University of St Andrews
School of Computer Science
Junior Honours

CS3052
Computational Complexity

Practical 2: Turing Machines

April 2018

150003431

Contents

1
Introduction

1.1 Overview

This practical covered implementation and analysis of Turing machines for solving binary problems such as language recognition. Its model of computation is explored and its theoretical computation is compared with that of experimental. The implementation of Non-deterministic Turing machine is also investigated. This report outlines my approach to the programming of the algorithms and analysis of implementations.

1.2 Instructions

Code

The test suite is designed as a Maven project where source code and JUnit test is located in relevant directories. The test can be run on command line, at the directory where “pom.xml” file is located. It can be used to test all 4 problems of Task 2 and the additional problem of Task 4: by adding appropriate arguments upon execution.

Data

All Turing machine description files are located in “testData”. Each file is denoted in order with “tm4.txt” as being the description file of a Non-deterministic Turing machine. Any txt files in “t*_in.txt” format will contain example input string which is generated for testing purposes. The test will run the main method of *runTm* program. Any csv files in “t*_output.csv” format located in “sample_outputs” directory are the output csv file containing the result of experiment runs.

An empty directory called “output” is provided, as the path where any output files created by testing would be directed to.

Important Commands

Following commands can be run on command line (Terminal) in the lab machine. All tests for the problems under the provided setting will take no more than 30 seconds.

- **mvn clean**
  Remove current compiled class and output files for a new build.
- **mvn clean test**
  The project will be rebuilt, and run test with default settings. Without any extra arguments passed, the test will test problem 1 and checks that all valid input strings are correct as expected. This can be run to generate the output which can be used as a comparison to that generated by non-deterministic Turing machine set up. The output file will be called “t1_dtm_700_output.csv”.

- **mvn test -Dproblem="${number between 1 and 4}"**
  Each number i passed to *-Dproblem* as string will denote running ith problem of task 2. The data will be tested on random inputs of size n. The output file will be called "t*_output.csv".

- **mvn test -Dproblem="${number between 1 and 4}" -Derrors="f"**
  The *-Derrors* property added on will change the format of the test. That is, random strings generated to test will have a mix of valid and invalid strings which will be either accepted or rejected by the Turing machine. Hence, the test will now check that the Turing machine can make correct decisions. The output file will be called "t*_errors_output.csv".
- **mvn test -Dtype="nd"**
  The *-Dtype* property defined alone or optionally along with *-Derrors* proprety will be testing the Non-deterministic Turing machine. The output file will be called "tm4_output.csv".
- **mvn exec:java -Ddtm="${description file filepath}" -DreadSymbol="${input file filepath}" -Dtype="${number between 1 and 4}"**
  This command can be used if you would like to run the program `runtm` directly. Given a string inside the input file, it will print out the result on command line. Input file can be created manually or by calling the tests first and then using this command to run the program directly.

More details will follow in the next chapter regarding the testing.

2

Task 1

Write a program that can simulate a given (one-tape, deterministic) Turing machine (or TM) on a given input string. Specifically, you should create a program `runtm` *that takes as input a TM description file and a text file, and runs that TM on that input.*

2.1 Design and Implementation

The general design is object-oriented, having class instances to be used for reading input files and processing inputs after constructing Turing machine simulators. Hence, the program `runtm` is written to take command line arguments of Turing machine description file and input file paths. The program consists of DTMReader to read the description file and build the simulator by collecting information including states, tape alphabet and transitions. Then it will build a simulator DTM which will read the input. Some assumptions have been made while designing this program. Firstly, the input file should ideally have one string which to read and decide to accept or reject, since the program itself would return a single boolean to the result. Secondly, the alphabet denoted inside the description file will be the tape alphabet.

Deterministic Turing Machine File Reader - DTMReader

DTMReader extends from its superclass Reader; inheritance is applied for efficiency considering the future implementation for non-deterministic case. Any sanity checks of whether the description is in a valid format will be applied to the file during the reading stage: then any issues with the description file will be noted with a meaningful message to find where the error has derived from. Checking whether there exists transitions from a state with the same input symbol will be rejected as it does not support non-determinism.

Deterministic Turing Machine Simulator - DTM

DTM extends from its superclass TM due to the similar reason as for the Reader. It will have a list of states, a list of tape alphabet and a hash map of transitions. Key will be a tuple consisting of a state and a symbol which will allow getting the next combination a $O(1)$ time operation; where the value retrieved will be a Pair consisting of another tuple and a string, denoting the direction to move for the head of the tape. The tape will be a list containing blank symbols at the start and will be extended whenever required.

The *process* method of **TM** will read the input string into the TM machine to decide whether to accept or reject. It is an iterative implementation which is preferred over recursive due to the possibility of having space limitation on the stack as function calls accumulate. Since every transition will deterministically lead to a unique next combination, any invalid combination of current state and currently read symbol will lead to the machine rejecting the string. It will check whether it is at an accepting state for every transition and accept if found.

