UNIVERSITY
OF BUCHAREST

FACULTY OF
MATHEMATICS AND
COMPUTER SCIENCE

COMPUTER SCIENCE

Bachelor Thesis

REDUCING THE BUSY BEAVER

Graduate
Olăeriu Vlad Mihai

Scientific coordinator
Lect. Dr. Bogdan Dumitru

Bucharest, June 2024

Abstract

The Busy Beaver problem is a pivotal challenge in computability theory, highlighting the extremes of algorithmic behavior. It involves determining the maximum number of steps $S(N)$ and the maximum output generated $\Sigma(N)$ by a N-state halting Turing machine, offering insights to gain a clearer comprehension of what a big number truly is. This thesis emphasizes reducing the possible Turing machines that need to be examined to determine $S(N)$ and $\Sigma(N)$ by defining filtering methods for machines that are not useful in the context of the busy beaver, respectively, that are non-halting. The aim was to build a program in Rust that would execute, analyze, and classify Turing machines efficiently and be easily expandible for further research. The results demonstrated that the number of candidates could be significantly reduced to a much smaller number, but also strengthened the idea that with the increase of N, the N-state Turing machines became more vigorous, requiring more filtering techniques, especially for identifying non-halting machines.

Rezumat

Problema Busy Beaver este o provocare esențială în teoria calculabilității, evidențiind limitele acesteia și a puterii algoritmilor. Această problemă implică determinarea numărului maxim de pași $S(N)$ și a numărului maxim de caractere generat $\Sigma(N)$ de o mașină Turing cu N stări care se oprește, oferind astfel posibilitatea de a înțelege mai profund numerele mari. Această teză pune accentul pe reducerea numărului de mașini Turing care trebuie examinate pentru determinarea $S(N)$ și a $\Sigma(N)$ prin definirea unor metode de filtrare pentru acele mașini care nu sunt utile în contextul problemei busy beaver, respectiv filtre pentru mașinile care nu se vor opri niciodată. Scopul a fost de a construi un program în Rust care să execute, să analizeze, iar apoi să clasifice mașinile Turing în mod eficient și să fie ușor de extins pentru cercetări ulterioare. Rezultatele au demonstrat că numărul de candidați poate fi redus semnificativ la un număr mult mai mic, dar au întărit și ideea că, odată cu creșterea lui N, mașinile Turing cu N stări devin mai viguroase, necesitând mai multe tehnici de filtrare, în special pentru identificarea mașinilor care nu se vor opri.

Contents

Chapter 1

Introduction

Over the years, computer power has significantly increased, with integrated circuits gaining more and more transistors, according to Moore's law. Having such robust computers should make us ask ourselves what computational boundaries we have, or if a possibility of solving the undecided problems will ever arise. Luckily, these frontiers were established, firstly with the appearance of the Turing machine, followed by a variant with an even more intrinsic value, the busy beaver, that will be further denoted by BB.

Originally, a busy beaver of size N, $BB(N)$, is the halting Turing Machine with N states, with an alphabet tape consisting of symbols $\{0, 1\}$, that writes the most numbers of 1s on the tape starting from a tape that only contains 0s all over it. It can be viewed as a software program that, given a fixed input size, N, and a code of conduct, its transition function, will attempt to produce as much output as possible. Knowing that the busy beaver function runs in S numbers of steps, it is a certainty that any software program, which was translated to a Turing machine, once surpassing the S numbers of steps while executing, will never stop. Nevertheless, this makes busy Beaver the computational boundary we were looking for.

Unfortunately, the busy beaver is a non-computable function. On the ground that with a bigger number of states N, the $BB(N)$'s value grows faster than the value of any computable function, faster than the polynomial, exponential, factorial, and even the Ackermann function, it becomes almost out of the question to find the N-state halting Turing machine that produces this value. Also with the increase of N, more complex Turing machines would need to be tested, and it will be almost impossible to differentiate a non-halting Turing machine from a machine that takes a long time to run. But, thankfully, for small N values, the busy beaver values can be computed.

This thesis, entitled "Reducing the busy beaver", aims at finding methods of reducing the space of busy beaver candidates for any given N, with the thought of being followed by a computational solution to the problem: running the remaining candidates to find out the value for $BB(N)$.

1.1 Background

Intrigued by an unsolved problem in mathematics, precisely the Entscheidungsproblem (decision problem) [1] proposed by David Hilbert and Wilhelm Ackermann, Alan Turing introduced the concept of a theoretical machine that was capable of performing computations [16]. The decision problem asked that, given a mathematical statement, is there any algorithm that could determine the truth or falsity of the statement. With its machine, Turing was able to solve this problem. Moreover, he formalized the notion of computation and algorithm, and a version of his machine, the universal Turing machine, was able to compute all computable functions.

There was yet to be discovered any other formalized machine to have more computational power than a Turing machine. Any model that offered unrestricted access to unlimited memory and unlimited time of execution was proven to be equivalent to the Turing machine. This observation resulted in an essential corollary, that different computational models, even with some of them being faster than others, or using fewer resources, would describe the same class of functions [11].

As powerful as they seem, there were still functions that could not be computed by a Turing machine. This class of functions was called the class of non-computable functions, or well-defined functions, with their meaning and definition still being vague [8]. One function belonging to this class concerns the Halting problem, which is the problem of finding if a given program will halt or if it will enter an infinite loop, given a certain input. Another function from the non-computable functions class is the busy beaver function, defined by Tibor Radó [8].

The busy beaver function seeks to identify a halting Turing machine that generates the maximum possible output. With the tape alphabet being $\{0, 1\}$, the tape being infinite, and initially having 0s on every cell, the Turing machine needed to write as many 1s as possible on the tape, using a given number of states. The busy beaver with N states was the one that produced the most output given N states. The objective of defining the busy beaver function was to showcase some simple non-computable functions, to get an insight into their fast growth, and of the whole class of similar functions.

1.2 Research problem

Despite its non-computable nature, the N-state busy beaver values for a few small Ns were computed. For the 1-state busy beaver, the maximum output produced was 1 for obvious reasons. Given only one state besides the halting state, the only valid option for the Turing machine was to write a 1 on the tape, and then halt. Otherwise, it would have entered in an infinite loop. Starting with the 2-state busy beaver, the search for the

[1] https://en.wikipedia.org/wiki/Entscheidungsproblem

best-performing Turing machine became more complicated.

The transition function of a Turing machine was defined as $\delta : (Q \times \Gamma) \rightarrow (Q' \times \Gamma \times \{L, R\})$, where Q is the set of states, $Q = Q \cup q_{halt}$, Γ is the tape alphabet and $\{L, R\}$ are the directions in which the machine can move. Thus, for an arbitrary N, the total number of possible Turing machine candidates is equal to $[4(N+1)]^{2N}$. The growth of N implied even a faster growth in the number of candidates.

In addition to the immense number of possible Turing machines, certainly among all the contenders were also non-halting machines. The problem of finding the machine that generates the most output was reduced to identifying non-halting Turing machines, excluding them from the machines that needed to be analyzed, and then running the rest of the machines.

Not only non-halting Turing machines needed to be discovered but also the useless ones in the context of a Busy Beaver. The futile machines were the ones that did not produce any output, to only write 0s on the tape, or the machines that replicated the behavior of smaller N-state busy beavers. The aim was to group the Turing machines into families of non-halting machines that had the same paternal behavior.

The research problem draws attention to the difficulty of labeling a Turing machine as non-halting. For some families, it was possible to catalog them as non-halting by only looking at their transition function. If a machine contained a transition such as $(q_0, 0) \rightarrow (q_0, 1, R)$, where q_0 would have been the starting state, it was clearly that it entered in an infinite loop right away. The real difficulty came for the Turing machines that needed to be run to be labeled as non-halting, the ones for which the behavior of the machine's head needed to be analyzed.

Besides searching for filtering methods of Turing machines, this paper also emphasized a research area focused on constructing a software system capable of maximizing resource utilization in this scope. It explores procedures for optimizing the execution of Turing machines through parallelization, ensuring efficient operation. Additionally, it addresses the challenge of simulating machines with unrestricted tape length and runtime.

In the end, this research pursued the objective of finding filtering methods to reduce the number of relevant Turing machines needed to run, to calculate the smaller N-state busy beavers, and also provide a software implementation for these filters that is easily expandable for further research.

1.3 Motivation

The driving force behind this research stems from two principal reasons: the theoretical possibility of resolving mathematical conjectures knowing the values for the busy beaver function, and getting a glimpse of the big numbers. The outcomes of knowing busy beaver values could help in research development in other areas of mathematics as well.

The busy beaver function has a fascinating property, it grows faster than any other computable function. Any algorithm, simulated on a Turing machine, can be tested to find whether it will halt or not using the busy beaver $S()$ function. Considering that the translation of the algorithm to a Turing machine has M states, all that is needed to be done is to run the algorithm and make a comparison. In the case of not halting before the number of steps taken by $S(M)$ to halt, it will never halt. Otherwise, respecting this boundary, the execution will result in a halt. An algorithm that searches for counterexamples within the definition of a mathematical conjecture and halts before the exact maximum number of steps allowed by the $S(M)$ will disprove the conjecture. Alternatively, it will run forever, proving that the conjecture is true.

Notwithstanding the burden of simulating mathematical conjectures on a Turing machine, researchers have done it, and using not so many states. An anonymous GitHub user explicitly defined a 27-state Turing machine that halts if Goldbach's conjecture is false [1]. Another example of a simulated conjecture is the Riemann Hypothesis, which was translated to a 744-state Turing machine that halts given that the conjecture is false [1]. If BB(27) and BB(744) were to be known, the associated Turing machines for the two conjectures should be left to run the exact amount of steps that were required by the beavers to halt, and then to be analyzed. Unfortunately, this scenario is far from becoming reality.

The other aspect standing behind the motivation for this research was the big numbers. In the time of the ancient Greeks, it was a common belief that the number of grains of sand was uncountable until Archimedes set an upper bound for this quantity. To express this boundary, which was equal to 10^{63}, he used the exponentials. At that time, and even now, this number was beyond the people's power of visualization, but an interest in understanding and comprehending their magnitude existed since that moment [2].

In contemporary times, in the 1920s, the Ackermann function made its appearance, which was the first computable, non-recursive primitive function. Its growth is extraordinarily fast, illustrating the limits of the primitive recursive functions and concerning a need for a better description of recursive functions.

Finally, yet importantly, the $BB(N)$ function, which is a non-computable function with computable values for small Ns. Considering that this function grows faster than any function from the previously mentioned classes of functions, knowing a part of its values could result in understanding problems of the same magnitude, including predicting the weather or understanding different metrics in the universe: light years, the lifespan of the stars, or even the mass of the planets.

This research thesis was backed up by the desire to better understand the significance of big numbers and to explore the computational boundaries that are set for today's machines. With success in this area of research, maybe better definitions of computability would arise and fulfill the lack of intuition towards non-computable functions.

Chapter 2

Related work

In the following chapter, related work in finding busy beavers will be presented, more precisely how were the Turing machines filtered, what non-halting families were found, and the methods used for running the machines of interest.

2.1 Radó and Lin's first steps

To begin with, the $BB(1)$ value is trivial and equal to 1. To find $BB(2)$, a part of the Turing machine was filtered out and the rest was computed, and the value found was 4. Starting with the $BB(3)$, the computations became more difficult. The conjecture was that $BB(3) = 6$, and the Turing machine for $S(N)$ halted in 21 steps. This conjecture was proved to be true by Radó and Lin [9].

