided.

```
public void enumerateTNFtree()
{
    TMStack.Clear();
    TMStack.Push(new TMPrototype()); // First prototype - only first transition exists in it

    //  ----   enumeration loop   ----
    while (TMStack.Count > 0)
    {
        TMPrototype tm = (TMPrototype)TMStack.Pop();


        #region emulationLoop

        //  -------------------------   emulation loop   ------------------------------------
        while ((tm.currentState != turingMachine.MAX_STATE) &&
```

[8] minus one due to first transition being pre-defined
[9] plus one due to a new state becoming reachable at each level
[10] minus one due to starting state being reachable from the start

```
                (tm.cycle < turingMachine.MAX_STEP))
        {

            if (tm.nextTransition() != null)
            {
                tm.step();
                if (tm.nonHaltPrediction)
                {
                    predictionCounter++;
                    break;
                }
            }
            else
            {

                //1.Enumerate possible transitions to defined states
                for (byte sym = 0; sym < turingMachine.ALPHABET_SIZE; sym++)
                    for (int sta = 0; sta < tm.nextAvailableState; sta++)
                    {
                        //Shift Left
                        Transition tr = new Transition();
                        tr.nextState = sta;
                        tr.write = sym;
                        tr.scrollDirection = turingMachine.scrollDirections.left;

                        TMPrototype tm2 = (TMPrototype)tm.Clone();
                        tm2.transitions[tm2.currentState, tm2.tape[tm2.headPosition]] = tr;
                        tm2.undefinedTransitions--;
                        TMStack.Push(tm2);

                        //Shift Right
                        tr = new Transition();
                        tr.nextState = sta;
                        tr.write = sym;
                        tr.scrollDirection = turingMachine.scrollDirections.right;

                        tm2 = (TMPrototype)tm.Clone();
                        tm2.transitions[tm2.currentState, tm2.tape[tm2.headPosition]] = tr;
                        tm2.undefinedTransitions--;
                        TMStack.Push(tm2);
                    }


                //2.Enumerate possible transitions to a new state
                for (byte sym = 0; sym < turingMachine.ALPHABET_SIZE; sym++)
                {
                    //Shift Left
                    Transition tr = new Transition();
                    tr.nextState = tm.nextAvailableState;
                    tr.write = sym;
                    tr.scrollDirection = turingMachine.scrollDirections.left;

                    TMPrototype tm2 = (TMPrototype)tm.Clone();
                    tm2.transitions[tm2.currentState, tm2.tape[tm2.headPosition]] = tr;
                    if (tm2.nextAvailableState < turingMachine.MAX_STATE)
tm2.nextAvailableState++;
                    tm2.undefinedTransitions--;
                    TMStack.Push(tm2);

                    //Shift Right
                    tr = new Transition();
                    tr.nextState = tm.nextAvailableState;
                    tr.write = sym;
                    tr.scrollDirection = turingMachine.scrollDirections.right;

                    tm2 = (TMPrototype)tm.Clone();
                    tm2.transitions[tm2.currentState, tm2.tape[tm2.headPosition]] = tr;
                    if (tm2.nextAvailableState < turingMachine.MAX_STATE)
tm2.nextAvailableState++;
                    tm2.undefinedTransitions--;
                    TMStack.Push(tm2);
                }

                break; //  *** end of life of this tm prototype ***
            }

        } //  ------------------------  emulation loop   ------------------------

        #endregion


        if (tm.undefinedTransitions == 0)
        {

            if (tm.currentState == turingMachine.MAX_STATE) //---HALTER
            {
                haltingcounter++;
                if (callBack != null) callBack();
            }//---HALTER
            else //---UNDECIDED, MAX_STEP
            {
                undecidedcounter++;
                if (callBack != null) callBack();
            }//---UNDECIDED, MAX_STEP

        }

        tm = null; //free the prototype representing this branch

    } // --- enumeration loop ---
} // void enumerateTNFtree()
```

listing 5 – C# source of Tree Normal Form TM enumerator method

6. Optimizations

In coding part of busy beaver problem (as in all computation problems) implementation specific optimizations account for lesser performance gains compared to algorithmic optimizations. A common pitfall for programmers is to apply implementation related optimizations instead of devising a better algorithm with reduced time and space complexity. The following optimizations are the most common algorithmic improvements over the naïve algorithm of exhaustive enumeration with flat memory allocation.

6.1. Tree Normal Form

For any n-state Turing machine there are *n!* isomorphic machines which are functionally equivalent. Any machine can be reconstructed as permutations of state numbers by renaming the states, but as their behavior will remain identical concentrating our processing resources on only one instance for each of those equivalence classes greatly improves time and space complexity of covering the whole machine space.

Presence of multiple valid TMs with isomorphic configurations results on multiple equivalence classes of TM subsets with identical functionality. The most important equivalence class in literature is Tree Normal Form (abbreviated TNF) described by Heiner Marxen on his paper [4]. Almost all the following attacks to busy beaver problem from the literature use some form of machine normalization [7] except those which are based on evolutionary techniques instead of enumeration. Representing TMs on Tree Normal Form allows functionally equivalent TMs to be represented identically in a normalized way.