3
Task 2&3

Devise Turing machines to solve the following problems. Also devise Turing machines to solve at least two other problems, of your choice. Submit the TM description of your solutions in the format described above, and in your report describe how you tested your solutions for correctness, and why you think they correctly solve the problems given

Analyse, theoretically or experimentally or both, the complexity of your TM algorithms. You will have to count how many transitions are taken by the Turing machine on inputs of length n, as a function of n. Submit your experimental results, and your analysis (theoretical and experimental), as part of your report. If you have data files, scripts to produce graphs, or statistical analyses, submit those as well.

3.1 Testing

For every problem, Turing machine algorithm used to solve it is discussed. For all the problems, it is assumed that it can also accept a blank character by default. The Turing machines designed are all single-tape. The correctness is tested by using various input strings to check that not only it accepts a string from the language it recognises, it also rejects those are not. When it comes to designing the TM description file, it is assumed that any transition that doesn't exist in the transition definitions will be rejected by the machine: meaning that there is no need of defining rejecting states. For the testing of the both occasions, methods generating correct input strings are written, while method to obtain invalid strings containing errors are also written.

Testing is done under the Junit test suite called *PalindromeTest*, which can be used to test all problems solved in this practical, despite of its name. It is a parameterised test which will test on every input string generated at the start. It will use the method `runtm` inside the program `runtm` which will have the equivalent functionality as calling its main method, but compatible in testing. The design of string generation and testing follows the similar format to the previous practical. The length of string size, its range, and repetition of each length to get various strings are set to appropriate values by default. Since the output of the testing will be useful for the later analysis, range of length n is set to be (100, 1000) with a step size of 100 and with 20 repetitions of a given length. As the string gets generated, the expected output (decision) is also recorded so that after running the simulator its result can be compared with the expected to verify its correctness. This design also helped finding errors and applying different corner cases.

3.2 Complexity Analysis

The complexity of each designed Turing machine is analysed both theoretically and experimentally. In order to prevent reduced accuracy, complexity will be denoted in terms of the length of an input string, n. Asymptotic complexity will be analysed so some simplification at the analysis stage is done. Additionally, for experimental approach, data collected will be the time taken to run the string and the number of transitions it will take to the accept states/rejected. The number of transitions can easily be compared with the theoretical complexity. This is also a preferable option considering that the number of transitions will be machine independent measure of complexity , which help to remove variance of machine power/speed as a factor of differences. Analysed values will be compared with graphical visualisation. Its complexity will be denoted as $f(n)$.

3.3 Problem 1: Palindrome

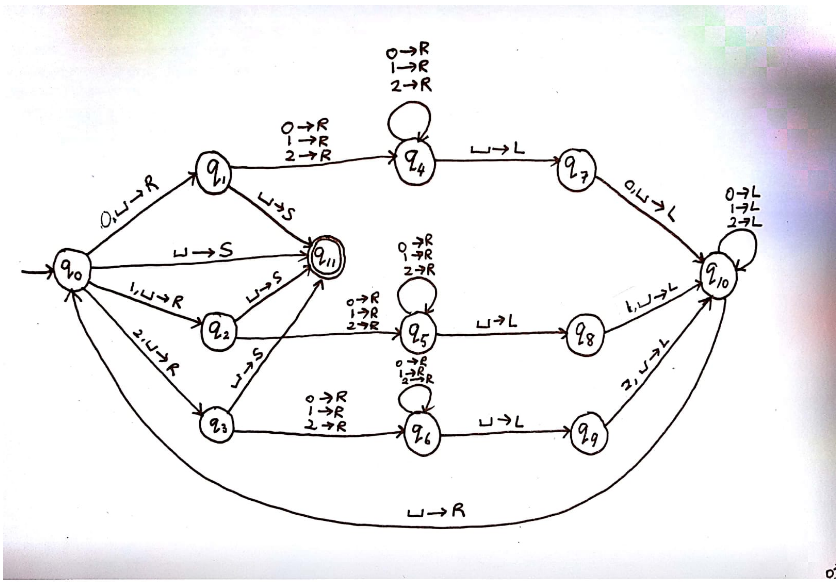

Figure 3.1: *A State Diagram of Deterministic Turing Machine accepting Palindromes*

Design inspiration of the above is from the secondary resource, [**t1**].
The Turing machine designed, as shown in figure 3.1, can accept palindromes of both even and odd lengths. Additionally, it is assumed that a string of length 1 can be also accepted as a palindrome.