Their work started with the observation that is not efficient to run all the possible Turing machines, which was equal to a total of $[4(3+1)]^6$ machines, or nearly 17 million machines. This number was reduced to 82,944 machines which were divided into four categories. These categories were created by only looking at the transition function of the machines, a static analysis. Afterward, the normalized machines were run for exactly 21 steps and the output for the ones that halted within that number of steps was saved. With this approach, only 40 machines remained non-halting after 21 steps of execution, which were called holdouts [9]. The remaining holdouts were analyzed by hand by the two scientists, who tried to identify non-halting patterns of the machines.

To construct the four categories, a few observations were made related to the transitions in the transition functions of the machines. One remark stated that for a Turing machine denoted as M, the Turing machine obtained by interchanging all left movements with the right movements, denoted M', is equivalent to it in the context of the busy beaver, such as $BB(M) = BB(M')$ and they halt in the same number of steps. As a result, from that point on only the machines that had a transition $(q_0, 0) \rightarrow (q_i, \text{symbol}, R), \forall q_i \in Q, \forall \text{symbol} \in \Gamma$ were considered. Other observations excluded the machines that were not producing any output at all or the ones that were going from

the starting state straight into the halting state because this behavior could not produce more output than $BB(1)$. The categories were equally divided into a total number of 20,736 Turing machines.

The analysis of the holdouts was done by looking at how the tape of a Turing machine was modified. The two big classes of non-halting Turing machines were called total recurrence and partial recurrence [9]. The total recurrence occurred when the head of the machine reached $cell_m$ after m steps and $cell_n$ after n steps, with identical tape instances relative to their position. This was clear because if the Turing machine behaved in a certain way in the first m steps, if it got to step n having the same configuration as the one when it reached $cell_m$, then it would repeat its behavior. It is important to note that by *configuration* it means the same tape values and the same state in which the cells were visited.

Partial recurrence is a type of recurrence that appears only in the left or the right direction of the tape. First let's consider that $cell_n$ was to the right of $cell_m$, and $cell_s$ was the leftmost cell modified by the head of the machine when moving from $cell_n$ to $cell_m$. The distance between $cell_m$ and $cell_s$ was denoted with k, and the cell positioned to the left by $k+1$ spaces was the left barrier of $cell_m$. For $cell_n$, the left barrier was the cell positioned $k+1$ spaces to the left of $cell_n$. If the tape values between the $cell_m$ and its left barrier were the same as the tape values between $cell_n$ and its left barrier, a partial recurrence occurred. The same principle applied when $cell_m$ was positioned to the left of the cell $cell_n$, but this time they had right barriers.

Henceforth, by running Turing machines for 21 steps and by classifying the holdouts into different classes of non-halting Turing machines, Radó and Lin were able to prove that the conjecture was right. The approach of reducing the possible candidates into smaller categories and the methods used for detecting the never-stopping machines stand as a basement for descendants of research that approaches this problem.

2.2 Complex behaviour of simple machines

In their seminal paper, Rona Machlin and Quentin Stout demonstrated the power of simple machines [5] through the lens of the busy beaver problem, emphasizing the broader implications of trying to classify a class of straightforward programs. The belief was that the failure brought by such classification of the programs belonging to a sufficiently strong class should be recognized better, this failure resulting from the search for the N-state busy beaver for a sufficiently big N. They also mentioned the theoretical possibility of reducing mathematical problems to halting problems, which would solve many mathematical conjectures if upper bound or concrete values were known for the busy beaver. Following up, the methodology for solving the $BB(3)$ and $BB(4)$ conjectures was described.

In this research paper, the reduction techniques used by Radó and Lin were extended to the point where transitions were added to a transition function as needed, which resulted in a more complex static analysis of the Turing machines. This method was called tree normalization [5] and the name came from the illustration of the transitions that were chosen as needed. For example, for a first transition $(q_0, 0) \rightarrow (q_2, 1, R)$, the machine will go to state 2 and will encounter a 0, so the following transition must be of the form $(q_2, 0) \rightarrow (q_i, symbol, \{L, R\}), \forall q_i \in Q, \forall \text{symbol} \in \Gamma$. With this approach, recognizing non-halting Turing machines by only looking at their transition function would be less difficult.

After this reduction and the allowance of running the machines for a couple of hundreds of steps, there were 2572 holdouts left for $BB(3)$ and 421.108 holdouts for $BB(4)$. The following step was to find non-halting Turing machine classes, to give a formal mathematical method for their classification, and to implement a program that automatically detected them. The classes described in the paper were simple loops, backtracking, Christmas trees, and counters.

To identify a simple loop, the tape configuration of the Turing machine should repeat its previous values in one of two ways. The first one, in which a configuration of the type XY was seen at some step during the execution, where X and Y were sequences of 0s and 1s, was repeated later on in the form of 0^*XY0^*, where $0*$ represents a non-empty sequence of 0s. The other situation was when the configuration 0^*XY0^* was followed by the 0^*VXY0^* configuration, where V was also a sequence of 0s and 1s.

Steps									
0					$\tilde{0}$				
1					1	$\tilde{0}$			
2					$\tilde{1}$	1			
3				$\tilde{0}$	1	1			
4				1	$\tilde{1}$	1			
5				1	1	$\tilde{1}$			
6				1	1	1	$\tilde{0}$		
7				1	1	$\tilde{1}$	1		
8				1	$\tilde{1}$	1	1		
9				$\tilde{1}$	1	1	1		
10			$\tilde{0}$	1	1	1	1		

Table 2.1: A Christmas tree [5]

Another method applied was related to the total, respectively partial recurrence patterns found by Radó and Lin when analyzing the 40 holdouts for $BB(3)$. The focus was moved towards the back-and-forth movement of the head of the Turing machine [5], behavior which was called a Christmas tree. Multiple variants of the Christmas tree were used to filter out the holdouts, some of the variations being the alternating Christmas tree

or the shadow Christmas tree [5]. The formal method of identifying the Christmas trees was quite sophisticated, so to have a clearer view of how a Turing machine that behaved in this way was identified, an example of such a tree is displayed in Table 2.1. It can be observed the back and forth movement of the head, and the tree structure given by the tape values.

One unique method of finding whether a machine will halt or not was to start from the halting state and move back until reaching the starting state, or a dead end. This method was called backtracking. In its description, an important detail was added, that it should be used only for a determined period of time because the backtracking itself could go into an infinite loop. By proving that the Turing machine could not reach the starting state from the halting state, it implied that it would never halt, because the halting state was unreachable as well.

The last class of non-halting Turing machines was called the counters class, in which every Turing machine acted as a type of binary counter [5]. Similarly with the Christmas trees, the formal definition of a counter was complicated, but in Table 2.2 the behavior of a counter can be analyzed.

Steps									
0					0				
1					1	$\tilde{0}$			
2					$\tilde{1}$	1			
3				$\tilde{0}$	0	1			
4					$\tilde{0}$	1			
5					1	$\tilde{1}$			
6					1	1	$\tilde{0}$		
7					1	1	1	$\tilde{0}$	
8					1	1	$\tilde{1}$	1	
9					1	$\tilde{1}$	0	1	
10					$\tilde{1}$	0	0	1	
11				$\tilde{0}$	0	0	0	1	
12					$\tilde{0}$	0	0	1	

Table 2.2: A counter [5]

After applying identifying all the Turing machines that belonged to one of the classes mentioned above, the number of holdouts for the $BB(3)$ was reduced from 2572 to 0 holdouts, respectively for the $BB(4)$ 182,604 to 210 holdouts. The rest of the machines were analyzed by hand, and the majority resulted in being variations of the binary counter, base-3 and base-4 counters. The most prolific filter involved detecting the simple loops, which reduced the number of holdouts by 96%.

2.3 Building the zany zoo

More recent developments, particularly the paper *Generating Candidate Busy Beaver Machines (Or How to Build the Zany Zoo)* [4] published by James Harland, questioned the formality of the filtering methods applied to non-halting Turing machines, prompting mathematically detailed methods to generate relevant candidates for the busy beaver. The focus was towards exploiting the relation between the states $q_0, q_1, ..., q_n$ for the N-state busy beaver and towards the fact that the analyzing of all of the machines needed to be done on the same input, a blank full of 0s.

Besides the standard definition of the Turing machine used, being a quadruple $(Q \cup q_{halt}, \Gamma, \delta, q_{start})$ [4], where Q is the set of normal states, Γ is the tape alphabet consisting of $\{0, 1\}$ and δ is the transition function, some more properties were defined:

- k-halting: contains k transitions of the form $(-,-) \rightarrow (-,-,q_{halt})$
- exhaustive: transition function is a total function
- n-state full: $|Q| = n$
- m-symbol full: $|\Gamma| = m$
- activity: the number of steps until until halting, or ∞ otherwise
- productivity: the number of non-blank characters on the tape after halting, or undefined otherwise

Firstly, a N-state Turing machine M was considered to be irrelevant in the context of the busy beaver if it had $activity(M) = \infty$, $activity(M) <= N$ or $productivity(M) = 0$. These were obvious cases, in which the machine M was non-halting, running for a small number of steps, or producing no output at all. A Turing machine that had the transition $(q_0, 0) \rightarrow (-,-,q_0)$ had the activity equal to ∞ and was not considered for the busy beaver. Another machine that contained the transition $(q_0, 0) \rightarrow (-,-,q_{halt})$ was also irrelevant because it went straight into the halting state and its activity was equal to 1.

Continuing with this, the equivalence between Turing machines was defined by means of a few lemmas. One of them stated that for two states $q_i, q_j \in Q \quad \{q_{start}, q_{halt}\}$ of a Turing machine M, by building another machine M' in which every occurrence of q_i is interchanged with q_j, that new machine is productivity and activity equivalent to M and also contains the same configuration. Similarly, for two symbols $symbol_i, symbol_j \in \Gamma, symbol_i \neq 0, symbol_j \neq 0$ of a Turing machine M, by constructing a new machine in which all appearances of each symbol are interchanged with the other one, an productivity and activity equivalent machine M' will result.

With the provided lemmas and definitions, a clear definition of a normal Turing machine was formulated. Importantly, it was demonstrated that any Turing machine relevant to the busy beaver problem is productivity equivalent to a normal Turing machine. This clear definition of relevance paved the way for developing the algorithm to generate candidate machines, more precisely, to generate only normal Turing machines.

2.4 About bbchallenge

The bbchallenge [1] is an open source project that targets to prove or disprove the conjecture posed firstly by Marxen and Buntrock [6], and later also supported by Scott Aaronson [1] that the $BB(5) = 47,176,870$ by providing a computational solution to the problem, backed by mathematical proofs of the correctitude of the filtering methods used. Their approach is identical to the one taken by Radó and Lin [9] in finding the truth about $BB(3) = 3$. After defining and applying their filter, the team of bbchallenge was left with 88,664,064 holdouts for $BB(5)$, out of which only $2,833$ remained to be run.

The main focus of the project was to find deciders for Turing machines. A decider is a program that will identify the nature of a Turing machine, halting or non-halting. They grouped Turing machines in families with similar behavior, which they called the *zoology* [15], and based on these families, the deciders were built. Currently, a total number of 6 deciders was built, the most recent one being published during the time this thesis was written, the bouncer's decider.

To describe a few of them, the cycler decider analyzes the history of the computation of a Turing machine, which contains the tape value, the position of the head, and the current state. If the same configuration happened to appear twice during the execution of a Turing machine, it means that its behavior would be repeated, thus resulting in it never halting. Another decider, the translated cycler, is very similar to the partial recurrence defined by Rado and Lin previously [9] and describes the Turing machines that are constantly shifting the same tape values to the left or the right of the tape. This decider was the inspiration for a filtering method used in this paper, which was described in the subsection 4.4.4.

The code base of the project was written in the Go language. Alongside the deciders that were implemented, formal mathematical proofs of the deciders were also constructed. The project is very strict regarding the quality of the code involved and the correctness of the proofs, encouraging the use of automatic proving tools such as Lean or Coq. In the latest updates on the project, which also introduced the bouncers decider previously mentioned, a system of formal verification of the certificates for the bouncers was added, written in Coq. Alongside the reduction from millions of holdouts to only $2,8333$, this formalization puts the research team a step closer to finding the truth about $BB(5)$.