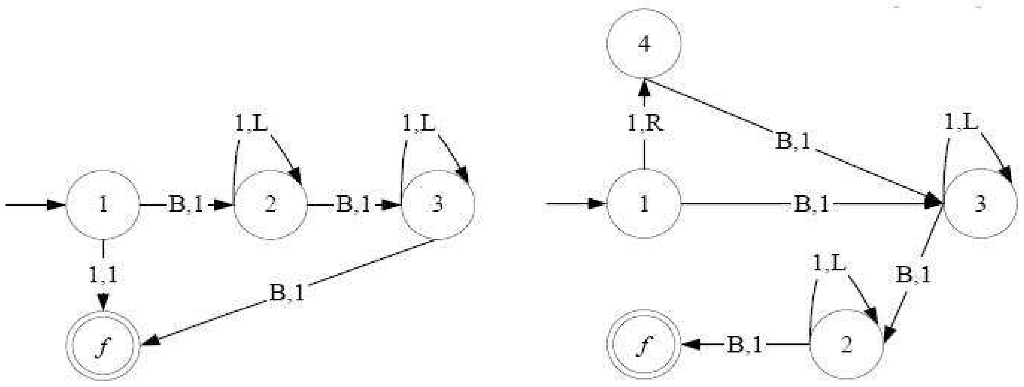

fig. 6.1 – Isomorphic Machine Topologies

Another benefit of enumerating machines according to Tree Normal Form is to conveniently eliminate sub-optimal topologies, like those which have multiple transitions to halting state and those which have states with no incoming transitions, during the enumeration saving us from unnecessary emulation.

TNF enumeration is based on partially defined Turing Machines topologies where not yet encountered transitions are not defined and unvisited states aren't given a number, keeping them equivalent. While using TNF the emulation and enumeration processes are fused to a single process. During the emulation of available machines when undefined transitions are requested, they get defined by forking the emulation to many clones of that machine, each having the undefined transition getting defined as one of possible valid transitions. This forking can be implemented as a recursion or back tracking. As long as a state isn't visited yet it remains anonymous with other unvisited states making them indiscernible. When valid transitions are enumerated upon request for definition of a transition, only the previously visited states are considered as anonymous states can not be used; but as long as there are still anonymous states left this transition is considered as it could also be a transition into the subset of undefined states of topology, thus an anonymous state (as all anonymous states are indiscernible, any one is as good as any other) is given a unique number making it part of the defined subset of topology. Obviously as there are more transitions than states, the undefined (anonymous) states deplete sooner than undefined transitions.

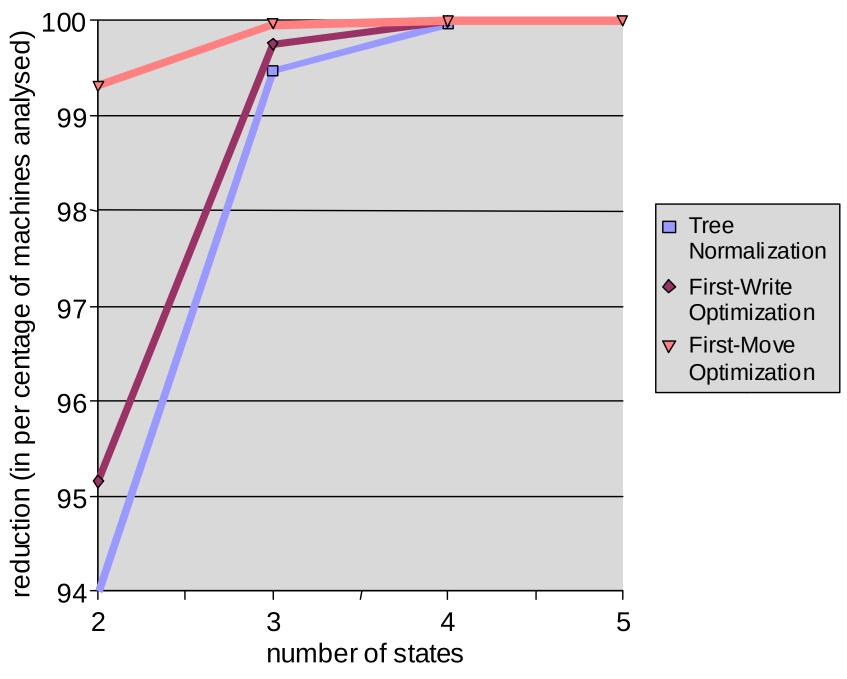

fig. 6.2 – Reduction Achieved by TNF [8] [11]

It can be seen from figure 9 that, as the number of states for considered busy beaver problem increases, additional optimizations to enumeration layer only marginally increase the effectiveness. For Turing Machines with 4 or more states the branch elimination factor (reduction of machines to consider) of enumerator converges to a range less than one percent, independent of the presence of other enumeration time optimizations. In light of this, our implementation of TNF enumeration only contains this two first move enforcing optimization on top of it although there many other esoteric extensions to TNF exist through out the literature.

It is obvious that the purpose of Busy Beaver problem is to leave a single machine which writes the maximum number of "1"s; therefore all kinds of eliminations

[11] Chart derived from data on Rensselaer AI & Reasoning (RAIR) Lab Presentation [8]

are welcome to researchers; but our argument for our dismissal of these esoteric techniques is based on the fact that the processing time resource needed for these techniques can be put to better use on different TM elimination techniques like our proposed predictor or any other halting/non-halting prover, having a higher efficiency.

6.2. Macro Machines

In his 1990 paper, “Attacking the Busy Beaver 5” Heiner Marxen [4] described a tape compression and macro-machine acceleration technique which became fundamental to all further busy beaver search implementations through out the literature. Later on 2004 Alex Holkner [10] has expanded the idea to *k-macro tape representations* and *macro arcs* which are advanced techniques of the same nature.