A string of odd length n will give

$$\Big(\sum_{i=0}^{\frac{n}{2}} 2(2i+1)+1\Big)+1$$

We can denote i being the length of substring w. Then for every i the TM will traverse twice the whole string with $2(2i+1)+1$ times, with extra step returning from the blank symbol denoting the end, coming back and forth. Then the last extra step to check the end. If n is odd, its $int(\frac{n}{2})$ will be $\frac{n-1}{2}$. So by replacing the current value to this, the closed form of the summation will give $f(n) = n^2 + \frac{3}{2}n - \frac{1}{2}$. Hence

$$f(n) = O(n^2)$$

The calculation will be the same for even length n, with only difference is that n will be used in the summation without any modification. Therefore, it is analysed that it will be running in polynomial time of order 2.

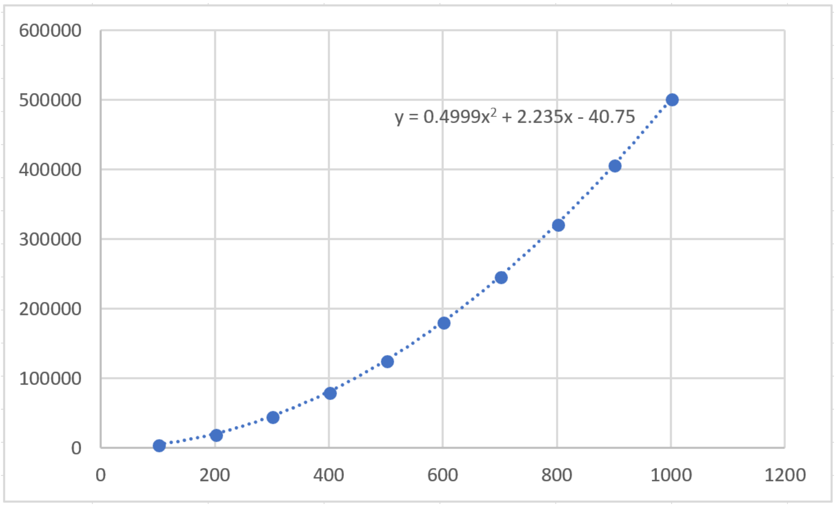

Figure 3.2: *A Plot of the number of transitions to accept inputs of varying length for problem 1*

Testing output is used to analyse its complexity experimentally, which is plotted in figure 3.2 and as a log-log plot in 3.3. The first plot shows a clear trend of increase in the number of transitions. The line of best fit estimates itself to be polynomial. The next plot also supports the argument: it displays a straight line with equation $y = 0.5511x^{1.986}$ showing that the number of transitions (y) is in monomial relationship with the length n (x) of order 2.