[1] More details about the bbchallenge project: https://bbchallenge.org/story

Chapter 3

Preliminary

3.1 Turing Machine

A Turing machine is a theoretical model of computation, defined by Alan Turing in 1936 in his thesis *"On computable numbers, with an application to the Entscheidungsproblem"* [16], capable of simulating any algorithm that a modern computer can simulate only by manipulating symbols on an infinite tape, for an unlimited amount of time. Moreover, it can simulate all the computable functions. The initial aim of Turing's thesis was to make an association between the computable numbers and the "a-machine" to disprove the Entscheidungsproblem, but ended up defining the standard model of computability.

Despite common beliefs that Turing identified the computability powers of his machine with one of modern computers, which is not historically correct, he tried to formalize the human capabilities of computation. He associated computation with "writing certain symbols on paper", and its machine with "normal" models of computation [7]. The idea of computability is in strong relation with the class of computable numbers, but do computable numbers include all the numbers that should be naturally considered computable? As stated in his paper, the arguments to support this statement are intuitive [16]. But until the current moment, there was not a single computable algorithm that could not be Turing computable, thus the Turing machine model still stands as the foundational model.

The formal definition of a Turing machine that will be used throughout of this paper is the one given by Sipser in his famous publication, *"Introduction to the Theory of Computation"* [12]. A Turing machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, Σ, Γ are all finite sets and:

- Q is the set of states,
- Σ is the input alphabet not containing the blank symbol _,
- Γ is the tape alphabet, where $\_ \in \Gamma$ and $\Sigma \subseteq \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,

- $q_0 \in Q$ is the start state,
- $q_{accept} \in Q$ is the accept state, and $q_{reject} \in Q$ the reject state, $q_{reject} \neq q_{accept}$

The machine operates on an infinite tape that initially contains only the input string and is full of blank symbols in the rest of the cells. It has a head with which it can read and write symbols on the tape and move around it, thus giving unrestricted access to the tape. The movements need to be done based on the definition of δ. Table 3.1 can be visualized as a configuration for a Turing machine that is currently in state q_i, and the tape's head is positioned on the third 0 from left to right.

...	0	1	0	1	1	q_i	0	0	1	1	...

Table 3.1: Turing machine with configuration $01011q_i0011$

At first, the state of the machine is q_0, and the first move of the head will be determined by $(q_0, 0) \rightarrow (q_i, symbol, direction), \forall q_i \in Q, \forall symbol \in \Gamma, \forall direction \in \{L, R\}$ in the transition function. This transition says that the machine will write the *symbol* in the current cell where the head is pointing, then it will move in the pointed *direction*, and finally, its state will be changed to q_i. For a better intuition, the states can be viewed as a "state of mind" of the Turing machine, dictating its behavior. Similar to finite automata, the Turing machine can decide whether a string is part of a language.

A Turing machine can end up in three situations: accepting, rejecting, or never halting. By halting, it means that it would stop in an accepting or rejecting state, so the first two situations are complementary to the last one. During the running of the Turing machine, if the current state becomes q_{accept} or q_{reject}, the machine will stop. Accepting means that the input belongs to the language described by the δ function, and conversely rejecting means that it does not.

For a language A, if all the strings belonging to it are used as input strings for a Turing machine M_1, ending up in the q_{accept} state, it means that the language A is recognizable by machine M_1. For the strings that do not belong to language A, machine M_1 will either enter in the q_{reject} state or it will never halt.

On the other hand, if a Turing machine M_2 can recognize all the inputs that are included in language A and also reject the ones that are not, this means that the language A is decided by machine M_2. Machines similar to M_2, for any other languages, are called deciders because they will halt every time, whether in q_{accept} or q_{reject}.

One of the most important theorems in the theory of computation, with a profound philosophical meaning, is the one that states that there are languages undecidable by a Turing machine, or more informally, there are problems that cannot be solved by a computer.

The first language to be proved to be undecidable is A_{TM}, where A_{TM} is the language consisting of all pairs $< M, w >$ with M being a Turing machine that accepts the input w.

Despite being undecidable, A_{TM} is Turing-recognizable, which means that the recognizers are more powerful than the deciders. The proof of this theorem was done by demonstrating that there are an uncountable number of languages, but a countable number of Turing machines, consequently a mapping between the two sets not being able to exist.

To better understand the power of an undecidable language, imagine that even some Turing computable functions might not be possible to be solved using modern computers, because they would require an impractical amount of time and memory to run. On the consideration that a physical computer is a materialization of a Turing machine, an undecidable algorithm that no computer can be built to run it represents a problem that is hard to ever be solved [7]. At first, this statement might seem odd since computational efficiency improves each year, with increasingly powerful physical computers being built. However, the key point is efficiency. For example, a quantum computer shares the same computational capabilities as an electronic computer but operates much faster. In terms of what can be computed, quantum computers, electronic computers, and all physical computers can perform the same computations as a Turing machine; the difference lies in their efficiency.

3.2 Halting problem

Another undecidable problem, which is in strong relation with A_{TM}, is the halting problem. It is the problem of finding out if a Turing machine will halt or not for a given input string or informally, if a computer program will finish or it will enter an infinite loop for a certain input.

The formal definition of the language is HALT = {<M, w>, M is a Turing machine that halts on input w}. The proof of the undecidability of HALT was done by contradiction by supposing HALT is a decidable language, it will result that A_{TM} is also decidable, which is false because a previously mentioned theorem states the undecidability of A_{TM}. One version of HALT is $HALT_E$ = { M, M is a Turing machine that halts on the empty input string}, which is also undecidable. This version of the halting problem is related to the busy beaver, a relation which will be analyzed in a further section of this paper.

Every programmer struggled with an infinite loop at least once in their career, and for sure they wished there was a tool to tell them if the input to their program would result in a loop or if it would finish. Unfortunately, this tool does not exist, because such a tool would decide the halting problem.

Scott Aaronson made a supposition in one of his articles [2] that such a tool existed, in which he imagined a super Turing machine that was able to solve the halting problem of the ordinary Turing machine. Now, the halting problem of a super Turing machine arises. Its solution is similar to the solution to the halting problem of the ordinary Turing machine, in which a more powerful class of machines needs to exist to solve the halting

problem for the super Turing machine. This would result in a hierarchy of classes of machines, one more powerful than the other. The main problem is that machines of the same classes cannot analyze themselves, a more powerful machine would be needed to solve the halting problem for the current class. As Scott Aaronson ingeniously said "to understand reality, we need to go outside of the reality", [2] the same principle applies to the Turing machine, to be able to tell if it will halt or not, a more powerful machine is needed, that exceeds our concept of computation.

As noted previously, the halting problem has immersive implications in mathematical conjectures, with all conjectures being able to be reduced to the halting problem. Because of its strictness, the halting problem does not provide any insight into how close we could be to this reduction, and it does not give any computational bounds for comparison.

The first one to observe the strictness of the undecidability provided by the halting problem, and also the lack of intuition towards its boundaries was Tibor Radó, who introduced the busy beaver function to provide solutions for these inconveniences.

3.3 Non-computable functions

A non-computable function, also known as a non-recursive function, is a function for which no Turing machine is capable of computing the values for every possible input of that function. Because such functions are not Turing-computable, it also means that no computer can calculate their values. One example of these functions is the busy beaver function.

In 1963, Tibor Radó introduced the busy beaver problem with the thought of strengthening the idea that a class of functions more powerful than the class of computable, general recursive, functions exists and that in function in this class should be well-defined in some manner [8]. The problem introduced aims at finding the halting Turing machines that produce the most output, given N number of possible states, besides the *halting* state. As noted earlier, the target Turing machine needs to use the tape alphabet $\Gamma = \{0, 1\}$, its head to move in the directions $\{L, R\}$ and to use N states. This busy beaver function is usually denoted with $\Sigma(N)$, conversely in this thesis, denoted with $BB(N)$ and is also the busy beaver function for which the filters were built.

Another non-computable function introduced by Radó is the maximum shifts function $S(N)$, which aimed at finding the N-state busy beaver that executed the most number of shifts, also called steps, before halting. This busy beaver function is the one that holds the boundaries of computation, thus finding the values for $S(N)$ is as difficult as finding values for $\Sigma(N)$, and the same filters can be applied in both instances of the busy beaver problem. In this research, the focus is shifted towards $\Sigma(N)$, with explicit application also in the case of $S(N)$.

For small values, the N-state busy beaver was computed, and among the values that

were computed, one of them was proved by the Radó, $BB(3)$. Until the current moment, the biggest busy beaver known is $BB(4)$, which is equal to 13 and halted in 107 steps. For $BB(5)$ a conjecture exists in which the Turing machines that produced the most output with 5 states halted in 47,176,870 steps [6].

Several confusions can appear related to the busy beaver function. It is certainly that the Turing machine with N states that produces the most output, that writes the most 1s on an initially empty tape exists, thus the $BB(N)$ number is computable. The uncomputability of the problem is in finding this Turing machine, among all the possible candidates. The number of Turing machines that need to be tested increases extremely fast, and not to mention that the number of non-halting Turing machines increases with the same speed. The manual analysis of Turing machines that halt after millions of steps is impossible, therefore a computing program would be needed to do this analysis automatically, and the methods used for detecting non-halting Turing machines should be mathematically proven to be correct.

For every Turing machine that would be considered the winner of the N-state busy beaver it should also be demonstrated that there is no other Turing machine that performs better. The usual procedure [5] [9] to demonstrate a conjecture of this type is to run all the possible Turing machine candidates for the number of steps that took the $S(N)$ winner to halt. The score of the machines that halted within that number of steps would be compared to the score of the $BB(N)$ winner, and for the rest, it should be proved that they will never halt.

Even though the approach of finding the N-state busy beaver is difficult, compared to other non-computable functions, such as the ones that involve the halting problem, it offers a more concrete picture of the power it has. The following chapter will focus on providing filtering methods for Turing machine candidates for the busy beaver function.

Chapter 4

Filtering

This chapter describes the core of the current research and the methods used for filtering busy beaver candidates. The procedures are described in detail, assisted by their mathematical definition, and separated into three categories: generation, compile, and runtime filters.

4.1 Importance

Firstly, it should be mentioned that the calculations in this section were done on the original definition of the busy beaver [8], which described it as a Turing machine with N operational states plus the halt state, an input alphabet equal to the tape alphabet consisting of $[0, 1]$, and the possible directions of the head, left and right.

The transition function was defined as $\delta : Q \times \Gamma \to Q' \times \Gamma \times \{L, R\}$, where Q is the set of operational states, Q' is $Q \cup q_{halt}$, respectively Γ the tape and input alphabet. From the previous definition we know that $|Q| = N$, $|Q'| = N + 1$, $|\Gamma| = 2$, $|\{L, R\}| = 2$.

The total number of possible transition functions is $(|Q'| * |\Gamma| * |\{L, R\}|)^{|Q|*|\Gamma|}$, which is equal to $[4(N + 1)]^{2N}$, hence the same number of Turing machine configurations. In the Table 4.1 it can be seen that the amount of possible machines grows extremely fast, and with such immense numbers, it becomes unfeasible to run each one of them.

Number of states	Number of candidates
1	64
2	20,736
3	1,677,216
4	25,600,000,000
5	2,821,109,907,456

Table 4.1: Number of possible Turing machine configurations for BB(N)

Thus, filtering the candidate Turing machines before running them became necessary in finding the N-state busy beavers. During the generation process of all of these Turing machine configurations, some of them can be discarded right away, without knowing their full configuration, only after the generation of a few transitions for the transition functions. This technique was taken by Rado himself, and it was later called tree normalization [5]. In this research paper, a similar method was used, referring to it as applying the generation filters.