A macro machine is a higher level Turing Machine which groups blocks of transitions (state, tape and head position) of a lower TM into a single transition. Macro machine simulations are provably faster than the original lower TM simulations, while helping the proof of non-halting. This is actually an implication of linear speedup theorem.

The linear speedup theorem for Turing machines asserts that for any Turing machine solving a problem in time $t(n)$, another machine solving the same problem in time $kt(n)+n+2$ can be constructed for a constant k.

A simple proof for $k = ½$ is as follows :

Assume a Turing Machine M solving a problem in $t(n)$ shifts with its tape alphabet consisting of c symbols and its transition function being defined as a deterministic finite automata with s internal states. We can construct a new machine M' having an alphabet with k^3 symbols where each symbol represents a combination of 3 symbols of machine M. The tape of machine M' is a compact representation of the tape of machine M, where cell i of machine M' represents the group of cells $2i-1$, $2i$ and $2i+1$

of machine *M* with a single symbol of the new alphabet. At each computation step, *M'* simulates the computation of *M* till the head of *M'* leaves the group of cells from left or right (all this simulation can be done in a single step because M can be in no more than sk^3 states without leaving the group of cells or repeating a state which means a loop). Throughout this simulation step *M* may reach the halting state, in which case *M'* also reaches the halting state; or *M* may loop, in which case *M'* does nothing (so also loops). A last refinement is that, as group of cells overlap, every transition between groups in *M* must be converted into *k* transitions between cells in *M'* to take account of the *k* different symbols that might have been written onto the cell belonging to both groups. The construction requires that each computation step in *M'* corresponds to at least 2 computation steps of *M*. Therefore *M'* takes no more than $½t(n)$ steps. By adding delaying steps to *M'*, we can guarantee that it takes exactly $½t(n)$ steps.

This proof can be generalized to all values of $k > 0$. The linear speedup theorem is the rationale behind complexity theory ignoring linear factors and representing the complexity of algorithms with big O notation.

6.3. Tape Compression

A technique similar to the one defined by linear speedup theorem is known as the "tape compression theorem" and allows for a similar constant factor reduction in the space requirements of a Turing machine. This theorem is as follows:

Let's suppose the language *L* is accepted by a deterministic Turing Machine *M* having space complexity *s(n)*, we can show that for any constant $c>0$, *L* is accepted by a deterministic Turing Machine *M'* having space complexity $c.s(n)$.

The same construction from linear speedup theorem is used here. As many symbols from the lower machine is encoded as a single symbol in the macro machine (upper machine) the number of tape cells necessary to represent a string is reduced to a constant fraction of cells required on the lower machine.

Albeit the name “tape compression theorem”, the amount of data necessary to store the macro machine’s tape is not reduced even by a single bit. To the contrary, the amount of bits needed to store the tape of macro machine is statistically a little bit higher, because the symbols at both edges of tape of macro machine may be partially used by lower machine but represented as a full symbol of macro machine alphabet.

Busy beaver problem not only has exponential time complexity but also exponential space complexity which burdens the exploration of the machine space. As tape compression theorem doesn’t improve the space complexity in terms of big O notation another technique is required to be able to simulate candidate machines. A very common compression technique is run length encoding where repeated occurrences of groups of symbols are encoded as a single symbol followed by the number of occurrence of symbol. As long as a string of symbols exposes some sort of repetition of a pattern, this string can be encoded more efficiently by using run length encoding. As like all forms of compression, use of run length encoding is prohibitive in cases where strings are pseudo-random with no statistically significant distribution; where run length encoding results with longer sequences due to encoding overhead.

Most bust beaver candidate Turing machine produce repeating patterns of symbols due to nested loops imposed by state topology, rendering them a perfect candidate for run length encoding the tape. Run length encoding of machine tape can be more efficient when combined with macro machines described above. Long sequences of 1’s or 0’s are rarely encountered; but as macro machine’s symbols correspond to combinations of symbols like “00”,”01”,”10”,”11” alternating strings like “0101010101010101” or complex repetitive patterns of different periodicity can be efficiently encoded too (with 5 bits instead of 16 for that specific example; where first 2 bits describe one of four macro symbol and next 3 bits describe the number of repetition). Optimal block length for run length encoding and macro machine depends on the histogram of blocks present on tape and the mentioned length/periodicity of repeating patterns..

7. Non-halting prediction

Through out the investigation of halting problem specific to Turing Machines, predicting whether a machine will halt or not by mere inspection of the setup of the machine is mostly dismissed because of the inherent complexity underlying in the machine topology that isn't penetrable by simple analysis. As predicting non-halting condition for the complete set of TMs would be equivalent to an oracle[12] for halting problem; given the previously shown proof of undecidability of halting problem this implies that possible predictors have to be incomplete, which means predicting only for a subset of valid TMs and leaving the rest undecided.

Although investigation of the setup of a Turing Machine gives some insight for possible prediction of halting, like disconnected transitions or obviously infinite loops of the starting transition[13], it only contains minimal information which leads to a predictor leaving most machine definitions undecided. Obviously if additional information can be made available to a possible black box halting predictor, this extra information can be used to decrease the number of undecided instances. As we assume that the black box predictor already has complete information about the configuration of a particular instance of Turing Machine and the Busy Beaver problem, the only extra information that can be made available is the simulation history of that machine for a limited number of steps.

In the following sub-sections we will first define our proposed efficiency measure for halting/non-halting predictors, then lay our own predictors features layer by layer from trivial to complex.