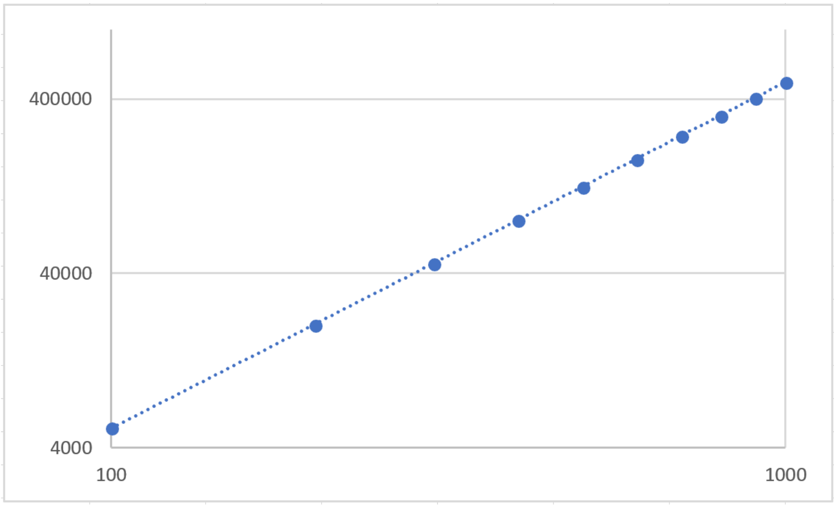

Figure 3.3: *A Log-Log Plot of Figure 3.2*

3.4 Problem 2: Binary Addition

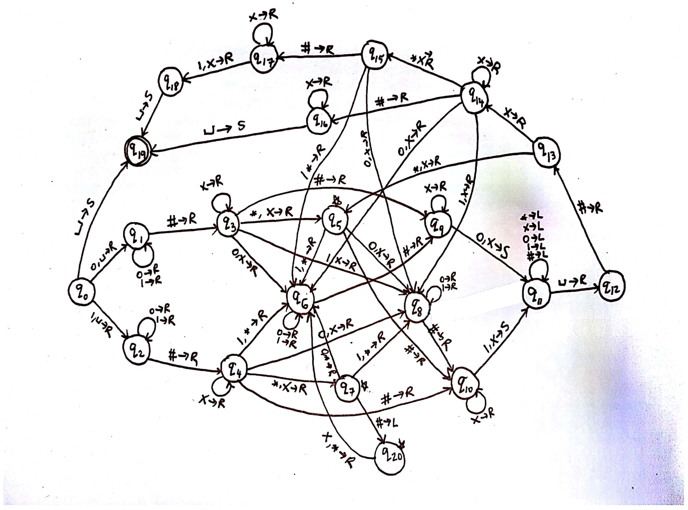

Figure 3.4: *A State Diagram of Deterministic Turing Machine for Problem 2*

For theoretical analysis, it is simplified that every substring of binary number w will have length m: this will allow us to estimate the total length of the string to be 3m. Then the design of the algorithm is that it reads every (i)th digit of w1 and w2 and checks its added value by going to (i)th digit of w3. After reaching (i)th digit of w3, it will move back to (i+1)th digit of w1. Any carrying bit found after checking the bit value of w2 will be denoted by the special character * rewritten back, rather than a marking character X. This will be then taken account when reading the next digit. I have tried to reuse transitions if necessary to minimise the number of states for efficiency.

The design deals with any corner cases such as differing length of w1 and w2, and having a last carry bit to write in w3. Hence, the number of traversals gives us the number of moves to be $m * (4(m + 2))$ giving $4m^2 + 8m$ where extra 2 includes $\#$ characters in between the substrings. Therefore, considering $n \approx 3m$the complexity can also be estimated to be a polynomial time of $O(n^2)$.

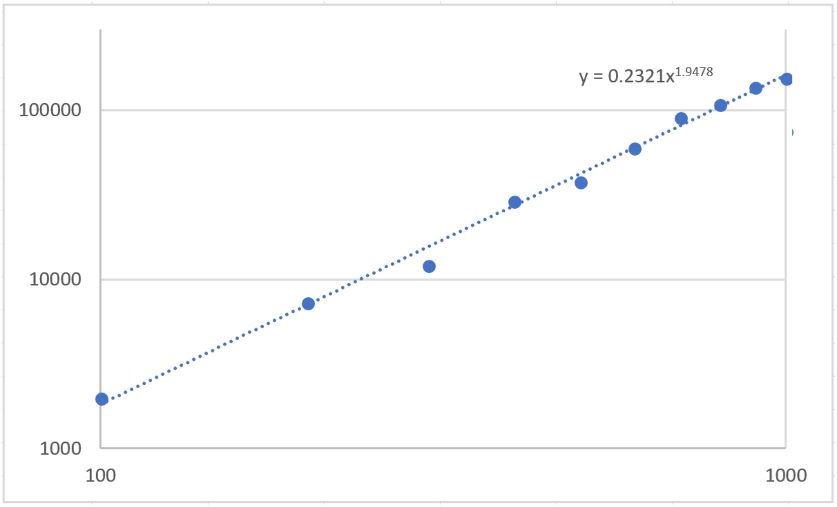

Figure 3.5: *A Log-Log Plot of the number of transitions to accept inputs of varying length for problem 2*

Experimental result plotted in figure 3.5 does agrees with the estimation of monomial relationship of order 2.

3.5 Problem 3: Equal Number of 0 and 1

The first additional problem I have come up with is the TM which can recognise strings of the language {w | from alphabet (0,1) which has the equal number of 0s and 1s}.

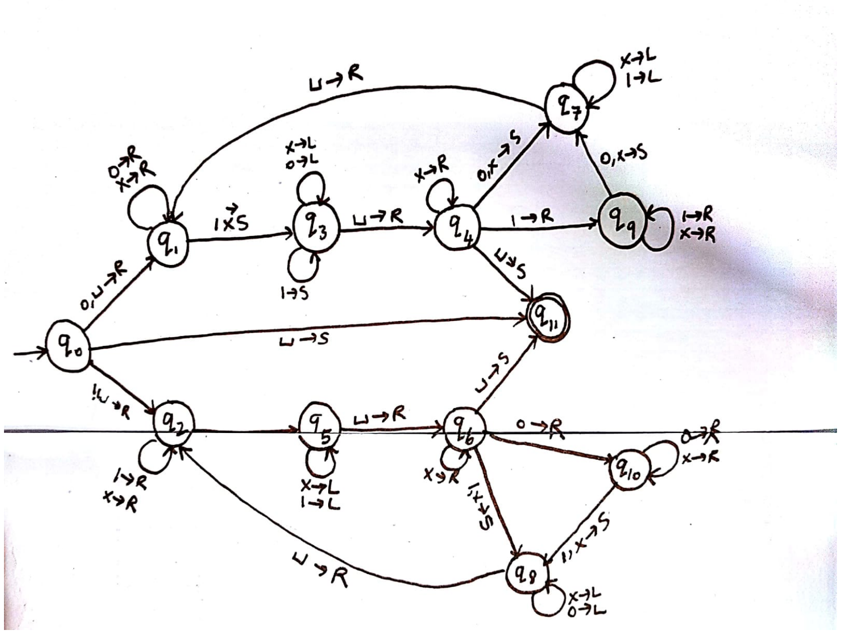

Figure 3.6: *A State Diagram of Deterministic Turing Machine for Problem 3*

For a given input, the first character will always be marked with a blank character which will be used as a marking point denoting the front of the input string when traversing the string back and forth. So for every 0 it will traverse to find matching 1 or vice versa depending the first character of the string read. Intermediate steps will involve marking off every first 0 or 1 found as X. Then the machine will head back to the front of the string to read again since the arrangement of 0 and 1 can be completely random. When it encounters no other character to match itself with the whole string will be rejected.

