Computational Complexity Practical 1 - Turing Machines : Implementation and Analysis

150000845

February 2017

1 Introduction

The aim of this practical was to design and implement a program to implement Turing machines based on input descriptions. Further to this, we had to come up with Turing machine descriptions for testing if a string is a palindrome, testing if a number was the result of binary addition to two given numbers beforehand and also two other Turing machines of our own. Finally, we were tasked with running experiments and analysis on different Turing machines to investigate how they perform.

In his 1936 paper *On Computable Numbers, with an Application to the Entscheidungsproblem*, Alan Turing devised the Turing machine as an abstract mathematical concept to solve the Entscheidungsproblem (*"Decision Problem"*) posed by David Hilbert in 1928. The machine consists of a collection of abstract states, a theoretically infinite tape and a read/write head over the tape. It operates by transitioning between states using a set of rules defined by the transition function. The input for this transition function is the current state and the symbol of the current tape segment under the read/write head, and the output is the next state to transition to, the symbol to write onto the tape segment and a direction of left or right for where the read/write head should next go. The collection of states include one accept and one reject state, and once either of these has been reached the process terminates.

The machine can be used as a description for any algorithm. Since the read/write head gives it the capability of going back and forth through memory freely and being able to read and change values as the algorithm chooses, it is more powerful compared to a finite state automaton and even a pushdown automaton, since the pushdown automata can only reach the bottom of the stack of input symbols by pushing away and effectively forgetting all other symbols that come after it. With this power, the Turing machine is capable of recognising non context-free languages.

2 Design

2.1 Turing Machine Builder

The Turing machine builder is required to parse an input file that describes a Turing machine and convert it into a Turing machine object. My parser will work using the file input format by :

- Take the first line of the input file and read the number of states from it.
- For the number of states defined in the previous line, create new state instances and take their labels from each line that comes after the number of states line.

- Take the alphabet line and convert it into an array list to add to the Turing machine object. This can be used to check whether a symbol on the input tape is within the tape alphabet of the machine or not later. We also add the blank character to the alphabet.
- We figure out the number of transitions in the machine using (number of states - 2) * length of tape alphabet, since the accepting and rejecting states will not have any transitions going out of it.

Since large machine files can be difficult to read and write without blank lines and can become difficult to understand, I will implement a system where blank/whitespace lines and lines starting with '#' will be ignored by the parser to allow for blank lines and comments in machine input files.

2.2 Turing Machine Models and Definitions

2.2.1 Check for Palindrome

- High Level description - From the start symbol, check the symbol and then overwrite it with the blank. Then go to the next blank at the end of the word by traversing right on the tape and check the letter before it the blank by moving once left. If they are the symbol is the same as the first one then overwrite it with blank, check the symbol to the left and work towards the front of the word by moving left to see if the second letter in from the left is the same as the second letter in from the right. In both instances, if the letters differ, then go to reject state. If we keep going and never find any reject states, then we'll eventually find 2 x's next to each other and can accept.
- Implementation Level description

  1. Read first letter and cross it off.
  2. Navigate along to the end of the word i.e until we find a cross or blank space, and move inwards to get next letter.
  3. If this letter matches the first letter, then cross it off and move left. Otherwise, move to reject state.
  4. Check this letter and cross it off. Move left until we find the first cross we made and move right by one to get the second inmost letter.
  5. If this letter matches the other letter then cross if off and move right. Otherwise, reject.
  6. Repeat the above sequence until after you cross of a letter and move in another direction you land on another cross.

2.2.2 Binary Adder

- High Level description - Read the first number and eliminate it with a '|'. Go to the first number in the next number i.e after the '#'. If these 2 numbers add up to more than 1, then replace that character with a 'c'. Otherwise, cross the number off. Then check the first number after the next '#'. If the 2 numbers together that make a binary number that ends in 0 and we encounter a 1, or we add 2 numbers together that make a binary number that ends in 1 and we encounter a 0, then reject. Otherwise, cross this number off. Go forwards until we find either a 1 or blank. If we find a blank, then go back and make sure that there are no more numbers to use in either of the two numbers to add and we can accept. If its not a blank or there are still numbers to use, go back to the last '|' and repeat the above process.