Besides the generation filters, some filters can be applied when knowing the full configuration of the Turing machine, and they were called compile filters, obviously because they can be put in the application only after knowing the complete transition function.

Last but not least, after starting to run Turing machines, it can be observed that some of them might have similar behavior, consequently making it possible to group them into families of Turing machines. The aim was to find families of Turing machines that will never halt, that will run forever, and to filter them out. As the name suggests, those filters were called runtime filters.

Even with a threshold set for the maximum time or steps to run a Turing machine, applying filters would reduce the computation power needed significantly and would simplify the search for a winner. In the following sections, each one of the filter types mentioned above was investigated thoroughly.

4.2 Generation filters

It was mentioned before that the filtering technique during the generation of all possible Turing machines was called tree normalization [5], and we should examine why it is called this way. As a convention, the starting state was noted as q_0, and the rest of the states were noted as q_1, q_2, ..., q_{n-1}, q_{halt} for $BB(N)$.

The initial tree normalization was generating transition as needed. If a transition was putting the Turing machine in an infinite loop, the generation was stopped there. For example, the transition $(q_0, 0) \rightarrow (q_0, 0, R)$ would read the first 0 on the tape, it would not change the tape, and it would move to the right without stopping.

On the other hand, if it is not possible to affirm that the machine would loop endlessly, from the transitions generated until the current moment, the generation would continue. Considering the first transition to be $(q_0, 0) \rightarrow (q_1, 1, R)$, it was impossible to tell if this machine would get into an infinite loop because it was yet to know how the machine would behave once reaching q_1, so the generation would continue from $(q_1, 1)$. This generation is illustrated in Figure 4.1 and was also explained by Machlin and Stout [5].

With inspiration from this approach, the method used for generating all Turing machines implied a reduced backtracking algorithm. At first, all possible transitions were generated, a total of $|Q| \times |\Gamma| \times |Q'| \times |\Gamma| \times |\{L, R\}|$ transitions, equal to $8N(N + 1)$.

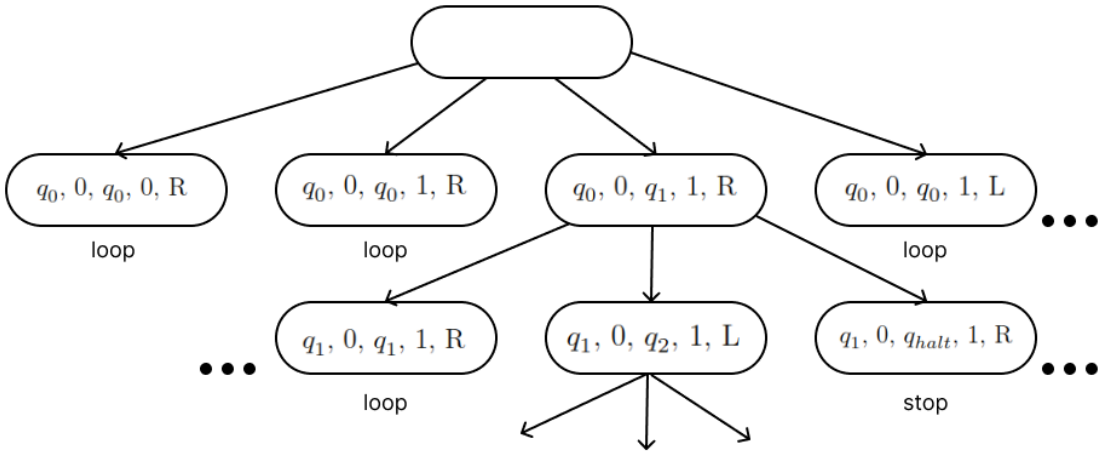

Figure 4.1: Tree normalization [5]

Starting from the first transition, $(q_0, 0) \rightarrow (q_0, 0, R)$, the generation process kept adding transitions to a hash map of transitions, where the key was formed by (state, symbol), which represented the left-hand side of a transition. The transition was added to the current hash map of transitions only if the $(q, symbol)$ of the transition was not already part of the hash map function. Transitions were added while the size of the hash map was smaller than $|Q| \times |\Gamma|$. Once reaching the size of $|Q| \times |\Gamma|$, the transition function was completed, then transformed to a Turing machine, and afterward saved in a list of Turing machine configurations, which would be analyzed later on.

A backtracking algorithm starting from this idea would result in a time complexity of $O([4(\mathrm{N}+1)]^{2\mathrm{N}})$, and for a N starting from 1, this would grow significantly fast, as mentioned in Table 4.1. Moreover, it would generate a lot of redundant Turing machines, as mentioned earlier. Instead of generating them all, once a new transition is added to the current transition function, the whole transition function is checked against some filters. If it passes all the filters, it will be included in the next steps of generation, otherwise the generation from that point on will be stopped.

Not only that the filters designed with the thought of finding, obviously, non-halting Turing machines, but also with the thought of not using redundant transitions or configurations. The generation filters will be described in the subsections that follow. It was mentioned before that the filtering technique during the generation of all possible Turing machines was called tree normalization [5], and we should examine why it is called this way. As a convention, the starting state was noted as q_0, and the rest of the states were noted as q_1, q_2, ..., q_{n-1}, q_{halt} for $BB(N)$.

The initial tree normalization was generating transition as needed. If a transition was putting the Turing machine in an infinite loop, the generation was stopped there. For

example, the transition $(q_0, 0) \rightarrow (q_0, 0, R)$ would read the first 0 on the tape, it would not change the tape, and it would move to the right without stopping.

On the other hand, if it is not possible to affirm that the machine would loop endlessly, from the transitions generated until the current moment, the generation would continue. Considering the first transition to be $(q_0, 0) \rightarrow (q_1, 1, R)$, it was impossible to tell if this machine would get into an infinite loop because it was yet to know how the machine would behave once reaching q_1, so the generation would continue from $(q_1, 1)$. This generation is illustrated in Figure 4.1 and was also explained by Machlin and Stout [5].

4.2.1 Halting transition skippers

In the first place, before the description of filters that target transition functions, a filter for the generation of the transition was used. A crucial observation made was that once entering the halting state, the behavior of the Turing machine is not of interest, only the symbol written on the tape, which should be 1, to make sure that the number of 1's in the output is maximal.

Having settled that, instead of generating every transition of the form $(q_i, symbol_i) \rightarrow (q_{halt}, symbol_j, direction_j), \forall symbol_j \in \Gamma, \forall direction_j \in \{L, R\}$, the only transition of interest is $(q_i, symbol_i) \rightarrow (q_{halt}, 1, direction_j)$ and $direction_j$ can be chosen arbitrary from $\{L, R\}$. For a single pair of $(q_i, symbol_i)$, the number of transitions was reduced from 4 to 1. For the whole Q set of states and the Γ alphabet of symbols, the number of transitions that include the halting state as the outgoing state was reduced from $4 \times |Q| \times |\Gamma|$ to $|Q| \times |\Gamma|$, a decrease of $3 \times |Q| \times |\Gamma|$ transitions.

The overall number of transitions was reduced from 8N(N+1) to N[8(N+1) - 6]. Imagine that instead of using this filter, the BB(4) contest, the winner would have been run multiple times, even though the behavior would be the same for all instances. Or, in a worse case, the one when a Turing machine that never halts is being run multiple times, which would need a complicated mechanism to be detected as non-halting. Such a scenario would consume a lot of computation power without any purpose.

4.2.2 Start state loop

As mentioned at the beginning of this section, one obvious transition that can put a Turing machine in an infinite loop would be $(q_0, 0) \rightarrow (q_0, 0, R)$. The Turing machine starts in q_0, reads the first 0 from the tape, then moves to the right and continues with this behavior for an indefinite period.

The start state loop filter targets the Turing machines that have, in the composition of their transition function, transitions that go into an infinite loop from the start state. Those transitions are:

$$(q_0, 0) \rightarrow (q_0, 0, L), (q_0, 0) \rightarrow (q_0, 1, L)$$

$$(q_0, 0) \rightarrow (q_0, 0, R), (q_0, 0) \rightarrow (q_0, 1, R)$$

Consequently, when the Turing machines were generated and those transitions were part of the transition function, the generation process stopped building them to the fullest. A total number of $4 * (|Q| * |\Gamma| - 1) * (|Q| * |\Gamma| * |\{L, R\}|)$ Turing machines were filtered out, which is equal to a total of $16N * (2N - 1)$ machines.

4.2.3 Neighbour loop

An equivalent comportment of immediately going into an infinite loop can be achieved by transitioning from the start state to a neighbor state that will put the Turing machine in a non-halting state.

Given the starting state reading the first 0 from the tape, then moving to a new state, if the new state would remain in its state and continue moving in the same direction it moved initially, then it means it would loop endlessly. A transition function that would go into a loop created by the neighbor state of the starting state would have the following transitions:

$$(q_0, 0) \rightarrow (q_i, symbol_n, direction)$$

$$(q_i, 0) \rightarrow (q_i, symbol_m, direction),$$

$$i\forall \in \{1, 2, ..., N - 1\},$$

$$\forall symbol_n, symbol_m \in \Gamma,$$

$$\forall direction \in \{L, R\}$$

While the matter of the symbols written on the tape would be trivial because the Turing machine will eventually get into an infinite loop, the direction indicated by the transitions is crucial. It can only be stated that the Turing machine would never halt if the neighbor keeps going in the same direction, or other words, keeps going into previously not visited tape cells.

In the case of moving in opposite directions, after two steps of execution, the head of the tape will be in the starting position, thus making it unable to tell whether the machine will loop or not.

4.2.4 Naive beavers

The winner of $BB(1)$ is trivial, naive might say, and its value is equal to 1. Because the Turing machine has only one state, the configuration of any machine would

have only two transitions. In the event of trying to write multiple 1s on the tape, the machine would go infinitely to the left or right as a result of not having any other choice. Consequently, the only halting machine would contain the transition $(q_0, 0) \rightarrow (q_{halt}, 1, \text{direction}), \forall \text{direction} \in \{L, R\}$.

Starting from $BB(2)$, the expectation is for the value to be greater than the value of $BB(1)$ because the number of states has increased and more complex Turing machines can be built with more states. Thus, in the context of searching for new busy beaver winners, Turing machines that go directly into the halting state should be filtered out.

The naive beavers filter targets those transition functions that contain such transitions, which go directly from the starting state to the halting state, as:

$$(q_0, 0) \rightarrow (q_{halt}, \text{symbol}, \text{direction}), \forall \text{symbol} \in \Gamma, \forall \text{direction} \in \{L, R\}.$$

4.3 Compile filters

Straightforward filters were applied to speed up the generation of Turing machines and to discard the ones that were non-halting. With a complete transition function, more compelling behaviors could be observed. The compile-in-compile filters came from the fact that all the Turing machines that passed the previous filters were fulfilled with transitions, completely compiled.

There are still obvious machines that have been filtered out, but one of the filters described in this section, based on a simple observation, would make an interesting subject for further research.

4.3.1 Never trying to halt

Remembering the definition of the transition function, which is $\delta : (Q \times \Gamma) \rightarrow (Q' \times \Gamma \times \{L, R\}$, it can be noticed that there is no specific rule that states that it must contain at least one transition that goes into the halting state q_{halt}, it is only a possibility for this to happen.

In the context of optimizing the seeking for a busy beaver champion, it was redundant to keep the Turing machines that did not have, in their composition of the transition function, a transition that would end up in the halting state since no matter how much output these machines would produce they will never halt.