---

[12] An oracle is an abstract machine used to examine decision problems. An oracle is able to decide certain decision problems in a single operation, where problem can be of any complexity class including undecidable problems, like the halting problem.

The halting paradox is still valid for such machines. Although they can decide whether particular Turing machine, input tape pairs will halt, they still can't decide whether machines with equivalent halting oracles are halting or not. This observation creates a hierarchy of machines, namely the arithmetical hierarchy, each with a more powerful halting oracle and an even harder halting problem.

[13] Starting transition is the transition taken when a 0 is read by starting state.

7.1. An efficiency measure for halting predictors

It is obvious that any correct halting predictor that benefits from the information of limited simulation history becomes a complete halting predictor as the limit of simulation history size approaches infinity. It can be seen that this last observation is independent of the naivety or the implementation of the predictor; it is solely based on correctness. Given these if we analyze all predictors from a perspective of correctness and completeness, we conclude that completeness is unachievable for predictors but having correctness property it is asymptotically reachable in terms of simulation history.

Based on these observations we propose an efficiency measure for halting and non-halting predictors in terms of the growth rate of ratio of decidable machines to all machines.

M := { All Turing Machines with n-states and k-symbols}

H := { All halting Turing Machines with n-states and k-symbols }

history(m,t) : simulation history of machine m for first t steps

P_t := { (m) | predictor(history(m,t)) $\rightarrow m \in H$ } (or $m \notin H$ for non-halting pred.)

$$\boxed{\textit{efficiency(predictor)} = \int (|P_t| / |M|)\, dt}$$

The explanation of the above expression for efficiency measure of a predictor is as follows. As we have mentioned, any predictor having correctness attribute reaches completeness as the number of simulation speeds available for prediction approaches infinity. We define a predictor as more efficient if it can decide more TMs halting property with less simulation steps available in comparison to other predictors. The ratio of cardinalities of “so far decided” set P_t to “all machines” set *M* gives the efficiency of a predictor at a fixed point in simulation history; when we integrate this ratio in respect to *t* we obtain a measure to correctly compare the speed of coverage of predictors over set *M*.

7.2. "Just observe" halting predictor

For halting Turing machines the number of shifts performed before halting has an exponential distribution over steps taken till halting.

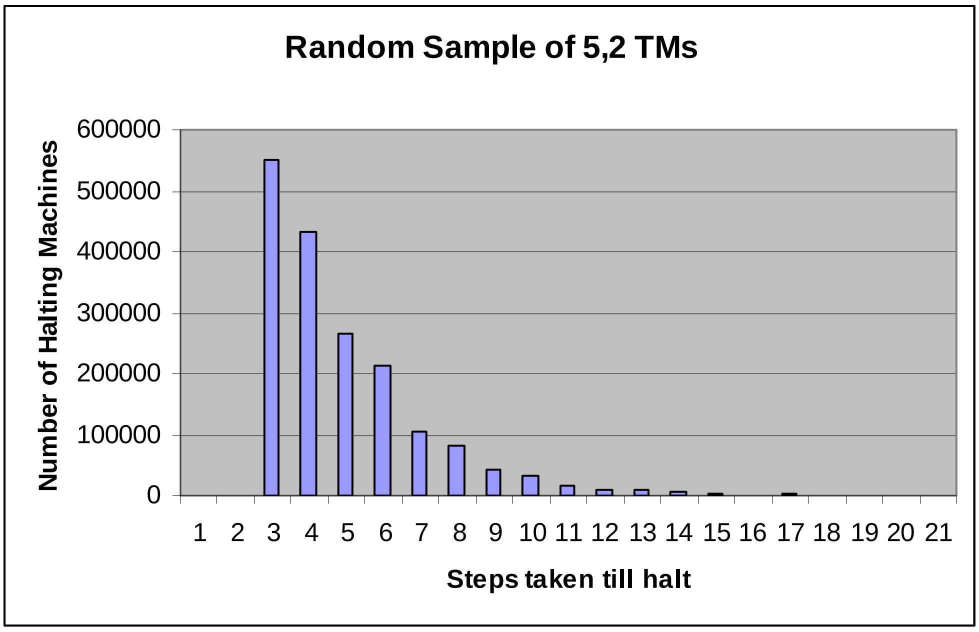

fig. 7.1 – Halting Distribution of 2.10^6 Random Sampled TM

Figure 10 showing the halting histogram is generated from results obtained by our reference C# implementation with a random sampling of a population size of 2.10^6 samples over the space of valid 5-state 2-symbol Turing Machines. The non-linear decrease of halting cases shows that a great number of halting instances use very few steps before halting. Specifically %88.7 of halting Turing Machines do so before their 22nd shift.