If we encounter a 'c' at any point, then we need to check the number that comes after it to determine the carry process - if the first number + the second number are equal to 0 then we need to look for a 1 in the corresponding number in the result number, otherwise we look for a 1. If the carry comes before a hash, then we may need to move the hash forward one character into the result number provided that its crossed off.

- Implementation Level description :

  1. Check first digit of first number. Write a '|' character over it to denote where we stop going backwards later.
  2. Move to the '#' and get the first digit after it. If the two digits add up to a binary number that ends in 0 then we expect to see a 0 in the result digit. Otherwise, we expect to see a 1. If the sum of these first 2 numbers is greater than 1 then we write over the number with a 'c' so that we remember to carry next time. Otherwise, go over it with 'x'.
  3. If we need to carry, i.e we read a 'c', we need to flip our expected results, using the number after the 'c' as the digit to compare with - i.e, if the first number is a 0 and the number after the 'c' is a 0, then we expect to see a 1 in the result number, as the carry adds a 1. Otherwise, expect a 0.
  4. Get the result number to compare with by going to the first digit after the next '#' and after all the x's that come after it. If it does not match our expected result, then reject. Otherwise, cross it off and move forward until we either encounter a 1 or blank character.
  5. If we encounter a blank character then we move left on the tape until we reach the previous '#'. If we check the symbol before it and its a cross, it means the whole number has been used and this number has nothing else to add and we repeat the process for the first number. Is the first letter doesn't have any symbols left between its last '|' and the hash between the first and second numbers, then we accept, as we have added everything correctly with nothing left to add.
  6. If we encounter a 1 before the blank character or there are still numbers left to use, then we go to the most recent '|' character and restart the addition process.

2.2.3 Bubble Sort

- High level description - Get the first symbol. Go to the next symbol. If this is alphabetically out of order, then we write over it with the previous letter, go back to the previous letter and write over that with the letter we just changed and keep going. We must remember that we have made a swap. If we do not make any swaps and we reach a blank character then we reach the accepting state. We only reject if there is an unexpected letter somewhere.

- Implementation level description :

  1. Get the first symbol and move right along the tape.
  2. If this symbol is alphabetically the same or lower than the previous symbol, then we keep searching until we find one out of order or we reach the end of the word.
  3. If the symbol is alphabetically before the previous symbol, then we need to swap them around and make sure we know whilst traversing the rest of the word that the symbol is out of order. Write over that the current symbol with the previous symbol. Go back to the last tape segment and overwrite the symbol on that tape segment with the symbol from the segment before. Keep traversing the word, making any other swaps.

4. If we reach the end of the word and we find that there were no swaps during that traversal of the word, then move to the accepting state. If we made swaps, then go back to the start of the word and repeat the above process until a traversal with no swaps is made.

2.2.4 String Reverse

- High level description - Move the tape right until we reach the end of the input. Read the at the end of the string '#'. Move the tape left so we can read the first character of the reversed string and then cross it off. Move left until we reach a blank character and overwrite that blank character with the first symbol we crossed off. After this, move left until we reach the hash and then after that continue left until we find a symbol that is not crossed off i.e letter or blank. If we find a letter, we repeat the procedure as before. If we find a blank space then we can move to the accepting state.
- Implementation level description :
  1. Move the tape right until we come to a blank or hash symbol. If there is not a hash symbol at the end, write one there.
  2. Go left to the last symbol and cross it off. Then move right to the first blank character after the hash and write the last symbol in that blank space.
  3. Move left back to the hash symbol and from there to the first symbol before the hash that is not a cross. If this is whitespace, then move to accept state. If it is another letter, then repeat the process of crossing it off and writing it on the next blank symbol after the hash.

3 Implementation

3.1 Build and Run Instructions

The build and run instructions are kept in the file "README.txt" in the main directory. When the project is built, all Turing machines are tested with unit tests to make sure that they work correctly, using the contents in "TestFiles".

3.2 Project Structure

All of the code is contained in the "src" directory of my project. I have compiled the JavaDoc of my source code into the "Docs" directory so that it can be viewed easily for project structure. The instructions to view it in a web browser can also be found in the "README.txt" file.