Theoretically, the number of steps to reach a character that will be marked will be repeated due to traversing back to head. Every character in the string will be read this way meaning that the total number of steps will be

$$(\sum_{i=1}^{n-1} i) + n$$

where the last n steps will be applied in the end: where the machine will traverse the whole string, all marked, to find the accepting state. So the closed form of this summation will give exactly quadratic, n^2. Hence $f(n) = \Theta(n^2)$.

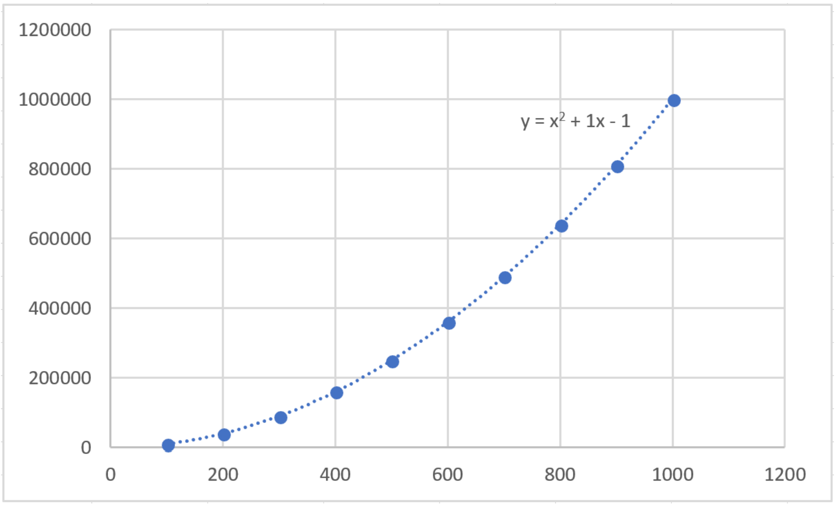

Figure 3.7: *A Plot of the number of transitions to accept inputs of varying length for problem 3*

According to the test output, the plotted graphs indeed coincide with the expected performance. According to the figure 3.7, the line of best fit matches very closely to quadratic with coefficient of x^2 being 1. Similarly, the graph in figure 3.8 displays a straight line as the line of best fit with its power close to 2.

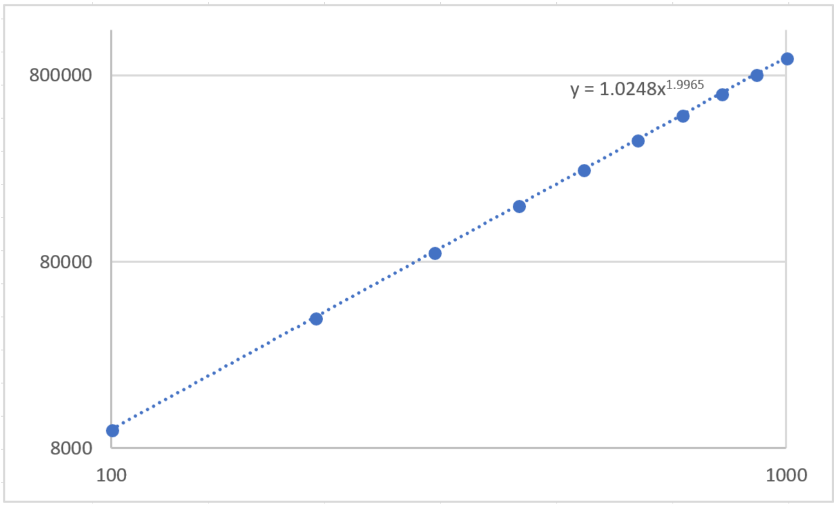

Figure 3.8: *A Log-Log Plot of the number of transitions to accept inputs of varying length for problem 3*

3.6 Problem 4: $a^ib^jc^k$

The second additional problem I have come up with is the TM which can recognise strings of the language $\{a^ib^jc^k | where i \times j = k$ for $i, j, k \geq 1\}$.

