School of Computer Science, University of St Andrews, CS3050, 2022-23

CS3052: Practical 1 – Turing Machines

Assignment: P1 (Practical 1)

Deadline: 22 Feb 2023 at 9:00 pm

Weighting: 45% of coursework weight

Please note that MMS is the definitive source for deadline and credit details. You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturers regarding any queries well in advance of the deadline.

---

1 Objective

This practical is aimed at developing understanding of the Turing machine abstraction, by building a TM simulator and also designing a number of Turing Machines for various tasks.

Competencies Addressed

- Develop your understanding of the Turing Machine abstraction
- Develop your general programming skills
- Develop your skills in clearly and concisely reporting your work.

2 Practical Requirements

You are required to complete the first three tasks listed below. Task four is just offered for your own interest and to help your understanding of the material for the later parts of the course.

Credit: This practical is worth 45% of the coursework.

Submission: Upload a zipfile with your report in PDF and any supplementary information (code, data, spreadsheets, etc.) to MMS.

Important Note: The specification of this practical is deliberately quite long and quite precise, to allow for automated testing. It is *essential* that your code comply exactly with the specification below and that `stacscheck` is able

to build and test your programs on the School Linux systems. Submissions that do not meet these conditions will receive a reduced mark.

Update This is version 2, which was updated on 26 Jan to clarify the optional ststatus of task 4.

3 Overview and Background

In lectures you will have been introduced to the Turing machine model of computation, a historically and conceptually important model of computation. Dating to the beginnings of computer science, this model has very simple building blocks, while still being capable, in a precise mathematical sense, of all computation we know how to do. This practical is about programming this model within a programming language of your choice, simulating the behaviour of Turing machines on given inputs.

4 Specification

Task 1

Write a program that can simulate a given (one-tape, deterministic) Turing machine (or TM) on a given starting tape. Specifically, you should create a program `runtm` that takes as input a TM description file and a file describing the starting tape, and run that TM on that input. In our definition of a Turing Machine we largely follow the lectures in that:

- Our Turing machines have a "half-infinite" tape (which extends as far as needed in one direction only).
- They always begin computation with the cell at the "finite" end being current.
- Each TM has a single accept state and a single reject state, neither of which may have any outgoing transitions, and no other halt state. Note that this corresponds to an acceptor in the lectures.
- Our Turing Machines do not produce output. A given TM run with a given starting state will either accept, reject, or run forever.

You should allow TM description files in the following format:

- The first line is the word `states` followed by the number of states `n` (including the accept and reject states);
- Then there should be `n` lines, each containing the name of a state, possibly followed by whitespace and then a `+` denoting that this is the accept state or a `-` denoting that this is the reject state;

- Then a line with the word `alphabet` followed by the number of letters in the tape alphabet, followed by those letters, separated by spaces;
- Then a number of lines representing the transition table, each of which has the form

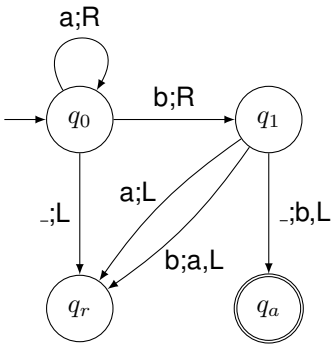
  `<state1> <tapeinput> <state2> <tapeoutput> <move>`

  where `move` is one of `L` (move left); `R` (move right) or `N` (no move). `state1` and `state2` are mentioned in `states` and `tapeinput` and `tapeoutput` are `'_'` or are mentioned in `alphabet`. `state1` may not be the accept or reject state.

By convention, the first state mentioned is the start state. Note also that we use underscore (`_`) for the blank character, which should not be mentioned explicitly in the `alphabet`.

Also by convention, if no transition is given from a particular state with a particular current symbol, a transition which leaves the symbol unchanged, moves left and enters the reject state should be supplied.