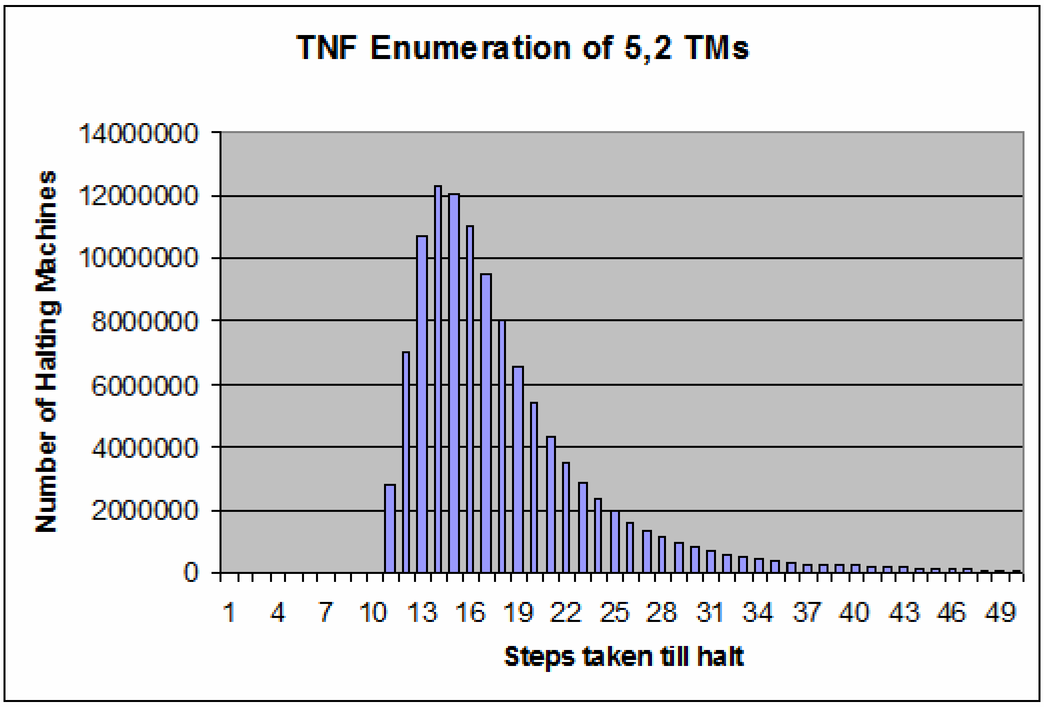

fig. 7.2 – Halting Distribution of all TNF enumerated 5,2 TMs

Figure 11 demonstrates the halting histogram generated from emulations of all 5-state 2-symbol Turing Machines enumerated using TNF and simulated by the same C# code. For TNF enumerated case 111.201.352 machines out of all 671.859.240 enumerated machines halted before their 50th step which makes roughly 16.6% of all machines.

The difference of percentage of early halting machines between the random sampling and the TNF enumeration is the result of effective pruning done by TNF; because when using TNF, not only the obviously non-halting machines are pruned, but also the non-productive and sub-optimal machines are pruned; which explains the drop on the percentage of early halters on TNF enumerated set of machines. As mentioned before, ratio of TNF enumerated machines to all possible machines is in the order of $1{,}06 \times 10^{-5}$ for 5-state and 2-symbol machines, showing the extreme reduction without any loss of information.

7.3. State Connectivity Check

As stated Turing Machine spaces contain many valid TM definitions with no connectivity between the starting state and the anonymous halting state. In the case of the exhaustive enumeration of the TM space (instead of TNF enumeration where complete and optimal connectivity is assured by enumeration layer) a connectivity check between the states let us prove the non-halting of these defective or sub-optimal TMs. A simple recursive procedure can traverse the state transition graph of Turing Machine starting from the halting state, marking each visited state as “visited” and mark the connected states as “to be visited”. This algorithm keeps marking the states till the starting state is reached or no more state is marked as “to be visited” which means a non-halting TM because of the lack of connectivity between start and halt states. This approach is a trivial variant of Dijkstra’s algorithm with no weights assigned to connections.

As all possible TMs are tested with this simple initial test, the code for it is kept as lightweight as possible. To minimize function call overhead due to creation of new stack frames at each recursive call, recursion removal is applied on our reference implementation which replaces the recursive calls with iterative checks over a stack of flags. We begin with only the halting state present at the stack and iterate till either the stack gets empty indicating no more state to check (thus halting state being unreachable from starting state), or the starting state is pushed which indicates the presence of a path between starting and halting state, initial connectivity check returns the respective reachability status. The same can be achieved by beginning with starting state on the stack and iterating while checking for halting state.

```
//returns true if halting state is unreachable by topology
public bool connectivityProblem()
{
    foreach(State st in states) st.flag = 0;
    states[0].flag = 1; //first state to check
    states[MAX_STATE].flag = 5; //look for this


    while(true){

        State stateToCheck = null;

        foreach(State st in states) {    //Find a state marked to be checked
            if (st.flag == 1)
            {
                stateToCheck = st;
                break;
            }
        }

        if (stateToCheck == null) return true;  //no more state marked to check
        else
        {
            stateToCheck.flag = 2;              //mark as checked
            foreach (State nst in stateToCheck.nextState) //all reachable from
            {                                             //this one
                if (nst.flag == 5) return false; //Reached final state
                if (nst.flag == 0) nst.flag = 1;
            }
        }

    }
}
```

listing 6 – C# source of connectivity checking function of TM class

```
function TturingMachine.haltingStateReachable : boolean;

var state_stat : array [0..MAX_STATE] of integer;
    curSta,y,z : integer;

begin
 result := false; //assume unreachability of halting state

 for y := 0 to MAX_STATE-1 do state_stat[y] := 0; //initialize with 0
 state_stat[MAX_STATE] := 1; //start from final state

 repeat

  for curSta := 0 to MAX_STATE do if state_stat[curSta] = 1 then begin //Find marked to checked
   state_stat[curSta] := 2; //mark as processed (2)

   //check for states reachable from current_state
   for y := 0 to MAX_STATE-1 do begin
    for z := 0 to ALPHABET_SIZE-1 do begin

      if states[y].nextStateNo[z] = curSta then begin
       if y = 0 then begin //Reached the starting state -> Connected
        result := true;
        exit;
       end else if state_stat[y] = 0 then state_stat[y] := 1;
      end;

    end;
   end;
   break;
  end;

 until curSta = MAX_STATE+1; //No state is marked with 1

end;
```

listing 7 – Object Pascal source of connectivity checking function of TM class

7.4. Configuration Exhaustion

A basic proof of non-halting for a specific sub-group of Turing Machines is for machines that is stuck to a limited amount of tape during all its simulation or for machines having the amount of tape space explored growing too slowly in respect to shift count. This kind of non-halting can be proven by showing that there are only finitely many different configurations that can happen while we run a Turing Machine on a limited tape area and after that many steps at least one of them must have been repeated; hence we conclude for that particular instance of TM not to halt.

