TM: A text-based, object-oriented, Turing machine simulator

R. Matthew Kretchmar
Department of Mathematics and Computer Science
Denison University
Granville, OH 43023
kretchmar@denison.edu

Abstract

TM implements various facets of a Turing machine. The syntax for our Turing machine follows the convention adopted in *Elements of the Theory of Computation*, 2nd Ed. by Harry R. Lewis and Christos H. Papadimitriou. TM consists of a basic executable for simulating simple Turing machines, a template option to extend the flexibility of generic Turing machines, and a development environment for combining simple Turing machines into more complex Turing Machines.

1 Overview

There are seven parts to this document:

1. Where to obtain the simulator
2. Turing machine syntax
3. Tape syntax
4. The basic tm executable
5. The tm preprocessor
6. The tm development environment
7. Examples and notes

This document assumes the reader is familiar with the basic operation of Turing machines. For introductory information on Turing machines, read the Lewis and Papadimitriou text or any one of the other numerous texts on Theory of Computation and/or Formal Automata.

While the generic notion of a Turing machine is universal, there are various alternatives in specifying the syntax and operation of a Turing machines. The reader might wish to examine the Lewis and Papadimitriou text for the specific Turing machine syntax supported by this simulator. However, enough of the details of our syntax convention should be given by this document so that it is self-supporting.

2 Obtaining the Simulator

Load the following URL into your browser and follow the directions on this web page.

`http://www.denison.edu/~kretchmar/tm/`

3 The Turing machine syntax

In this section we introduce the specific syntax for our Turing machine.

A *Turing machine* is a quintuple $M = (K, \Sigma, s, H, \delta)$ where:

K is a finite set of states

Σ the alphabet, is a finite set of symbols

s is the start state ($s \in K$)

H is the finite set of halting states ($H \subset K$)

δ is the transition function where each member of δ is of the form $(s, a, (t, b))$ where s is the current state, $a \in \Sigma$ is the currently scanned tape symbol, t is the next state, and b is either a symbol to write to the tape or a movement action (left, right).

The following is an example of a Turing machine configuration file for $L_{\sqcup}$ which is a Turing machine that seeks the first blank to the left of the current head:

```
#L_.ab: scans to the left to find the first blank symbol
#       to the left of the current head position.

(
{A,B,h},
{a,b,_,^},
A,
{h},
{ (A,a,(B,<)),
  (A,b,(B,<)),
  (A,_,(B,<)),
  (A,^,(B,>)),
  (B,a,(B,<)),
  (B,b,(B,<)),
  (B,_,(h,_)),
  (B,^,(B,>)) }
)
```

Let us examine the different parts of this file. First, we note that the entire configuration is enclosed in parenthesis ( ) with five parts within to denote M as a quintuple. The first part is a set, K, given in braces. The second part is a set, Σ, again in braces. The third part is the start state, s. The fourth part is a set, H, of halting states. The final part is a set, δ, for the transition function. This syntax for specifying a Turing machine must be followed exactly; any deviation will be reported as an error. Whitespaces are allowed to be inserted anywhere in the configuration. Also, comments begin with a # so that the remainder of the line is ignored.

Let us look at the rules for each part:

K

Valid states are single characters. All characters are valid except the reserved characters of { } , _ ^ < > * and whitespaces. However, it is recommended that uppercase letters be used to denote states so that the symbols are not confused with the alphabet. The same symbol cannot be repeated in the set (sets are not allowed to contain duplicate items).

Σ

Valid symbols for the alphabet are single characters except for { } , < > * and whitespaces. When specifying an alphabet, two symbols *must* be included: the left-end of tape symbol ^, and the blank symbol _. It is an error to duplicate the same character in the alphabet set. To avoid unnecessary confusion, it is discouraged to use alphabet symbols that are also states

although this is legal.

s

The start symbol must be a member of the start state K.

H

The set of halting states must be a subset of K.

δ

The transition function δ is a set of ordered triples. Here are the different parts of each ordered triple $(s, a, (t, b))$:

- a state from $s \in K - H$
- a symbol from $a\ in\Sigma$
- an ordered pair (t, b) where
  - $t \in K$
  - $b \in (\Sigma \cup \{<, >\})$

All rules for δ of the form $(s, a, (t, b))$ must also meet the following conditions:

- $b \neq$ ^ because it is illegal to write the left-end of tape symbol.
- if $a =$ ^then $b =>$ because we must always make a right-move when encountering the left-end of tape.
- It is illegal to duplicate a rule (cannot have multiple rules with the same s and a) because δ is a function.
- It is illegal to omit a rule for some $s \in K - H$ and some $a \in \Sigma$ because δ is a function and must be fully specified.