3.3 Turing Machine Implementations

The solutions to each Turing machine problem that I had to solve can be found in the "TestFiles" directory and in the appropriate sub directory in the file "machine_input.txt". For example, to find my configuration for the binary addition, it is in "TestFiles/BinaryAddition/machine_input.txt". I followed my strategies in my design for each problem and also implemented commenting and whitespace on the machine input files so that they can be read more easily. I have used these comments to explain the functionality of individual states in the configuration files.

4 Tests and Results

4.1 Palindrome

4.1.1 Complexity Analysis

Every iteration of the palindrome goes from one end to the other, each time getting shorter. The first iteration will traverse n segments of the tape, and then the second iteration will traverse $n-2$ segments of the tape, until it traverses 1 segment to find a cross on the next segment. If we work out this sum, it becomes :

$$n + (n-2) + (n-4) + ... = \frac{1}{2}\sum_{k=1}^{n} k$$

We can take this further by using the identity :

$$\sum_{n=k}^{n} k \equiv \frac{n(n+1)}{2} = \frac{1}{2}n(n+1) = \frac{1}{2}(n^2+n)$$

and so therefore the process can be further summed up as :

$$\frac{1}{2}\sum_{k=1}^{n} k = \frac{1}{2}(\frac{1}{2}(n^2+n)) = \frac{1}{4}(n^2+n) = \frac{1}{4}n^2 + \frac{1}{4}n$$

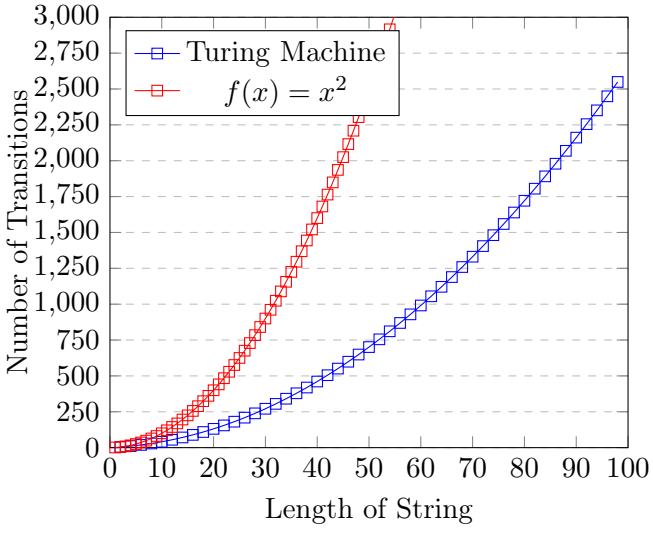
Using this, we can deduce the time complexity as :

$$O(\frac{1}{4}n^2 + \frac{1}{4}n) = O(\frac{1}{4}n^2) = O(n^2)$$

4.1.2 Test Results

To generate test data for the palindrome Turing machine, I created random palindrome strings of varying even lengths between 1 and 100 - they are all acceptable palindromes. I piped the output program i.e the test data, into "TestData/palindromeData.txt", which is included. For each length of string, 10 strings of that length were generated and run in the machine and the mean was taken for the final result.

Number of Transitions needed for Turing machine to Complete as String Length Varies



4.1.3 Remarks

As we predicted, the number of transitions needed increases quadratically as the string length increases. We can see from the trend too that it increases at a slower rate than $f(x) = x^2$. Our calculations predicted this, as we saw a $\frac{1}{4}$ constant being multiplied with our n^2 value, which explains the difference between the two curves.