For an *n*-state *m*-symbol Turing Machine the number of possible configurations within a tape range of *k*-blocks can be expressed as $m^k \times n \times k$ which grows exponentially in-terms of tape length. Let's define that expression as a function

$$\boxed{\mathit{max\text{-}config}(n,m,k) = m^k \times n \times k}$$

Consequently, after every *max-config* steps of simulation for a specific (n,m)-TM the length of explored tape space should expand at least by one cell if it didn't already; failure to perform this behavior results with detection of that machine as a non-halting machine due to previously mentioned configuration exhaustion issue.

The inverse of *max-config* function is the measure of minimum tape length required to be explored till a point in order to avoid an infinite loop due to identical configuration. We can call the inverse of *max-config* function as *min-tape* which therefore grows logarithmically in terms of number of simulation steps. We will use this *min-tape* function on the later section "Modeling the Bounds" as the halting lower bound of tape space explored vs. shift count graph.

7.5. Slow Loop Fast Loop

An intriguing non-halting detection technique is described by Heiner Marxen on his article [9] with the proof "left as an exercise". The technique consists of running two independent simulations of the same machine configuration concurrently, specifically a slow and a fast one. For every two simulation steps of the fast simulation a single simulation step for the slow one is executed and configurations of two simulations are compared. If the configurations of both simulation instances are identical this means the machine is in an endless loop. To quote him about his admiration to this ingenious technique: "The fascinating fact is, that we will detect all loops by this procedure. … With this fast/slow approach we need a very limited amount of additional memory, slow down the simulation not even by a factor of 2, and detect the loop before the slow machine works on the second round of the cycle. Yes, there are other methods to do it, but I happen to very much like this technique."

When analyzed this technique of simulation allows us to access "a point in past" of the simulation history of a Turing machine, with ever increasing distance, thus allowing a comparison over a sliding time window with the cost of very little time and space complexity increase. In this context Heiner Marxen [9] defines the term configuration as follows: "A configuration of a TM contains the complete information necessary for its further operation: tape contents, head position and state. While the definition of the TM is needed to make use of a configuration, it is normally not considered to be part of a configuration."

Obviously as the tape contents are compared in their entirety and a loop is detected only if two configurations are identical including the tapes' contents, this methods only detects endless loops stuck within a limited tape area. Therefore this technique is only an improvement over the mentioned configuration exhaustion method which is also stated by Heiner Marxen.

8. Proposed non-halting predictor

Here we propose a new non-halting predictor for the Busy Beaver problem. Our predictor uses the ratio of tape space explored to cycles taken during a point in simulation history as an indicator of information density of tape for that point of history; allowing to predict the unconstructability of a counting process for that many cycles. We propose a special kind of indicator plot for TMs based on previous definition (which we call “shift vs. tape area explored plot”) and show that all halting Busy Beaver Turing Machines must keep explicit track of its temporal position in its simulation history; then we construct a predictor based on that measure to show non-halting, based on inability to track temporal position.

8.1. Ratio of shifts to explored tape space

Visual inspection of the figures 12 and 13 confirms the existence of a non-linear upper and lower bound on length of tape space explored in function of shifts taken for halting machines. For plotting the halting machines’ traces TMs which haven’t halted till a shift limit are discarded, and by re-simulation each simulation step of the halting TMs is plotted as a point on a chart, where horizontal axis is the length of tape space explored and the vertical axis is the number of shifts performed. Each TM has been assigned a unique color therefore the group of plotted points for a TM creates a line showing the trace of mentioned ratio during the course of its simulation. We used our reference C# implementation to plot multiple simulation runs with both randomly sampled and TNF enumerated halting TMs under different parameters which all result with the same characteristic plot.

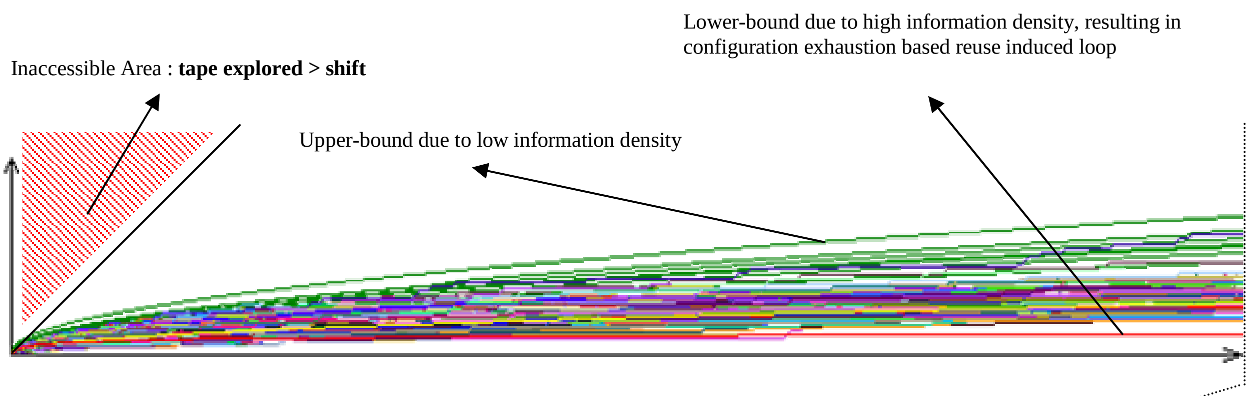

fig. 8.1 – First 600 shift of “shift vs tape area explored” plot for 5000 randomly sampled 5,2 Halting TMs on blank tape; each machine’s trace is marked with a unique color.