4 Tape syntax

Here is are some example tapes:

^_aaaaaaa__

^___aaa_aa*b_

Tapes must meet the following conditions:

- Valid tape characters are restricted to members of the alphabet.

- The left-end of tape symbol, ^ , must always be the first symbol and can never be anywhere else in the tape.
- The * is optional and is used to denote the position of the Turing machine scan head. If unspecified, the head will assume to be at the left-end symbol.
- Tapes cannot contain any whitespaces within.
- Tapes are assumed to contain infinite blanks to the right even though the extra blanks have not been explicitly specified.

5 The basic tm executable

From the command line, execute the command *tm* to run the Turing machine simulator. The simulator reads two things from stdin:

1. A specification for a Turing machine
2. Multiple tapes

If the Turing machine specification is invalid, the program halts. If the specification is valid, the Turing machine then scans each tape and performs a computation on each tape. If any of the tapes are invalid (either in syntax or during execution) the Turing machine halts processing on that tape and then proceeds to the next tape.

From the unix prompt, it is natural to use concatenation, file redirection, and pipes to execute the machine. Let's assume we have the following two files:

L_.ab

```
#L_.ab: scans to the left to find the first blank symbol
#       to the left of the current head position.

(
{A,B,h},
{a,b,_,^},
A,
{h},
{ (A,a,(B,<)),
  (A,b,(B,<)),
  (A,_,(B,<)),
  (A,^,(B,>)),
  (B,a,(B,<)),
  (B,b,(B,<)),
  (B,_,(h,_)),
  (B,^,(B,>)) }
)
```

tmblank_tapes

```
^___aaaa*aaa
^_*_aaa
```

File *L_.ab* contains the specification for the Turing machine which moves to find the first blank to the left of the head. File *tmblank_tapes* contains two test tapes. We run the following command:

```
cat L_.ab tmblank_tapes | tm >output
```

The contents of the file *output* now contains the following:

output

```
ret = 0
M = (K,S,s,H,d)
        K = { A, B, h }
        S = { a, b, _, ^ }
        s = A
        H = { h }
        d =     A a -> (B,<)
                A b -> (B,<)
                A _ -> (B,<)
                A ^ -> (B,>)
                B a -> (B,<)
                B b -> (B,<)
                B _ -> (h,_)
                B ^ -> (B,>)

Turing Machine Starting Execution!
-------------------------------------------------------
Computing on tape: ^___aaaa*aaa_
(A,^___aaaa*aaa_)
(B,^___aaa*aaaa_)
(B,^___aa*aaaaa_)
(B,^___a*aaaaaa_)
(B,^___*aaaaaaa_)
(h,^___*aaaaaaa_)

-------------------------------------------------------
Computing on tape: ^_*_aaa_
(A,^_*_aaa_)
(B,^*__aaa_)
(B,^_*_aaa_)
(h,^_*_aaa_)
```

The tm command first scans the Turing machine configuration and then prints it to the screen. Then tm simulates this Turing machine on each of the input tapes. It shows the entire computation sequence for each tape.

6 The tm preprocessor

Consider these two simple Turing machines:

L.ab

```
#L.ab  moves one square to the left and halts
(
{ A, h },
{ a, b, ^, _ },
A,
{ h },
{       (A,a,(h,<)),
        (A,b,(h,<)),
        (A,_,(h,<)),
        (A,^,(A,>))}
)
```

L.01

```
#L.01  moves one square to the left and halts
(
{ A, h },
{ 0, 1, ^, _ },
A,
{ h },
{       (A,0,(h,<)),
        (A,1,(h,<)),
        (A,_,(h,<)),
        (A,^,(A,>))}
)
```

These two machines perform exactly the same task: move one square to the left and then halt. The first machine operates on an alphabet of $\{a, b\}$ while the second machine operates on the binary alphabet, $\{0, 1\}$. It seems redundant to store both these machines and other machines that simply move one square left for different alphabets. Thus, we introduce a Turing machine template:

L.i

```
#L.i  moves one square to the left and halts
(
{ A, h },
{ *, ^, _ },
A,
{ h },
{       (A,*,(h,<)),
        (A,_,(h,<)),
        (A,^,(A,>))}
)
```

Notice that this machine is exactly the same as the previous ones except that it's alphabet is limited to an asterisk. The * is a wildcard that is used to *instantiate* a template with a specific alphabet.