4.2 Binary Addition

During every iteration of the binary addition process, the Turing machine transitions continuously right to compare 2 numbers to the result, and the transitions left back to the start. At the start, we traverse through more elements of the first number and very few of the result number, meaning that supposing that every number in the string, including the result, is roughly the same length, we only traverse $\frac{2}{3}n$ at each traversal, since we also need to traverse the whole of the second number. Therefore, a run of around

$$2 * \frac{2}{3} * n = \frac{4}{3}n$$

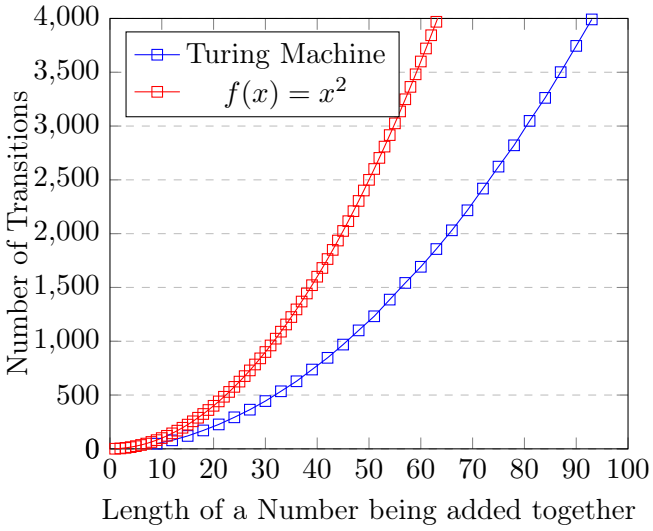
is done at every iteration. This is done until every results is found. The number of iterations we make in a situation where each number is in equal length is $\frac{1}{3}n$, since we are checking once for each digit in every number. Therefore, we can deduce the time complexity by :

$$O(\frac{4}{3}n * \frac{1}{3}n) = O(\frac{4}{9}n^2) = O(n^2)$$

4.2.1 Test Results

To generate test data for the binary addition Turing machine, I created binary additions with varying length by summing random numbers between ranges 2^n and 2^{n+1} for all $0 < n < 30$. I took these numbers, reversed their binary representations and concatenated them together with the '#' symbol so that the machine could use them. I piped the output program i.e the test data, into "TestData/binaryAdditionData.txt", which is included. For each length of string, 10 strings of that length were generated and run in the machine and the mean was taken for the final result.

Number of Transitions needed for Turing machine to Complete as String Length Varies



4.2.2 Remarks

As we predicted, the number of transitions needed increases quadratically as the string length increases. The gradient is lower than the $f(x) = x^2$ graph and this is explained by the calculations that we predicted earlier, giving us roughly $\frac{4}{9}$ the rate of growth as $f(x) = x^2$.

4.3 Bubble Sort

The bubble sort is a classic $O(n^2)$ sorting algorithm. It operates by comparing every consecutive pair in a set of data and if they are out of order then they are swapped. This is repeated until a complete traversal of the set of data has been completed without having to make any swaps. In its worst case, the sort has to traverse the set of n pieces of data $n-1$ times, making the worst case time complexity $O(n^2)$. At every iteration of the bubble sort, the element with the greatest weight in the list that is out of place is put back into its place, guaranteeing that at least one element is corrected at every iteration. For example, in the first iteration, the element with the greatest weight is guaranteed to end up at the end of the data set at the end of iteration. The average time complexity would be where $\frac{1}{2}n$ iterations have to be made and so can be expressed as :

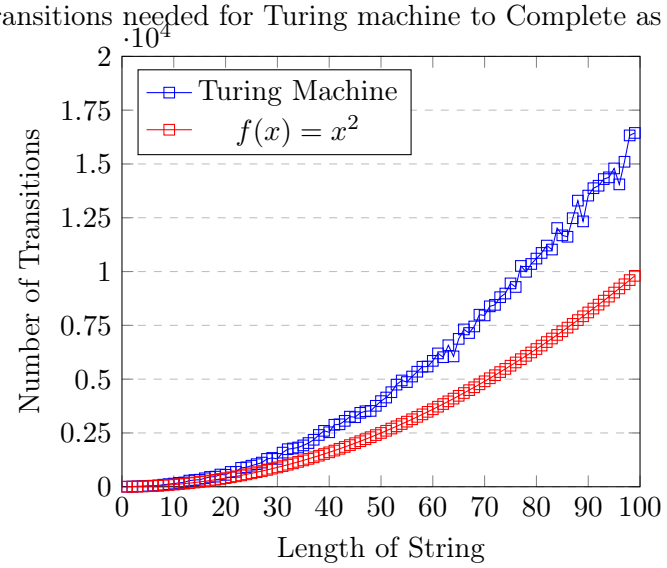
$$O(n * \frac{1}{2}n) = O(\frac{1}{2}n^2) = O(n^2)$$

The best case would be where only one pass has to be done over all the elements, meaning the time complexity comes out as $O(n)$.