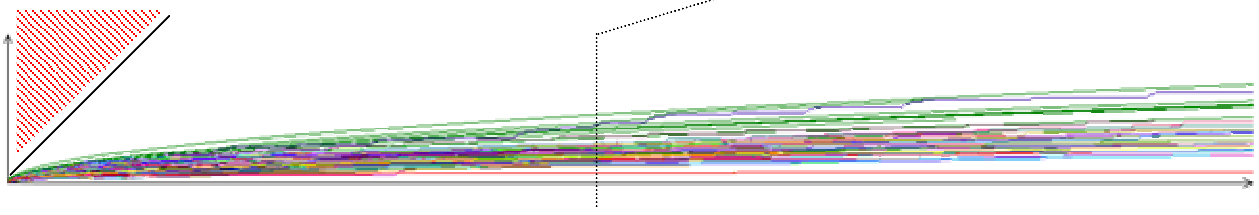

fig. 8.2 – First 1200 steps of the same plot as fig. 8.1

The same plot for non-halting TMs, displayed on figure 14 which is obtained by explicitly discarding the halting machines, results with mostly linear traces having a very different characteristic than the plot of halting TMs. This distinct behavior of halting Turing Machines allow us to build a non-halting predictor, based on detecting machines that cross one of two bounds at any point in their simulation. Luckily as most of the non-halting machines exhibit a linear trace on these plots, most of them cross one of the two bounds (mostly the upper bound) pretty early on their simulation history.

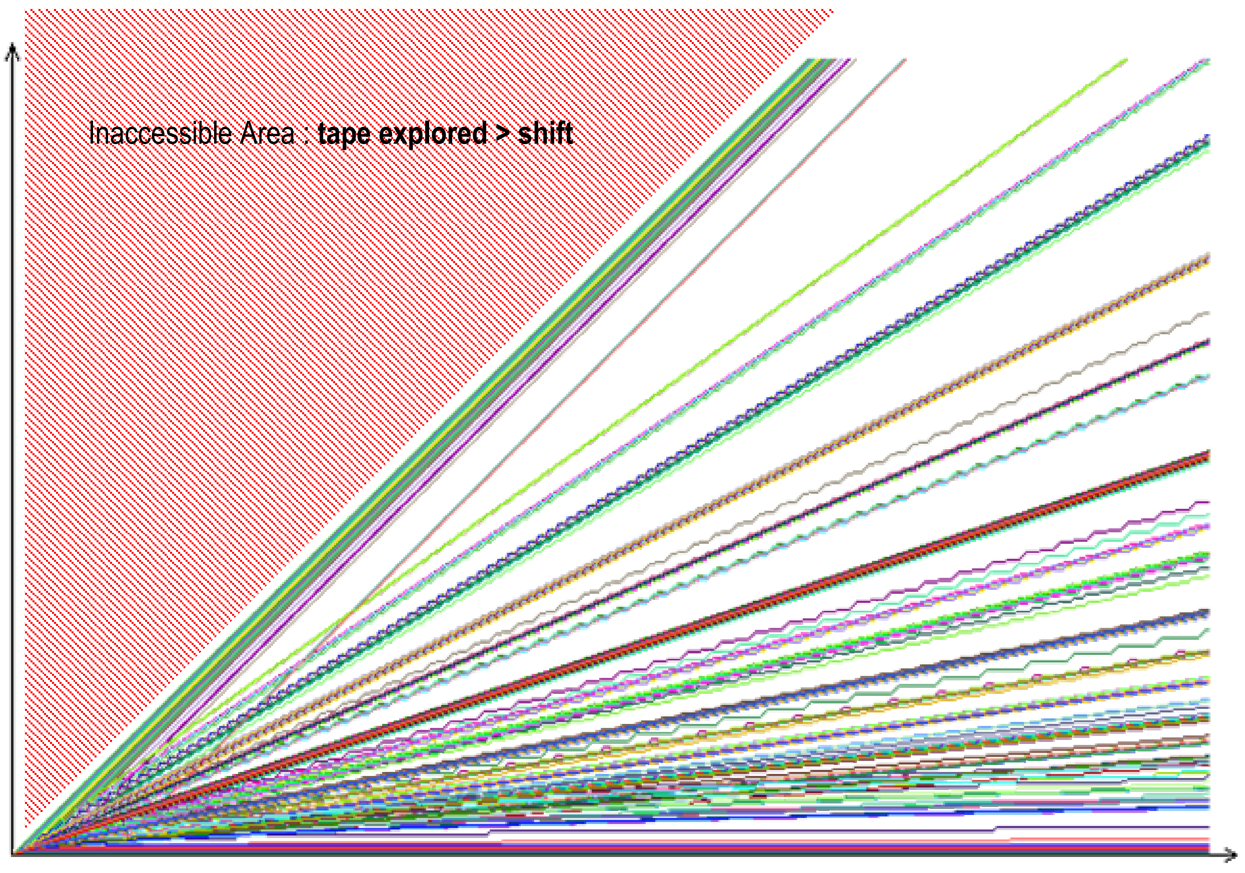

fig. 8.3 – First 600 shifts of "shift vs tape area explored" plot for 10^5 randomly sampled 5,2 TMs not-halting within those shifts; each machine's trace is marked with a unique color.