This filter would iterate through all the Turing machines that passed the generation filters, respectively through their transitions, and look for at least one transition of the form:

$$(q_i, \text{symbol}_n) \rightarrow (q_{halt}, \text{symbol}_m, \text{direction}),$$

$$\forall q_i \in Q, \forall \text{symbol}_n, \text{symbol}_m \in \Gamma, \forall \text{direction} \in \{L, R\}$$

On the assumption that no transition was found to respect the given pattern, the Turing machine would not pass the filter it will be popped out from the list of valid Turing machines.

4.3.2 Never increasing the score

Another critical aspect that was taken into account when analyzing Turing machines with complete transition functions was to keep only the ones that were trying to increase the score, which are the ones who are producing the most output. If no output was generated at all, no 1's could be written on the machine's tape the machine is of no use for the busy beaver function.

This filter would iterate through all the Turing machines that passed the generation filters, respectively through their transitions, and look for at least one transition of the form:

$$(q_i, \text{symbol}) \rightarrow (q_j, 1, \text{direction}),$$

$$\forall q_i, q_j \in Q', \forall \text{symbol} \in \Gamma, \forall \text{direction} \in \{L, R\}$$

Assuming that no transition was found that respected the given pattern, the Turing machine would not pass the filter it would be excluded from the list of valid Turing machines.

4.3.3 Equivalent Turing machines

To continue with, a genuine examination of the unicity and importance of the states $\in Q'$ concluded that the starting, respectively the halting states are special. They cannot be interchanged with other states, nor ignored in any manner. For the remaining states $\in Q'$, this would not be the case, as it has been previously proved that they are swappable, and if swapped, the resulting machine would behave exactly like the original one [4].

Let TM be any Turing machine, in the current context, a Turing machine that passed the generation filters and the compile filters defined, containing N states. State q_0 is the starting state, and state q_{halt} is the halting state, the special states. The other normal states are noted with $q_1, q_2, ...q_{N-1}$.

Let TM' be the Turing machine derived from TM, but in which every appearance of the state q_i in the transition function of TM would be replaced with q_j, and vice versa, where q_i and q_j are two normal states of TM. The new Turing machine TM' would have the same behavior as TM, it would write the same amount of output, and if it halts, it would halt at the same step as TM.

The current filter aimed to compare each Turing machine in pursuit of finding a mapping between two or multiple normal states of the two machines. A mapping consisted of multiple pairs of tuples (q_{tm1}, q_{tm2}), where $q_{tm1}, q_{tm2} \in Q/\{q_0, q_{halt}\}$, which meant that the state q_{tm1}, from the transition function of $TM1$, could have been interchanged with the state q_{tm2}, from the transition function of $TM2$. Under the circumstances of finding a valid mapping, it would have been possible to state that the Turing machines $TM1$ and $TM2$ would have the same behavior, output, and more importantly, the same score for $BB(N)$.

As intuitive and easy to apply as it seems in theory, this filter method has some significant practical issues. To begin with, it is a filter that needs to run sequentially, without being parallelized. For the first two machines, $TM1$ and $TM2$, if a valid mapping was not found, they should be kept for further comparison with the rest of the machines. Continuing, a search for a mapping between $TM3$ and $TM1$, $TM2$ would be done. Again, if no valid mapping was found between any pair, $TM3$ should be kept for comparison with the remaining machines. As a result, each Turing machine depends on all previous Turing machines analyzed, and none of them could be skipped, because it would mean that also a mapping could be skipped.

Overlooking the absence of parallelism, this filter was implemented and tested for $BB(2), BB(3)$. With a reduced number of Turing machines, after applying the generation and compile filters already mentioned, the idea was to iterate through all machines and create a list of transition function templates. Each new machine found in the iteration would be compared against all the current templates, and if at least one template matched the machine, it would be filtered out, because it meant that a machine with an identical behavior was already chosen. Otherwise, a new template would be created based on the current machine and added to the list of templates.

Transitions of the form $(q_i, \text{symbol}_i) \rightarrow (q_j, \text{symbol}_j, \text{direction}), \forall q_i \in Q, \forall q_j \in Q'$, $\forall \text{symbol}_i, \text{symbol}_j \in \Gamma, \text{direction} \in \{L, R\}$ were encoded as q_i, *symbol*$_i$, q_j, *symbol*$_j$, *direction*. The direction was encoded as: $L = 0$, $R = 1$; respectively, the states were encoding using their number: $q_0 = 0$, $q_1 = 1$, and $q_{halt} = 101$. All elements were concatenated into a string, and separated by a comma.

Transition functions that contained the transitions $tr_1, tr_2, ..., tr_m$ would firstly have their transitions encoded, then they were encoded as $tr_1|tr_2|...|tr_m$. All transitions were concatenated into a string, and separated through a '|'. A complete example can be seen in Table 4.2.

A template was implemented as a list of tuples (regular expression, state from, state to), each of the regular expression representing a transition pattern, and were of the following form: "$(\backslash d), \text{symbol}_i, (\backslash d), \text{symbol}_j, \text{direction}$", where $\forall \text{symbol}_i, \text{symbol}_j \in \Gamma, \forall \text{direction} \in \{L, R\}$. "State from" was extracted from the first $(\backslash d)$ in the regular expression, and "state to" from the second $(\backslash d)$. Therefore, for each transition of a Turing machine, a tuple (reg-

Number	Transition	Encoding
1	$(q_1, 1) \rightarrow (q_{halt}, 1, R)$	1,1,101,1,1
2	$(q_0, 0) \rightarrow (q_1, 1, R)$	0,0,1,1,1
3	$(q_1, 0) \rightarrow (q_{halt}, 1, R)$	1,0,101,1,1
4	$(q_0, 1) \rightarrow (q_{halt}, 1, R)$	0,1,101,1,1
1,1,101,1,1|0,0,1,1,1|1,0,101,1,1|0,1,101,1,1		

Table 4.2: Encoding of a transition function

ular expression, state from, state to) was created, and the whole template consisted of all tuples for that Turing machine.

To check a Turing machine against a template, the filter made use of the encoding of that machine. Using the encoded string of the Turing machine, each regular expression was applied to it. In the case of not matching, it meant that the machine was not of that template, so it would continue the checking with another template.

Otherwise, if the regular expression matched, the mapping would be created. For simplicity, the following notations need to be made:

$$q_{tfrom} = \text{"state from" state in the template's tuple}$$

$$q_{tto} = \text{"state to" state in the template's tuple}$$

$$q_{mfrom} = \text{"state from" state in the matched regular expression}$$

$$\text{on the current Turing machine}$$

$$q_{mto} = \text{"state from" state in the matched regular expression}$$

$$\text{on the current Turing machine}$$

A mapping is a hash map where the keys are represented by all the states from the template, all q_{tfrom} and q_{tto}, and the values are the associated states from the Turing machine that is being checked against the template, q_{mfrom} and q_{mto}.

After a regular expression was matched, it was checked if the values q_{tfrom} and q_{tto} were keys of the mapping. Given that they were, check whether their hash map values are equal to q_{mfrom} and q_{mto}; if they were, it would have continued and applied the other regular expressions, otherwise it would have stopped, because the current mapping integrity was violated. On the other hand, if they were not keys of the mapping, the new keys q_{tfrom}, q_{tto} would have been added with their values q_{mfrom}, q_{mto}.

An important note is that the part of the Turing machine encoding that matched the regular expressions would be removed from the encoding, to avoid the case when there

are two transitions of the same pattern.

The filter would keep applying the regular expressions, populating the mapping with q_{tfrom}: q_{mfrom}, q_{tto}: q_{mto}, and checking the mapping integrity for the keys that already existed. If all the regular expressions were applied without breaking the integrity of the mapping, the Turing machine would be filtered out. Otherwise, a template would be created and used for checking the remaining Turing machines.

Turing machine 1	Turing machine 2
$(q_0, 1) \to (q_1, 1, R)$	$(q_0, 1) \to (q_2, 1, R)$
$(q_0, 0) \to (q_2, 1, L)$	$(q_0, 0) \to (q_1, 1, L)$
$(q_1, 1) \to (q_2, 1, R)$	$(q_2, 1) \to (q_1, 1, R)$
$(q_1, 0) \to (q_0, 1, L)$	$(q_2, 0) \to (q_0, 1, L)$
$(q_2, 1) \to (q_1, 1, L)$	$(q_1, 1) \to (q_2, 1, L)$
$(q_2, 0) \to (q_2, 0, R)$	$(q_1, 0) \to (q_1, 0, R)$

Table 4.3: Turing machines built on the same template

An example of two Turing machines that have the same template can be checked in Table 4.3. If q_1 was changed, in all of its appearances, with q_2, in Turing machine 2, then the Turing machine 2 would be the same as Turing machine 1.

Continuing on the impracticality of this filter, after several iterations, the number of templates will grow, hence a Turing machine would need to be checked against more and more templates. As previously mentioned, the filter was tested on $BB(2)$ and $BB(3)$, and it took much time to run. Moreover, the values for $BB(2)$ and $BB(3)$ are already known, the target is to find filters that will fit for $BB(5)$, $BB(6)$, and so on, beavers that will be more complex, with many more templates, and the filter would perform worse.

Conversely, to make the filter more feasible for computing, a probabilistic approach could be used for choosing multiple Turing machines that should be run against all the current templates. A probability could be generated within a normal distribution, and if it is bigger than a certain threshold, the next 10 Turing machines would be computed in parallel. This would make the filter much faster, and could also reduce the number of Turing machines that will be executed afterwards.

4.4 Runtime filters

After dropping the non-halting Turing machines, which were possible to identify only by looking at their partial and complete transition functions, this section focuses on analyzing the behavior of the remaining machines during their execution time.

A portion of machines did not require much time to be classified as non-halting due to their non-halting nature coming out only after a few steps by visiting too many new cells or by repeating the same output on the tape multiple times. The real burden came with identifying Turing machines that are looping, translating the tape-written symbols multiple times, or other repeating patterns that would be observed after thousands or even millions of steps of execution.

The category of runtime filters is the most extensive and impressive, and when explored, can give the most insight into how powerful the Turing machines are and how just a simple set of rules can generate such an unexpected output.

4.4.1 Short escapers

The short and long escapers filtering ideas were first defined when another attempt to reduce the number of Turing machines analyzed was attempted, but for a different problem, which tackled the calculation of the Kolmogorov complexity using the output frequency of small Turing Machines [13].

Two of the generation filters focused on the Turing machine that would be non-halting by remaining in a transition that would be looping. These filters are called the start state loop and neighbor loop. With the increasing number of states, for the $BB()$ of greater input, the mentioned filters are insufficient to detect all of these simple loops within the Turing machines.

Nevertheless, this filter solved this problem by checking whether the tape size increased in the last step, to the left or the right on a transition of the same degree as $(q_i, 0) \rightarrow (q_i, \text{symbol}, R)$, or as $(q_i, 0) \rightarrow (q_i, \text{symbol}, L)$, where $\forall q_i \in Q, \forall \text{symbol} \in \{L, R\}$.

Having increased within such a transition, it meant that the head of the Turing machine was on a new cell, therefore it was going to read a 0 from the tape and continue increasing the tape size using the same transition. The cases of self-looping were successfully and immediately detected by this filter.

4.4.2 Long escapers

Now that all the Turing machines that contained a transition or a combination of transitions that would put the machines in a blatantly obvious infinite loop were dealt with, the focus was moved toward more complex non-halting behaviors. In that sense, a Turing machine that kept increasing its tape size and alternating between two, three, or multiple states was also in an infinite loop. Let's consider the Turing machine called LE (long escaper), which contains the following transitions in its transition function:

$$(q_i, 0) \rightarrow (q_j, 0, R)$$

$$(q_j, 0) \rightarrow (q_i, 0, R)$$

$$\text{where } q_i, q_j \in Q$$

If LE was run and at some point the state q_i or q_j was reached, with the tape being increased at the previous step, the machine would enter a non-halting state. It would alternate between q_i and q_j infinitely, increasing the tape to the right. Another example worth mentioning includes the following transitions:

$$(q_i, 0) \rightarrow (q_j, 0, R)$$

$$(q_j, 0) \rightarrow (q_k, 0, R)$$

$$(q_k, 0) \rightarrow (q_i, 0, R)$$

$$\text{where } q_i, q_j, q_k \in Q$$

The same behavior would be simulated by the Turing machine that contains these transitions once any of the q_i, q_j or q_k states were reached. Thus, this type of loop could appear for any number of states involved, a number at least greater than 2.