4.3.1 Test Results

To generate test data for the bubble sort, I created random strings of increasing lengths within the alphabet and made my Turing machine sort them. I piped the output program i.e the test data, into "TestData/binaryAdditionData.txt", which is included. For each length of string, 10 strings of that length were generated and run in the machine and the mean was taken for the final result.

Number of Transitions needed for Turing machine to Complete as String Length Varies



4.3.2 Remarks

As we predicted, the number of transitions needed increases quadratically as the string length increases. Although the gradient isn't what we expected, it can be explained by the fact that for each pair that needs to be swapped in the Turing machine bubble sort, 2 transitions are done to backtrack once and then get back to the space, which will cause some overhead.

4.4 Reverse String

The reverse string operation works by traversing to the '#' which is at the end of the tape, which is an $O(n)$ operation. From there, it goes back and forth between the string to reverse and the new reverse string on the other side of the hash. Since each time we are traversing away from the hash, then back to the hash, then away on the other side after the hash, then back to the hash, the process can be summed up as :

$$(4 + 8 + ... + 4(n-1) + 4n) = 4(1 + 2 + ... + (n-1) + (n)) = 4\sum_{k=1}^{n} k$$

We can take this further by using the identity :

$$\sum_{n=k}^{n} k \equiv \frac{n(n+1)}{2} = \frac{1}{2}n(n+1) = \frac{1}{2}(n^2 + n)$$

and so therefore the process can be further summed up as :

$$4\sum_{k=1}^{n} k = 4(\frac{1}{2}(n^2 + n)) = 2(n^2 + n) = 2n^2 + 2n$$

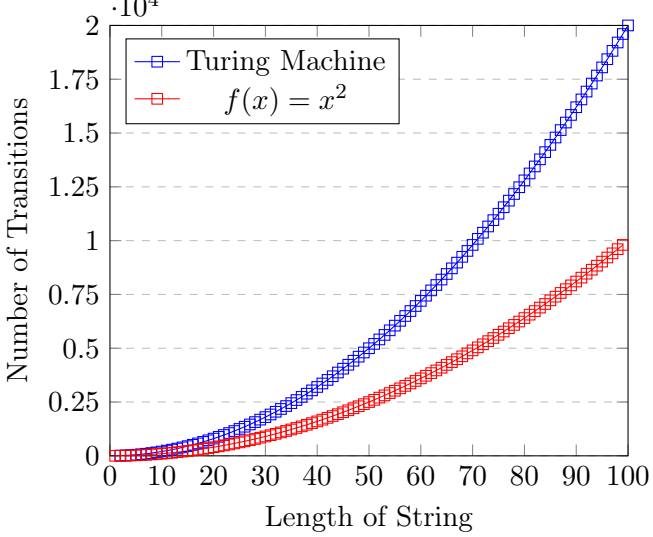
Using this, we can deduce the time complexity as :

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

4.4.1 Test Results

To generate test data for the reverse string, I created random strings of increasing lengths within the alphabet with a hash appended to it and made my Turing machine reverse them. I piped the output program i.e the test data, into "TestData/sortData.txt", which is included. For each length of string, 10 strings of that length were generated and run in the machine and the mean was taken for the final result.

Number of Transitions needed for Turing machine to Complete as String Length Varies



4.4.2 Remarks

As we predicted, the number of transitions needed increases quadratically as the string length increases. We can clearly see as well that our calculations predicted the overhead that our Turing machine would have over the $f(x) = x^2$ line too, as our gradient is greater and we predicted it would be roughly double the rate of growth.

5 Conclusion

The Turing machine as a mathematical model is far more powerful than some of its predecessors, such as the FSA. I have enjoyed working with them for this practical and greatly enjoyed running statistical analyses on the time complexities of these machines in order to deduce how these machines perform.