8.2. Modeling the bounds

As previously observed, halting Turing Machines exhibit a polynomial relation between number of shifts it took and tape length it explored at each point in its simulation history. On the other hand a major proportion of non-halting machines has a linear relation between the number of steps (shifts) to tape length explored.

Lets define *max-tape(n,m,s)* as the function (and as the counter border of *min-tape* defined previously) giving an upper bound of allowable tape space explored in terms of shift count for elements of set of halting TMs using the same conventions as before where n denotes the number of states and m denotes the number of tape alphabet symbols.

$$\boxed{\textit{max-tape}(n,m,t) = (an+bm+c).t^{1/2} = k.t^{1/2}}$$

This function along with *min-tape* function defined on section 7.d respectively provide us mathematical expressions of our proposed lower and upper bound for the area on the “shift vs. tape length explored” plot which a halting TM can not get out of.

8.3. Calculating the coefficients

In order to derive the coefficients *a,b* and *c* required to calculate constant *k* of *max-tape* function, values of *k* should be calculated for many pairs of (n:state count, m:tape alphabet size) which in turn can be used to model a linear relation that gives *k* in terms of *n* and *m*.

The problem with this approach is that calculating the correct value of *k* even for a single pair of *n* and *m* requires complete enumeration of TMs for that *(n,m)* with emulation of each to a reasonably large steps. In our simulations calculating *k* for *(5,2)* case took several hours of computation which resulted in $k_{5,2} \approx 2.78$ which is the coefficient of upperbound curve seen on figure 12. We currently dismissed the calculation of *k* for larger pairs due to limited computational resources at our disposal.

9. Conclusion

Some of the many projects that may extend our work is the calculation of enough coefficients to derive a complete mathematical expression for the upper-bound that can be used on any state count, symbol count pair. Also our predictor can be used as the basis of an attack to acquire new upper bounds for (6,2), (7,2), (4,3), (5,3), (2,5), and (2,6) cases of busy beaver problem. Another possible extension to our predictor would be the linearity analysis of TM traces which doesn't leave the area limited by our bounds; because once a trace is proven to be linearly increasing (in overall), it can be proven that this trace would eventually cross our upper bound at a computable intersection point without expensive emulation till that point.

This work implies a complex relationship between the computational complexity of a system and its spatial or temporal information density; which is exploited as a predictor on our case. This relationship can be further investigated in perspective of computation models, algorithmic complexity and information theory. Also the proposed efficiency measure for predictors allows us to create an efficiency hierarchy of halting prediction methods; further investigation of what makes a halting predictor more efficient would allow us to develop new measurement techniques for computational complexity in terms of algorithmic information theory, but most importantly to obtain a deeper understanding of the behavior of inherently black box instances of different computational models based on short fragments of simulation histories.

References

[1] Rado, T. “On non-computable functions”, *The Bell System Technical Journal*, vol. 41, no. 3, pp. 877-884, (May 1962).

[2] Wood, D. “Theory of Computation”, Harper and Row Publishers. (1987).

[3] Michel, P. “Historical Survey of Busy Beavers”, *http://www.logique.jussieu.fr/~michel/ha.html* (2008).

[4] Marxen, H. Buntrock, J. “Attacking Busy Beaver 5”, *Bulletin of the European Association for Theoretical Computer Science*, Vol 40. (1990).

[5] A. Brady. “The Determination of the Value of Rado’s Noncomputable Function §(*k*) for Four-State Turing Machines”. *Mathematics of Computation*, 40(162):647-665. (1983)

[6] F. Pereira, et al. “Busy Beaver – The Influence of Representation”, *Proceedings of EuroGP'99*, (1999).

[7] B. van Heuveln, et al. “Attacking the Busy Beaver Problem by incorporating the Tree Normalization Method into a Farmer/Worker Scheme”, *Rensselaer AI & Reasoning (RAIR) Lab Internal Proceedings*, (1999).

[8] Rensselaer AI & Reasoning (RAIR) Lab Data, *http://www.cs.rpi.edu/~kelleo/busybeaver/index.html* (January 2004).

[9] Marxen, H. “Macro Machines” *http://www.drb.insel.de/~heiner/BB/macro.html* (December 2004).

[10] Holkner, A. "Acceleration Techniques for Busy Beaver Candidates", *Proceedings of the Second Australian Computing Conference*, (2004).

[11] F. Pereira, et al. “Busy Beaver: An Evolutionary Approach”, *Proceedings of the 2nd Symposium on Artificial Intelligence*,Havana, (1999).

Biographical Sketch

Author is born in Istanbul in 1980. He graduated from Lycée de Galatasaray and got his Bs. in Computer Engineering from Kadir Has University with Honors. He has a 2nd place award on national project competition organized by TUBİTAK where 1st prize isn’t awarded. He currently holds Microsoft Certified Professional (MCP), Microsoft Certified System Engineer (MCSE), and ExpertRating Delphi Expert certifications.