With better analysis of this kind of loop came the observation that a Turing machine will move infinitely to the left or the right if the number of consecutively visited new cells is bigger than the number of states that the machine was built on [13]. If we are running a machine for $BB(N)$, and its tape head moved over $N + 1$ previously unvisited cells, it means that at least one state was seen twice during that movement, it will continue moving over unvisited cells without stopping.

In the implementation, a counter was built that kept account of how many times the head of the Turing machine had visited new cells, or more precisely, how many times the tape size increased, because in the program that executed the Turing machine, the tape was not infinite, its size was increased as needed. If the counter exceeded the number of states the machine was built on, the execution stopped, and the machine was classified as non-halting. In the case of going into an old cell, the counter was reset.

4.4.3 Cyclers

One obvious filter that needed to be included was the one that verified if the current configuration of the Turing machine was seen before. In this context, the configuration of a Turing machine means the group formed of the tape content, head position, and the current state. If the same configuration was seen twice, it meant that the machine entered an infinite loop, so it would not halt.

To check whether the current configuration appeared before, a history of computation needed to be constructed. This history of computation contained tuples of the form $(\text{tape content}, \text{head position}, q_{current})$, where the actual content of the tape was encoded

with the hash function SHA256.

When running the Turing machine for a long time, this approach could cause problems, because the probability of collision between the encodings of the tapes during the execution would increase. Luckily, it is insufficient for the tape encoding to be the one to collide, the head position and the current state should also be identical for a collision of the whole tuple to happen.

To completely mitigate this possibility, another form of encoding could have been used, which is called run-length encoding. With run-length encoding, it is expected that the input that should be encoded is repeating some of its values. For example, if the tape content is 111110001110000, the run-length encoding where 0 is substituted with a, and 1 with b, is equal to $5b3a3b4a$. A substitution was needed in this case because the semantic values of 0 and 1 could be misinterpreted.

This encoding could lead to performance issues because once the tape got big enough, the encoding size would also be bigger. Comparing it to the fixed size of SHA256, run-length encoding would use a lot more memory. Depending on the system that the filter is executed on, a more suitable encoding can be chosen.

The implementation consisted of keeping a vector of tuples, the history of computation, which contained $(\text{tape content encoded}, \text{head position}, q_{current})$. If the configuration was part of the vector of tuples, the execution stopped, then the machine was classified as non-halting. Otherwise, the current configuration would be added to the history of computation, and the execution would keep on going.

4.4.4 Translated cyclers

Last but not least, to conclude the chapter that described the filters used for the busy beaver problem, the filter method for the translated cyclers will be presented. A Turing machine was considered a translated cycler when it kept on writing the same content on the tape but translated to the right or the left of the tape [14]. This type of cycler concentrates the behavior of the filtering methods mentioned in [5] used for all the types of trees. To better understand this behavior, the following example will be examined, where $\tilde{i} \in \Gamma$ represents the current position of the machine's head:

As shown in Table 4.4, at step 25, it can be observed that the Turing machine had written five 0s on the tape, which were followed by the same content that was written on the tape at step 1. Not only the content but also the state was identical in step 25. If the machine had been left to run from step 25 onward, it would have kept adding more 0s before the already written symbols: 00000010, 000000010, and so on. Therefore, it was safe to say that this Turing machine entered an infinite loop.

In this case, the translation involved moving the string "10" to the right, infinitely, and writing 0s before it. The same behavior could easily be seen in the left direction of

Steps									
0					$\tilde{0}$				
1					1	$\tilde{0}$			
2					$\tilde{1}$	1			
3				$\tilde{0}$	1	1			
4			$\tilde{0}$	1	1	1			
5		$\tilde{0}$	1	1	1	1			
6		1	$\tilde{1}$	1	1	1			
.	.	.	.	.	.	.	.	.	.
24	0	0	0	0	0	$\tilde{1}$			
25	0	0	0	0	0	1	$\tilde{0}$		

Table 4.4: Translated cycler to the right

another machine.

The implementation of this filter consisted of populating a hash map, with keys $(state, direction)$ and its values being the tape content at a precise time of the execution. A new entry was inserted when the head of the machine visited a new cell, the hash map key would have been the state reached at the current moment, and the direction the one in which the head moved. Also, the hash map value would represent the values written on the tape after the head movement.

The new entry was inserted if the key $(state, direction)$ was not part of the hash map. Otherwise, if the Turing machine found a new cell, in a state and a direction that was already seen before, the content of the tape would have been checked for a translated cycle.

If the tape content stored in the hash map has been part of the content of the current tape at the end or beginning of the tape, it means that it was translated to the left or the right. In the other case, the tape content for the entry $(state, direction)$ would be replaced with the current content of the tape, and the Turing machine would continue executing.

Chapter 5

Architecture

5.1 Overview

To tackle the demanding computations required by the busy beaver function, selecting the right programming language was crucial. The decision needed to balance speed and efficiency, with clear contenders like C and C++ standing out due to their renowned performance capabilities. These languages offer powerful low-level tools for synchronization and parallelization, such as threads, semaphores, and mutexes. However, despite the apparent advantages of C and C++, in this thesis, Rust was chosen.

In recent years, Rust has surged in popularity, establishing itself as a go-to language for system-level programming and also renowned for having a friendly and dedicated community that welcomes new people to become Rust programmers. It was chosen out of a desire to understand the attractive combination of fastness and memory safety that it provides, which also resulted in a simpler way to write multi-threaded software.

Impressively, Rust achieves memory safety without using any garbage collector, in contrast with the other languages that provide this feature, such as Python or Java. Instead, it has a unique memory management system [1], called the borrow checker [2]. It controls memory by allowing only one owner to a piece of data, meaning that two pointers are not able to modify the same data simultaneously. You either share the data, by creating multiple non-mutating references to it, or you mutate it, indicating that the owner was changed. This behavior also results in the prevention of data race conditions between threads, because they cannot access the same data at the same time. Consequently, this feature represented a strong reason for choosing Rust.

Rust comes with other interesting design choices as well, including the absence of inheritance to avoid its overhead, which was replaced with the traits system, enums that allow you to define a type by enumerating its possible variants, and zero-cost abstractions, to name a few.

[1] Ownership model: https://doc.rust-lang.org/1.8.0/book/ownership.html
[2] Borrow checker: https://doc.rust-lang.org/1.8.0/book/references-and-borrowing.html

In this scenario, Rust was used to generate and filter Turing machines. Once these tasks were completed for a specified problem size $BB()$, the resulting machines were stored in a database. Since inserting data into the database involved I/O operations, an asynchronous runtime was employed. Tokio [3], a crate in Rust, facilitated the execution of asynchronous and concurrent code for this purpose.

As stated above, parallelizing the code without creating additional overhead by creating too many threads was the central purpose of fastening the process of analyzing busy beaver candidates. The architecture comprised two primary processes: generating Turing machines and executing them. Each process will be detailed in subsequent sections.

For insight, the generator produced all possible Turing machines, which were then filtered using generation and compile filters. Afterward, filtered machines were forwarded to the Turing machine runner for execution. During the execution, the runtime filters were continually applied to detect the machines that should be stopped, because they would have gone into a non-halting state otherwise. Upon completion, information about each machine was stored inside a database: if it halted or not, the number of steps it took until it halted or was stopped, the score, and the number of seconds it took to execute. The overall architecture was illustrated in Figure 5.1.

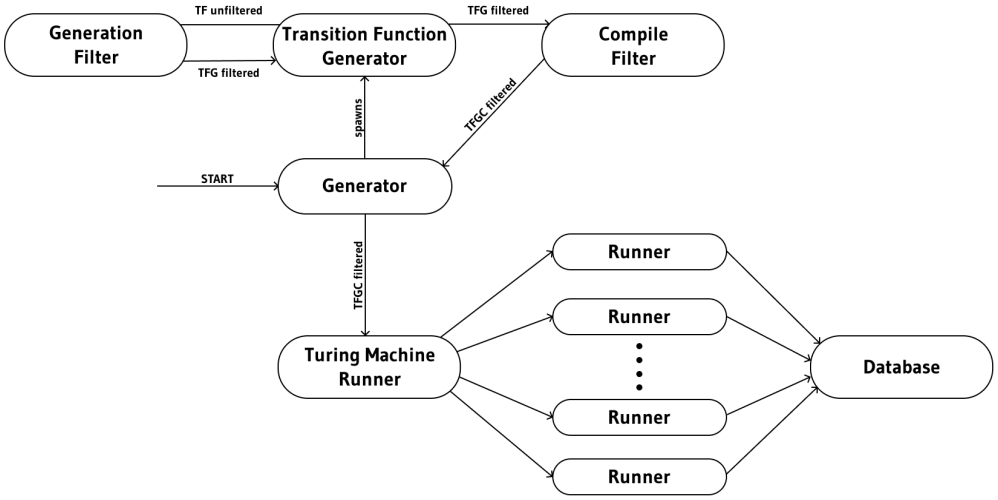

Figure 5.1: Architecture

5.2 Generator

The first step in reducing the busy beaver's candidates was to generate them. During the generation process, the transition functions needed to be filtered by the generation and

[3]Tokio asynchronous runtime: https://docs.rs/tokio/latest/tokio/

compile filters, which required a lot of computational resources. Thus, the architecture was built to facilitate the parallelization of these two filters as much as possible: while a portion of transition functions was being built by the transition function generator, the other portion that contained the complete transition functions was being filtered by the compile filter.

The generator created an additional thread tasked with executing a backtracking algorithm to construct the transition functions. Once a portion of these functions was generated, they were sent to the compile filtering thread. After undergoing filtration, the refined functions were returned to the generator. A comprehensive explanation of this data flow is provided in subsequent subsections. The architecture described above is illustrated in Figure 5.2, where the following naming conventions were made:

1. TF unfiltered = transition functions unfiltered
2. TFG filtered = transition functions filtered by generation filters
3. TFGC filtered = transition functions filtered by generation and compile filters

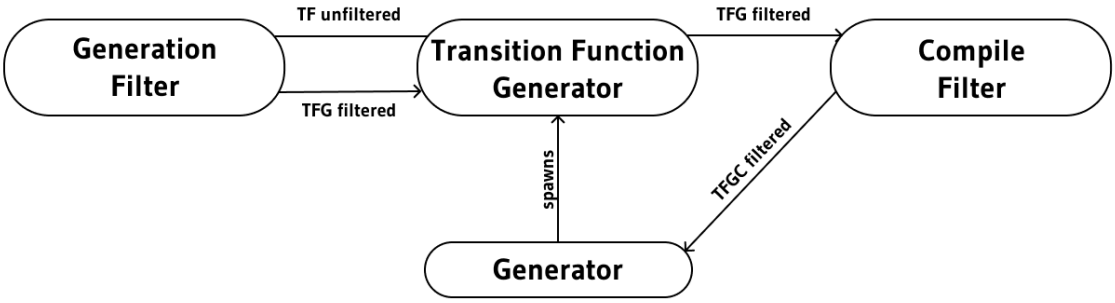

Figure 5.2: Generation architecture

5.2.1 MPSC channel communication

In Rust, communication between threads can be done using a mpsc channel, which stands for multiple producers, single consumer channel. In such a channel, data can be sent from multiple threads, the producers, to only one thread, the consumer. This exchange is unidirectional, implying that the consumer thread could not send data back to the producers. The creation of a channel returned two objects, one Sender and one Receiver, which transferred variables of a certain data type that was set in their constructors. The producers cloned the Sender object and used the clone to send messages through the channel.