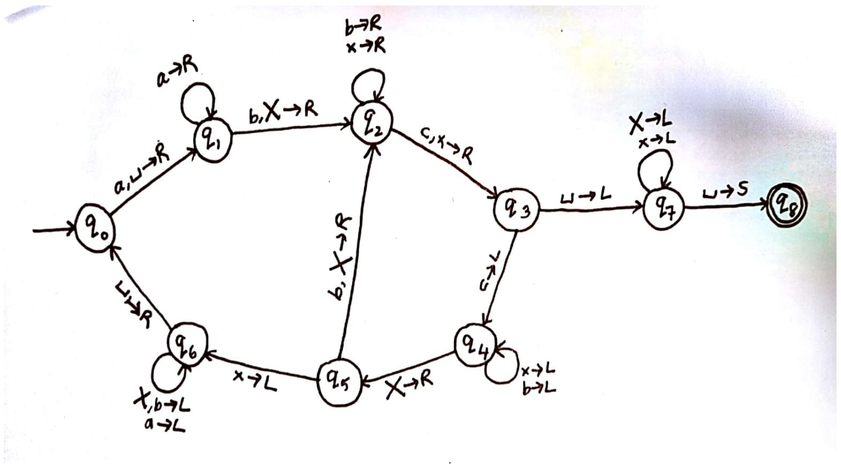

Figure 3.9: *A State Diagram of Deterministic Turing Machine for Problem 4*

For every a read, as many c will be marked off as b available. When all the b are marked, the head will traverse back to restore; while c will be kept as being marked. When there exists any remaining c having marked all b and a, the string will be rejected. Similar to the previous problem, on a single tape, this requires marking off read characters to keep track of which position for each a, b and c the current head should be. Two different marking characters are used for b and c while blank character is used for a. Hence, substantial number of traversal is executed.

Note that the total length n will then be $i + j + k$ where $k = ij$. Considering that we will be looking at asymptotic complexity this form can be simplified by assuming i = j, which will give $n \approx 2m^2 \therefore O(m^2)$. Then the following factors will contribute to the total number of transitions.

$$f(n) \approx m(4(m + 1) + m) + 2m(m + 1) + (m^2(m^2 + 1))$$

The inner $4(m + 1)$ denotes number transitions made for marking off b and c. Then m added on is the steps required to restore back entries with b prior to the next reading of a. This will be done another m times that is by the number of as exist. The one of $m(m + 1)$ denote the last traversal of the whole string before finishing to reach the accept state. The other is the number of transition of as while we go back and forth. The last term denotes the intermediate traversals of crossed off cs which should be read as the head travels from c to b. Therefore, this can be rewritten as

$$f(n) = O(m^4) = O(n^2)$$

This was a problem where the experimental output did not coincide with the theoretical estimation to some extent. In retrospect, this may be due to that strings being generated with random length may have exceeded the estimated length n for each execution. However, the figure 3.10 does show that for such exhaustive algorithm, the number of transitions can easily explode to polynomial.

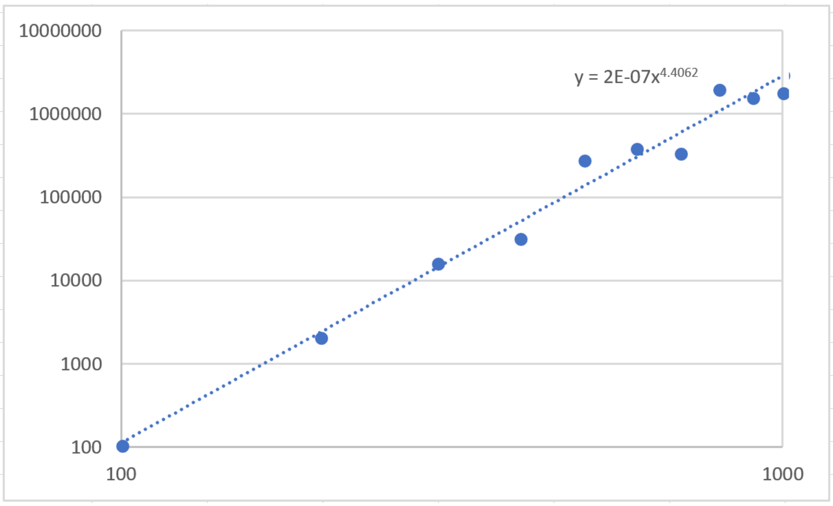

Figure 3.10: *A Log-Log Plot of the number of transitions to accept inputs of varying length for problem 4*