As an example, the TM



is written as:

```
states 4
q0
q1
qr -
qa +
alphabet 2 a b
q0 a q0 a R
q0 _ qr _ L
q0 b q1 b R
q1 a qr a L
q1 b qr a L
q1 _ qa b L
```

This could be shortened to:

```
states 4
q0
q1
qr -
qa +
alphabet 2 a b
q0 a q0 a R
q0 b q1 b R
q1 _ qa b L
q1 b qr a L
```

using the convention for filling in missing transitions.

Additional Details

1. Your simulator must be contained in a directory called `src` at the top level of your submission. Calling `make` in that directory with no arguments must build your program, producing a `runtm` executable in that same directory (which may be a script that runs the real executable, if that is more convenient). All names are case sensitive.

2. State names may contain any printable ASCII character and may be of any length, except that no state may be named `+`, `-` or `alphabet`.

3. The tape alphabet may contain any printable ASCII character except `'_'`

4. It is an error for your Turing Machine description file to include more than one transition from the same state on the same tape symbol (but see Task 4 below).

5. Your program must be capable of dealing with a Turing machine with up to 10 000 states and 80 letters in the alphabet and simulating it for at least 100 million steps within the normal `stacscheck` timeout of 10 CPU minutes.

6. The tokens in the Turing machine description file can be separated by any amount of whitespace.

7. Your `runtm` executable must accept either one or two command-line arguments. In either case the first is a path to a file containing a Turing Machine description. The second, if present, is a path to a file containing an initial tape description. If only one argument is given you should start the machine with an empty tape. You may assume that paths do not begin with `-`.

    - The tape file consists of letters from the alphabet of the Turing machine together with whitespace, which is ignored
    - `_` may be used to indicate a blank position on the tape

- Your program should cope with an initial tape with up to 100 000 symbols.
- If the Turing machine description or tape file cannot be opened your program should exit with an exit status of 3.

8. For parts 1, 2 and 3, behaviour with fewer than one or more than two command-line arguments is undefined and will not be tested.

9. Your program may print to standard output as many lines as you like that begin with whitespace (these will be ignored by the checker) and may print anything you like to standard error. Otherwise it *must* print lines to standard output and exit exactly as follows:

   - if it detects a syntax error in the Turing machine description or in the initial tape description it should print `input error` and exit with an exit status of 2.
   - If the TM computation completes and ends in the accept state, it should print `accepted` and then a line giving the number of steps executed (in decimal) and then the final state of the tape (in the same format as the starting tape description) and exit with an exit status of 0.
   - if the TM computation completes and ends in the reject state, it should print `not accepted` and then a line giving the number of steps executed (in decimal) and then the final state of the tape (in the same format as the starting tape description) and exit with an exit status of 1.

10. When printing the tape state, you should start at the finite end of the tape, printing any blanks (as `_`) at that end, but you should *not* print blanks after the last non-blank symbol on the tape. The one exception is if the tape is entirely blank, in which case you should print one `_`.

11. When counting the number of steps executed, do not count the "virtual" transition to the reject state supplied according to the convention above.

Task 2

Devise Turing machines to solve the following problems:

1. Recognise the language of balanced bracket sequences. That is sequences in $\{(,)\}^*$ with equal numbers of opening and closing parentheses, such that every opening parenthesis can be matched with a closing parenthesis that occurs *after* it. So ϵ, (), (()) and ()() are all in the language, but (,)(and ((((are not. The description of this machine should be in a file called `paren.tm` in the `src` directory.

2. Recognise strings from the language

$$\left\{ w_1\#w_2\#w_3 \;\middle|\; \begin{array}{l} w_1,\ w_2, \text{ and } w_3 \text{ are binary numbers, least significant bit first} \\ \text{and } w_1 + w_2 = w_3 \text{ in binary} \end{array} \right\}$$

For instance, 0#0#0, 01#1#11, and 00#111#111 belong to the language, and 0#0, 0#0#1, and 000#111#11 do not. The description of this machine should be in a file called `binadd.tm` in the `src` directory.