To allow communication between the generation thread and the compile filtering thread, two mpsc channels were created. The first one was used to send TFG filtered from the transition functions generator to the compile filtering thread, and the second one to send TFGC filtered from the compile filtering thread back to the generator. In this case, the generator acted as a wrapper for the actual task of generating transition functions, its scope being to handle the communication through the mpsc channels.

After the generation was finished, the channel that sent transition functions to the compile filtering thread was dropped. Following that, the compile filter needed to finish its task, and then it would drop the second communication channel as well.

5.2.2 Transition Function Generator

The transition function generator constructed all the transition functions needed to build the Turing machines for the busy beaver targeted, using the optimization techniques described in the 4.2 generation filters section.

Firstly, all the possible transitions were generated. With the transitions in hand, the backtracking algorithm for assembling transition functions was built. Its target was to create all the possible transitions of size $|Q| \times |\Gamma|$, specifically to cover the whole domain of definition for a transition function, meaning that each function contained all transitions $(q, \text{symbol}) -> (\text{-}, \text{-}, \text{-}), \forall q \in Q, \forall \text{symbol} \in \Gamma$.

Starting from an empty transition function, the algorithm kept on adding new transitions to it, using recursion. After adding a new transition, the transition function was checked against the generation filters and if it passed them, the recursive call was made. Otherwise, the previously added transition was removed from the transition function. Once reaching the size $|Q| \times |\Gamma|$, the transition function was added to a list of complete transition functions, and the recursive call was skipped.

To continue with, each complete transition function needed to be filtered by the compile filter. Sending them one by one through the mpsc channel would create a lot of context switches between the generation thread and the compile filtering thread, so instead, the complete transition functions were sent in batches. The default batch size was 100, but in the 6th section the optimal size for this parameter was explored.

To complete the backtracking algorithm, for every new recursive call it was firstly checked if the complete transition functions array reached the batch size desired, and if it did, they were sent through. Afterward, the array was reset.

Algorithm 1 Backtracking to generate transition functions

```
transitionFunction ← []
completeTransitionFunctions ← []
deepness ← 0
maxDeepness ← |Q| × |Γ|
batchSize ← 100

function GENERATE(index, transitionFunction, completeTransitionFunctions, deepness)
    if deepnees = maxDeepness then
        completeTransitionFunctions.push(transitionFunction)
        if completeTransitionFunctions.size() = batchSize then
            sendToCompileFilter(completeTransitionFunctions)
            completeTransitionFunctions ← []
        end if
        return
    end if

    for each transition in allTransitions[index :] do
        if transition not in transitionFunction then
            transitionFunction.add(transition)
            if generationFilterPassed(transitionFunction) is true then
                GENERATE(index + 1, transitionFunction, completeTransitionFunctions, deepness + 1)
            end if
            transitionFunction.remove(transition)
        end if
    end for
end function
```

In the end, when the backtracking algorithm stopped, if the complete transition functions array size was smaller than the batch size, the remaining transition functions were also sent to the filtering thread.

The recursive version of the algorithm that generates all the Turing machines did not provide a translucent way of identifying how many machines were filtered by which generation filter. This metric was needed in the chapter 7 to compare how well each filter worked for the filtering of different busy beaver problems. As a result, an imperative implementation was built using a queue to have more control and to facilitate the counting of the filtered Turing machines. Instead of making the recursive call, the new transition function was added to the queue and would have been filtered in the following iterations.

5.3 Turing machine runner

Thereafter, the Turing machine runner will execute every Turing machine that passed the generation and compile filters. This execution implied creating two separate tokio threads: one for running the Turing machines and one for inserting them in the database. The communication between the two threads was done using a tokio mpsc channel, where the producers were represented by the finished Turing machines sending their score, steps, and other information to the single consumer, which was the thread that inserted those machines into the database. This behaviour can be analyzed in Figure 5.3.

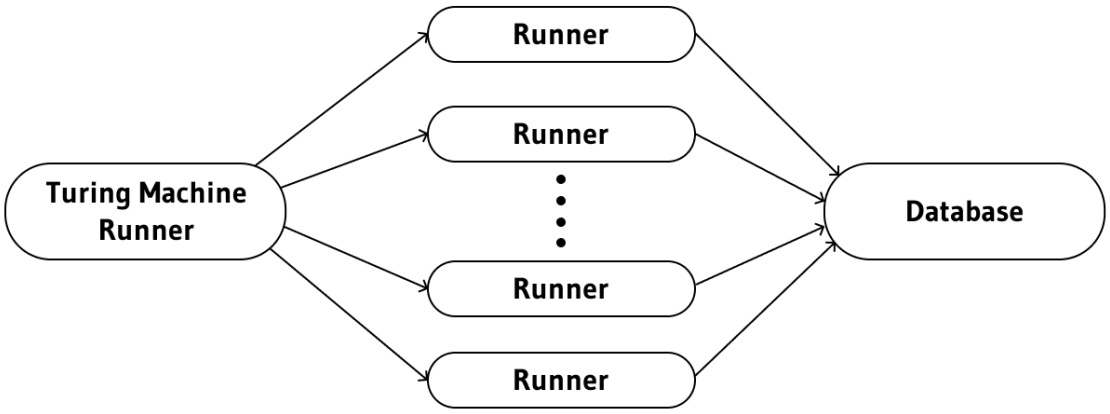

Figure 5.3: Turing machine runner architecture

A tokio mpsc channel was used in this case because the standard mpsc communication channel does not support asynchronous tasks, which in this case were needed to populate the database with the newly explored candidates.

Instead of executing each machine sequentially, we made use of the parallelism by creating a rayon pool thread [4] to run each Turing machine. The default value for the maximum number of threads that could run Turing machines at the same time was limited to 10 because creating one thread for each machine and running them simultaneously would result in CPU thrashing, consequently not optimizing the execution. This threshold will be added to the variables tuned in chapter 6.

When a Turing machine needs to be executed, it is submitted to the thread pool and assigned to an available thread. The thread executes the machine and then returns to the pool for reuse. The machine's results are saved, and once all machines are done, the results are sent to the database. For the initial analysis of $BB(N)$, results are inserted into the database in batches; if the experiment has been run before, the database is updated.

[4]Rayon thread pool: https://docs.rs/rayon/latest/rayon/struct.ThreadPool.html

5.4 Database

As outlined before, not only running Turing machines was computationally expensive, but also generating them. For smaller beavers, namely $BB(2)$ and $BB(3)$, the search for the winner could be done in only one execution of the program, but for bigger beavers, which are the ones of interest, this was not the case.

The search for the prime candidate involves multiple executions and analyses of Turing machines, necessitating access to the previously generated ones. This is where the importance of a database became evident.

After the generator stopped creating Turing machines, every machine that passed the generation and compile filter was inserted into the database. Now, consider that in the future the same experiment would need to be run again, having the machines stored in the database facilitates this process, because the extraction from there would be much faster than generating them all over again.

Therefore, if the program wanted to execute $BB(N, M)$, where N was the number of states and M was the number of symbols for the busy beaver function, a look-up in the already executed Turing machines would have been made to check if the generation process for this beaver was done previously. If it was, the non-halting Turing machines were loaded from the database, otherwise, they were generated.

5.4.1 Tables

The table schema from the database, in which the Turing machines were saved, can be found in Table 5.1. This structure facilitates extraction for a handful of different experiments: running all the machines that did not halt previously with a different maximum time or step threshold, running all the machines that took the same time to execute and examine patterns in their tape composition, and so forth. In addition, it made the extraction and analysis of the winners very straightforward.

Column name	**Data type**	**Description**
Transition function	TEXT	Encoding of the transition function
Number of states	TINYINT	Number of states of the BB
Number of symbols	TINYINT	Number of symbols of the BB
Halted	TINYINT	Halted or not during execution
Steps	BIGINT	Number of steps executed until stoppage / halt
Score	BIGINT	Score obtained until stoppage / halt
Time to run	INT	Time took to execute, in seconds

Table 5.1: Turing machine table in MySQL Database

5.4.2 Running on Docker

For further research, this software program might be migrated to a more powerful machine, with which more computationally expensive experiments could be run. In this case, the database should be easily portable and set up on the new machine. From this observation the need for Docker arised.

The MySQL Database ran inside a Docker 3.8 container. A persistent volume was created and mapped to the container to ensure that the data remained intact even if the Docker container was stopped or removed. This volume kept all the Turing machines that were inserted in the database. The environment establishment was done using another tool, docker-compose, that allows the developers to run multiple docker containers simultaneously, and to configure them more easily. Within the docker-compose file, a complete database container was configured with the database name, root password, a user with its password, the port to run on, and the volumes.

To connect to the database, the sqlx crate was used in Rust. With this crate, a pool of 8 connections to the database was created based on a connection string that included the MySQL database driver, database name, user, and password needed to connect to the database. This pool of connections was managed inside a struct called DatabaseManager, allowing the selection, insertion, and deletion of Turing machines within their table.

Chapter 6

Benchmark

Even with the most appropriate design choices being made, that is generating and inserting Turing machines in batches, respectively executing them in parallel threads, there was still room for improvement by exploring the best parameters for these particular tasks. In the following sections, the search for the optimal values of these parameters was done for the $BB(3)$ function.

6.1 Generator batch size

At the outset, when the Turing machines were generated, the compile filter waited to filter a batch of machines, and then it sent them back to the generator. The Turing machine batch size that was sent to the compile filter was the parameter explored in this section.

The search for the best parameter consisted of trying different values to determine which yields the best performance. The increase of the batch size continued until a decreasing trend in the time it took to generate the Turing machines was seen.

Batch size	**1st**	**2nd**	**3rd**	**Mean**
100	7.61s	7.65s	7.54s	7.60s
500	7.54s	7.68s	7.53s	7.58s
1000	7.24s	8.08s	7.69s	7.67s
1500	7.24s	7.38s	7.35s	**7.32s**
2000	8.14s	7.48s	7.76s	7.79s
5000	7.43s	7.37s	7.57s	7.45s
10000	8.02s	7.33s	7.66s	7.67s

Table 6.1: Turing machine generator batch size benchmark

6.2 Insertion batch size

After the Turing machines were generated, they needed to be inserted into the database. Inserting the machines in batches instead of inserting them sequentially was the better approach since it avoided the additional overhead of multiple network round trips, logging, and indexing.

In line with the previous search method, the batch size was increased until the performance was getting worse, and the best value was extracted from all the trials.

Batch size	**1st**	**2nd**	**3rd**	**Mean**
1000	12.03s	12.52s	12.10s	12.21
1500	11.77s	11.80s	11.89s	**11.82**
2000	11.73s	12.26s	11.74s	11.91
2500	11.91s	11.89s	12.14s	11.98
5000	11.70s	11.73s	12.03s	**11.82**

Table 6.2: Turing machine insertion batch size benchmark

6.3 Threads to run Turing machines

The last step, and the most difficult one regarding computation, was to run the Turing machines. This idealistic machine, in its description, has access to an infinite tape and has infinite computational time. In reality, this behavior was impossible to simulate, so it needed to come as close as possible to its theoretical attributes.

Unlike previous searches, a significant discovery was made in determining the ideal number of threads for running Turing machines. In recent years, CPUs have integrated multiple computing cores on the same physical chip, each assigned multiple hardware threads [10]. These threads are recognized by the operating system as logical CPUs, suggesting that, in theory, maximum computational power could be achieved by mapping each software thread to a logical CPU.