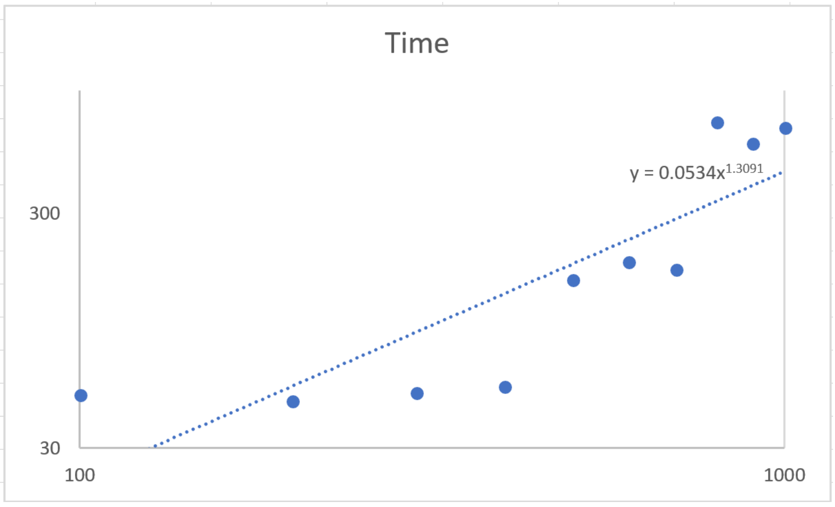

Figure 3.11: *A Log-Log Plot of the execution times for strings of varying length n for problem 4*

Since this was the case, the average of execution times for each length n is also explored and plotted as shown. By looking at the time graph, it does agree with the theoretical estimation, having a quadratic upper bound. However, it shows large variance of the points where some data points located in other areas might have easily change the line of best fit.

Summary of Performances

Overall, for all four problems, testing done on input strings of varying lengths show that the algorithms not only recognise valid strings but also reject any which are invalid. The latter is evident by the tests which verify the output with the expected results.

4
Task 4: Non-deterministic Turing Machine

mplement a simulator for a Nondeterministic TM, and demonstrate its power by programming either palindrome recognition or recognition of some other language by a Nondeterministic TM.

4.1 Design and Implementation

As an extension, the basic design agrees with that of Deterministic Turing machine as explained in Task 1. It consists of a NDTMReader and a NDTM for reading description file and processing input by simulating Non-deterministic Turing machine respectively. It is assumed that the input TM description file will have the same format, except now there may exist multiple transitions with the same pair of current state and input symbol.

Non-deterministic Turing Machine Reader - NDTMReader

The major difference between the two implementations would be that for Non-deterministic Turing machine, the hash map containing the transitions will take Node as a value of a key instead of a Pair: so that incoming state and symbol will be the parent node while multiple transitions with the given pair of inputs will be its child nodes. Node is used as part of my implementation since this can be later utilised in the processing step.

Non-deterministic Turing Machine Simulator - NDTM

The processing method takes a different approach to the way a string would be processed under determinism. The logic behind the method is a backtracking algorithm of a computation tree. It will be using a multi-tape implementation where one tape will be the processing tape, and another which will keep track of the position (i) where i denotes i+1 th transition / child node at every level of the tree. The second tape will be called the address tape.

During the very first run of processing in NDTM, it will select the first child node to move next whenever multiple choices are encountered. It will continue until it fails or accepts. From this time onwards, the method will restore its state before the last transition and continue. Hence, in the duration of the processing, the computation tree will be built, starting by setting the start state and start symbol as its root node. It will continue trying out by traversing to different child nodes: then it will halt and accept when it faces an accepting state or, it will reject if there exists no more child nodes to try and the address tape reaches its head on the first cell, indicating the root.

4.2 Problem 1: Palindrome Revisited

The palindrome problem is also solved using the Non-deterministic Turing machine simulator. The TM description file is written noted as “tm4.txt”. The non-deterministic version of the TM algorithm is shown as a state diagram below.

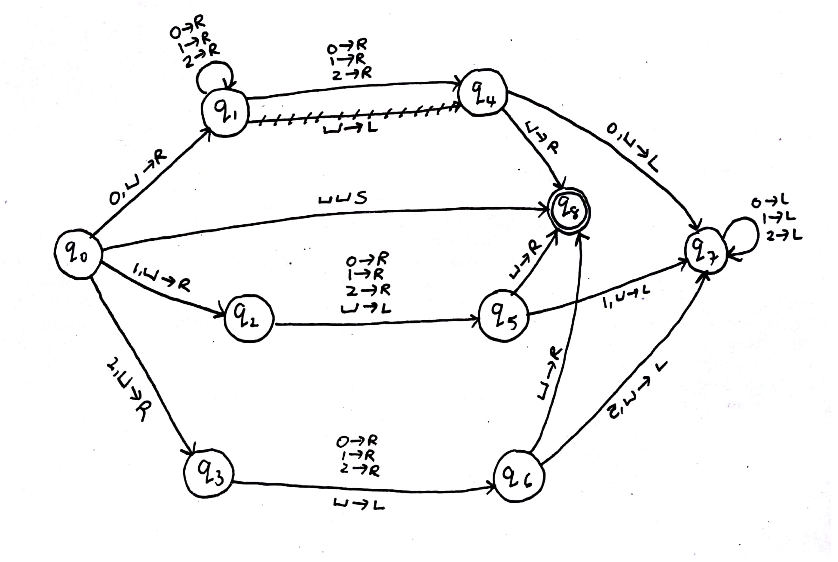

Figure 4.1: *A State Diagram for Non-deterministic Turing machine accepting palindromes*

The difference in the design is that, referring to the figure 3.1, states q1, q2 and q3 will also take the transitions involving iterative loops of reading 0,1,2 which are originally taken by states q4, q5 and q6. Additionally, the accepting state transition has been shifted thereby the total number of states has reduced.