Also devise Turing machines to solve at least **two other problems**, of your choice. For best marks the problems should be interesting, challenging, precisely specified and clearly different from these two.

Task 3

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.

Task 4 – Optional not for credit

Extend your simulator to include a non-deterministic option. If used in the form `runtm -n <machine> [<tape>]` it should accept TM description files with more than one transition from the same state on the same symbol and execute them as non-deterministic TMs. It should print the number of time steps executed (specifically the smallest number to reach an accepting state on any path of the computation or a reject state on all paths) and the `accepted` or `not accepted` message, but *not* the final tape, since this may not be well-defined. It does not need to be able to run for 10^8 timesteps in all conditions, but should be reasonably efficient. Demonstrate the power of your simulator by programming a non-deterministic TM `repeat.tm` that recognises the language

$$\left\{ ww \mid w \in \{0,1\}^* \right\}$$

That is the language of words such as 10111011 which consist of a subword followed by a repetition of the same subword. Include this TM description in your `src/` directory.

StacsCheck Tests

An extensive suite of `stacscheck` tests is provided in the `Tests` directory and its subdirectories, covering the basic operation of the simulator, the two required TM descriptions and the extension. You will need to explain in your report how you tested the two other TMs you developed and any additional tests for

any part of your work (for instance unit tests for components of your simulator) that you found useful to develop.

5 Notes

- You are free to program in any programming language you wish, provided that `make` called with no arguments in the `src` directory can compile your programs on the lab machines and produce an executable called `runtm` (which can be a script if that is more convenient).
- A reference implementation of a Turing machine simulator meeting the given specification is provided as a Linux binary `reftm` in the `Tests` directory. It is known to run on the School system and may run on other Linux installations. This is provided to allow you to develop and test your TMs independently of your own simulator, so that you can address the tasks in any order, and to allow the `stacscheck` tests to distinguish between errors in your simulator and errors in your TMs. The `reftm` simulator also accepts a `-v` ("verbose") option. When this is given it prints every transition followed.
- Turing machines are written about, including sample programs, in many books and at various places around the web. Personally, we would advise programming your own, as understanding code from someone else properly takes a long time. That said, you are free to research online, and if you make use of resources from elsewhere, give proper citations as usual. Be aware also that there are many slightly different, but fundamentally equivalent, definitions of a Turing machine, so some descriptions you find elsewhere may use different conventions.
- Remember that being able to independently reproduce both your experiments and data analysis is fundamental to good science. Please submit the raw data, the processed analyses, and any scripts or spreadsheets you used.

6 Submission

The output of this assignment will be a **report**, which should be in PDF, together with code, Turing machine descriptions, and data as noted in the individual tasks. You are encouraged (but not required) to use LaTeX for your report.

Your **data** if any should be presented in CSV format, an accessible format which is easy to produce and consume by a variety of tools.

Make a **zip archive** of all of the above, and submit via MMS by the deadline.

7 Marking

We remind you that it is not enough for your programs to be correct, you have to convince the markers that they are correct (checking correctness of an arbitrary program is uncomputable!)

We are looking for:

- good, understandable code for the simulator, tested and commented properly;
- correct, understandable solutions of Turing machine programming problems;
- interesting and challenging problems solved;
- demonstrated insight on how Turing machines work;
- good insight into complexity analysis.

Tasks 1 and 2 form the basic requirements. It is possible to get marks up to 13 by just doing Task 1. It is also possible to get marks up to 13 by just doing Task 2. Getting marks above 16 will require work on Task 3 as well as excellent quality work on the other tasks.

In all cases, if the basic requirements are not done to a good standard, the higher marks will not be available. (For example, having a perfect Task 3 will not produce high marks if Tasks 1 and 2 are badly done).