However, in practice, the optimal number of threads can vary due to factors such as CPU architecture, concurrent tasks, and task parallelism. Despite all these factors, this theoretical number gave an intuition on where the optimal number of threads might be. The expectation was to be around the theoretical value.

The computer on which the parameter search was done had 8 logical CPUs, meaning that the optimal value might lie around this number. Different tests were run in which the number of threads used was around this value. Also, the number of threads was increased until the performance started to worsen.

Threads	1st	2nd	3rd	Mean
1	13.94s	13.48s	13.52	13.65
2	7.56s	7.52s	7.69s	7.59s
4	4.74s	4.74s	4.87s	4.78s
8	4.14s	4.18s	4.15s	**4.16s**
10	4.18s	4.20s	4.16s	4.18s
12	4.17s	4.23s	4.25s	4.21s
16	4.58s	4.20s	4.27s	4.35s

Table 6.3: Threads used to run Turing machines benchmark

The optimal number of threads for the thread pool used to run Turing machines was equal to 8, which is the same number of logical CPUs of the machine that the program was executed on. It can be observed that the performance significantly increased until reaching a total of 8 threads, and then the thread count around that number gave almost the same good performance. Once starting to increase the number, the performance was slightly getting worse. As expected, the best performance was obtained by a thread count around the number of logical CPUs and, in this case, for the exact number of logical CPUs.

As a result of the optimal parameters seeking made, the benchmarking analysis conducted in this chapter offers valuable insights into the performance of the architecture across a spectrum of parameters. The main focus was to find the values that would make the investigation of the Turing machine the fastest. In a further analysis of the architecture, other metrics can be explored, including memory utilization, scalability, and the behavior when running Turing machines that take millions of execution steps.

Chapter 7

Conclusions

7.1 Results

This section will concisely present the results of the described research on finding ways to reduce the candidates for the busy beaver function. The method of analyzing the filters was similar to the ones described in the chapter 2 [5] [9], which consisted in applying the generation and compile filters to all the Turing machines, then trying to approve the conjectures for $BB(2) = 4$, $BB(3) = 6$ and $BB(4) = 13$.

	$BB(2)$	$BB(3)$	$BB(4)$
Halting skippers	68.36%	71.23%	72.75%
Start state loopers	14.06%	8.85%	6.41%
Neighbour loopers	3.12%	2.72%	2.26%
Naive beavers	3.52%	2.21%	1.60%
Never halters	6.94%	9.73%	10.97%
Never scores	0.46%	0.15%	0.04%
Total	96.46%	94.89%	94.04%
Remaining	732	854,488	1,525,760,000

Table 7.1: Generation and compile filters results

Table 7.1 showcases the results of running the generation and compiling filters on each busy beaver instance. An expected but also intriguing result is that with the increase of N for the N-state busy beaver problem, the two categories of filters are filtering out fewer Turing machines. This increases the fact that only by adding one state the resulting Turing machine can be significantly more influential than the previous version.

The following step was to run all the remaining N-states Turing machines for the exact number of steps in which $S(N)$ busy beaver halted. Due to computational limitations,

the following steps of the analysis were continued only for $S(2) = 6$ and $S(3) = 21$.

	$BB(2)$	$BB(3)$
Steps	6	21
Halting	416	402,156
Non-halting	316	452,332

Table 7.2: Execution of remaining Turing machines results

In Table 7.2, it can be observed how many Turing machines halted within the number of executed steps. Among the halting Turing machines, for each $BB()$, the highest score was equal to the score defined in each of the conjectures. This means that the filters did not filter out the winners, and the last step in this analysis was to run each machine that did not halt in the number of steps targeted and to prove that it will never halt. The transition function for the winner of $BB(2)$ corresponds to Table 7.3, respectively for the winner of $BB(3)$ corresponds to Table 7.4.

	0	1
q_0	$(q_1, 1, L)$	$(q_1, 1, R)$
q_1	$(q_{halt}, 1, L)$	$(q_0, 1, R)$

Table 7.3: $BB(2)$ winner, 6 steps

	0	1
q_0	$(q_1, 1, R)$	$(q_2, 1, L)$
q_1	$(q_2, 1, R)$	$(q_{101}, 1, L)$
q_2	$(q_0, 1, L)$	$(q_1, 0, 1)$

Table 7.4: $BB(3)$ winner, 11 steps

Lastly, the holdouts for each busy beaver instance were run for 1,000 steps. Table 7.5 shows that for all the cases, at least 90% of the holdouts were filtered by the runtime filters, and for $BB(2)$ all holdouts were filtered, meaning that the conjecture was proved to be true. Not only that the remaining holdouts need to be executed for more steps because they might cycle in an extended number of steps, but also new filters might be explored by analyzing further the tape behavior of the machines.

After running all the machines for $BB(3)$ for 21 steps, 5384 holdouts machines remained. The increase of the number of steps to 1,000 reduced the number of holdouts to 5352, all the 34 machines being filtered by the translated cyclers filter. Further investigation of these holdouts would consist of running the machines for an extended period to exploit the translated cycler filter more and searching for new runtime filters.

7.2 Adaptability

The primary aim of this thesis was to develop methods for filtering the original Busy Beaver function, focusing on Turing machines that use the tape alphabet $\Gamma = \{0, 1\}$.

	$BB(2)$	$BB(3)$
Short escapers	21.52%	19.67%
Long escapers	43.67%	7.99%
Cyclers	13.29%	12.06%
Translated cyclers	21.52%	59.11%
Total	100%	98.82%
Remaining	0	5,352

Table 7.5: Runtime filters results for holdouts with 1,000 steps

Introducing more symbols into the alphabet allows for the construction of increasingly complex Turing machines. To address this complexity, the architecture and code were designed to enable the analysis of Turing machines with more elaborate alphabets, in the end, to be able to find new filtering techniques for these machines.

Analyzing these versions of the Busy Beaver function demands exceptionally powerful computers, even for smaller instances. Consequently, the likelihood of identifying a winner diminishes significantly. Table 7.6 presents the boundaries for Busy Beavers using larger tape alphabets, highlighting the challenges and potential breakthroughs in this area of research.

	2 states	**3 states**
2 symbols	6	21
3 symbols	38	>= 119,112,334,170,342,540
4 symbols	>= 3,932,964	>= 10^{2048}

Table 7.6: $BB(N, M)$ for different Ns and Ms [17]

7.3 Further research

In light of these findings, the generation and compile filtering methods performed well in reducing the Turing machines that needed to be executed for every instance of the $BB()$ function analyzed, the reduction being over 94%. As for the runtime filters, their performance is also notable because they proved their validity by not filtering out the winning candidates and leaving only a tiny amount of holdouts to be examined. The suggestions for further research imply tackling the following aspects:

1. For the busy beaver with the alphabet $\Gamma = \{0, 1\}$, a more memory-efficient way of storing the tape could be explored. One option could be to store each cell as a bit

and to extend the tape as needed.

2. Finding a better implementation for the filter that identifies equivalent Turing machines. This filter could group the remaining holdouts by finding the ones equivalent when swapping their states or symbols and then analyzing only one member of each group. Such an approach would reduce the computational power needed by a large margin.

3. Implementing other existing runtime filters, as the counter [5] or the backtracking [5] methods. These filters would certainly reduce the number of holdouts because they were already proven correct and useful.

4. Running the holdouts and visualizing how the tape values change in time to come up with new ideas for runtime filters. The visualization of the tape is essential in discovering non-halting behaviors among Turing machines. This is how the idea for many mentioned filters came up, for instance, the Christmas tree and all its variations [5].

5. Using computer vision to automatically identify non-halting behaviors of the Turing machines by feeding a model with images of the tape. The number of holdouts for bigger busy beavers would be significantly higher, thus manually analyzing the tape behavior of the Turing machines might become unfeasible at some point. In this sense, a convolutional neural network could be trained to identify this behavior using images that represent the change of tape over time. A good visualization of the tape that could be fed into a computer vision model can be found on the bbchallenge website. [1]

These research directions will reduce the holdouts for each N-state busy beaver. The main focus should be to discover new runtime filtering methods because there will be more non-halting families of Turing machines among the vast beavers. Success in reducing the candidates for the busy beaver function and eventually providing, disproving, or defining new conjectures for its value will enhance our understanding of computational boundaries, complexity, and non-computability.

[1] Tape visualization: https://bbchallenge.org/13223284

Bibliography

[1] Scott Aaronson. "*The Busy Beaver Frontier.*" In: (2020). URL: https://www.scottaaronson.com/papers/bb.pdf.

[2] Scott Aaronson. *Who can name the bigger number?* [Accessed on 11/12/2023]. URL: https://www.scottaaronson.com/writings/bignumbers.html.

[3] Allen H. Brady. "*The Determination of the Value of Rado's Noncomputable Function for Four-State Turing Machines.*" In: *American Mathematical Society* (Apr. 1983). URL: https://docs.bbchallenge.org/papers/Brady1983.pdf.

[4] James Harland. "*Generating Candidate Busy Beaver Machines (Or How to Build the Zany Zoo).*" In: (Oct. 2016), pp. 5–13. URL: https://arxiv.org/pdf/1610.03184.

[5] Rona Machlin and Quentin F. Stout. "*The Complex Behaviour of Simple Machines.*" In: *Physica D: Nonlinear Phenomena* 42 (1990), pp. 85–98. URL: https://web.eecs.umich.edu/~qstout/pap/busyb.pdf.

[6] Heiner Marxen and Jürgen Buntrock. "Attacking the Busy Beaver 5." In: *Bulletin of the EATCS,* (1990), pp. 247–251. URL: https://turbotm.de/~heiner/BB/mabu90.html.

[7] De Mol and Liesbeth. *Turing Machines.* [Accessed on 16/02/2024]. URL: https://plato.stanford.edu/archives/win2021/entries/turing-machine/.

[8] Tibor Radó. "*On Non-Computable Functions.*" In: *Bell System Technical Journal* (1962). URL: https://ia802905.us.archive.org/21/items/bstj41-3-877/bstj41-3-877.pdf.

[9] Tibor Radó and Shen Lin. "*Computer studies of Turing machine problems.*" In: *Journal for the Association of Computing Machinery* (Apr. 1965), pp. 196–212. URL: https://dl.acm.org/doi/pdf/10.1145/321264.321270.

[10] Avi Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts.* Wiley, 2018, pp. 221–223.

[11] Michael Sipser. *Introduction to the Theory of Computation, 3rd edition.* Cengage Learning, 2013, pp. 181–182.

[12] Michael Sipser. *Introduction to the Theory of Computation, 3rd edition*. Cengage Learning, 2013, pp. 167–169.

[13] Fernando Soler-Toscano, Hector Zenil, Jean-Paul Delahaye, and Nicolas Gauvrit. "*Calculating Kolmogorov Complexity from the Output Frequency Distributions of Small Turing Machines*." In: (May 2014), pp. 9–10. URL: https://arxiv.org/pdf/1211.1302.

[14] Tristan Stérin. *[Deciders] Translated cyclers*. [Accessed on 22/01/2024]. URL: https://discuss.bbchallenge.org/t/decider-translated-cyclers/34.

[15] Tristan Stérin. *Current zoology*. [Accessed on 22/01/2024]. URL: https://discuss.bbchallenge.org/t/current-zoology/23.

[16] Alan M. Turing. "*On Computable Numbers, with an Application to the Entscheidungsproblem*." In: *Proceedings of the London Mathematical Society* s2-42.1 (Nov. 1937), pp. 230–265. URL: https://doi.org/10.1112/plms/s2-42.1.230.

[17] Wikipedia. *Exact values and lower bounds for Busy Beaver*. [Accessed on 13/12/2023]. URL: https://en.wikipedia.org/wiki/Busy_beaver#Exact_values_and_lower_bounds.