Comparison

In order to compare the performance of the two types of Turing machines, they are tested on the same input strings and the resulting execution time and the number of transitions are recorded for every length n.

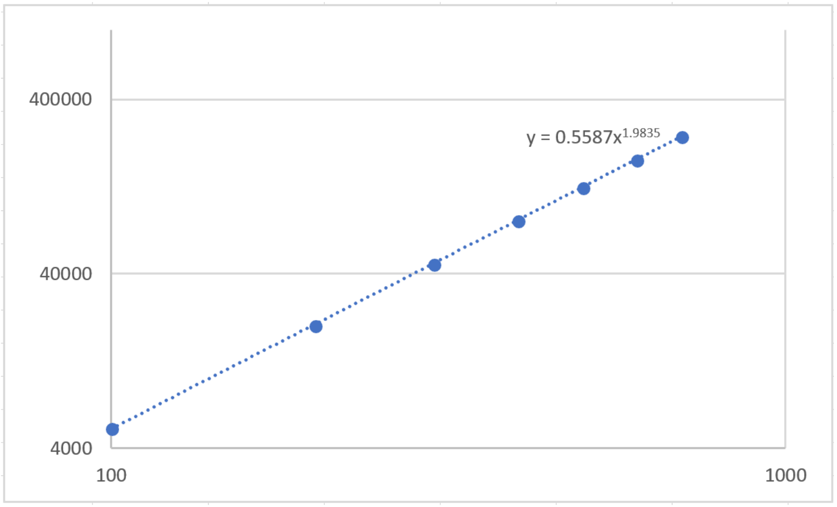

Figure 4.2: *A Log-Log Plot of Deterministic Turing machine solving Palindrome problem*

The csv outputs can be reproduced by running **mvn test** and **mvn test -Dtype="nd"** commands. The range of values tested are 100 to 700 with step size of 100 and 20 repetitions. The following plots demonstrates its performance.

As shown by the two plots, it is evident that the experimental results coincide with the statement that non-deterministic Turing machine is as powerful as deterministic equivalent: the order is very close, so as the coefficients. As analysed in the previous section on Palindromes, the figure 4.3 also displays asymptotically quadratic time complexity as upper bound. Hence, this shows that guessing upon making decisions won't affect the complexity in extreme way such that it could over- or under- perform compared to its deterministic equivalent.

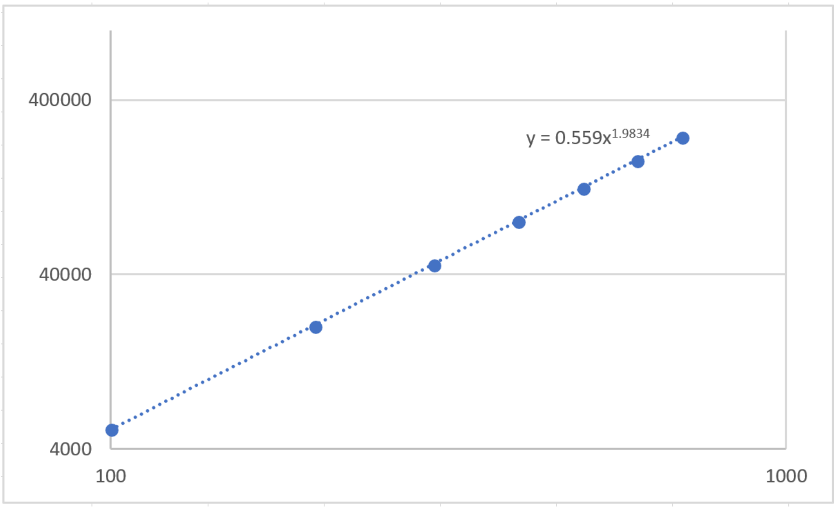

Figure 4.3: *A Log-Log Plot of Non-deterministic Turing machine solving Palindrome problem*

4.3 Conclusion

Overall, this practical involved designing Turing machine simulatorsto solve various decision problems and analysing and comparing theoretical and experimental complexity of Turing machines. At the end the implementation of non-deterministic Turing machine simulator and the comparison of its performance with deterministic Turing machine have drawn to the conclusion that they have the same computational power.

Bibliography

[1] https://www.youtube.com/watch?v=Ynd5on9g6Q8