There is a preprocessor called *instantiate* which reads from stdin (i) a specific alphabet and (ii) a Turing machine template, and then instantiates that template with the specified alphabet; it prints the instantiated Turing machine to stdout.

Notice that the template must still contain ^ and _ and that the specific alphabets cannot contain these elements. It is helpful to create a file for each standard alphabet. We have created two files:

alphabet.ab: $\{a, b\}$

alphabet.01: $\{0, 1\}$

which both conform to the standard alphabet syntax and rules (except that they contain no ^ nor _.

We then use the following commands to create $L.ab$ and $L.01$

```
cat alphabet.ab L.i | instantiate > L.ab
cat alphabet.01 L.i | instantiate > L.01
```

You can also specify wildcards as the next symbol. Here is the template for $Ln\_$ (move left to find the first non-blank to the left of the head):

Ln_.i

```
#Ln_.i: scans to find the first non-blank symbol to the
#       left of the current head position.
(
{A,B,h},
{*,_,^},
A,
{h},
{ (A,*,(B,<)),
  (A,_,(B,<)),
  (A,^,(B,>)),
  (B,*,(h,*)),
  (B,_,(B,<)),
  (B,^,(B,>)) }
)
```

The power of this feature is limited. It is only possible to use templates if all the members of the alphabet will have the same rules; yet, it does help to reduce the redundant storage of simple Turing machines for different alphabets.

7 The tm development environment

Simple Turing machines get to be large and unmanageable Turing machines without adding too much more complexity; it can be difficult and burdensome to debug a machine which seems "simple" in principle but can be large and complex in the formal syntax. To overcome this problem, Lewis and Papadimitriou have introduced a short-hand notation for combining simple Turing machines into more complex machines.

The basic idea is to accumulate a library of useful simple Turing machines specified in formal syntax. Such machines include move left, move right, move left to first blank, write an "a", and so on. It is not hard to construct a library of these simple machines.

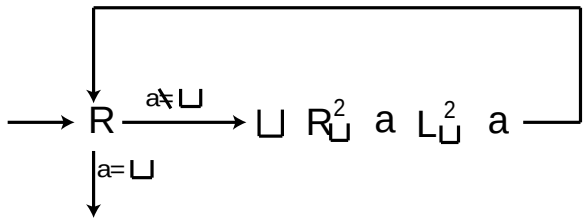
Then we create an input tape. The tape "object" is then handed off to a series of these simple Turing machines each of which performs some manipulations on the tape. When one machine halts, the execution thread and the tape are passed along to the next machine. In this way, we can "program" some rather complex Turing machine behavior by combining simple machines. We have implemented this same feature in our TM simulator.

In order to use this feature, you will have to write a small C++ program and compile/link with the tm simulator code. Fortunately, there are only a

handful of routines that you must call to implement this feature; building large Turing machines is rather intuitive in this development environment.

On page 190 of the Lewis and Papadimitriou text is the short-hand notation for the *copy* Turing machine. This machine will take the string just to the right of the head and make a duplicate copy of the string to the far right.

Here is roughly the diagram from the text:



This machine first moves one square to the right. Then, if the next character is not a blank (store this character in the *variable* a, write a blank, move to the right to find the first blank (twice), write the symbol in the variable a, move the left to the first blank (twice), write the symbol in a again, then move to the right and repeat.

This *copy machine* uses several simple machines including: R: move one square to the right, $\sqcup$: write a blank, $R_{\sqcup}$: move the right to find the first blank to the right of the head, and so forth. We have already built these simple machines by specifying their operation using the formal syntax of Section 1. Now we combine them into a single machine. Here is the C++ file that implements the copy machine:

simple.cc

```
#include <stdio.h>
#include "tape.h"
#define TM(s)    if ( (ret = tp->scan_tm(s)) != 0 )  return ret; else
#define READ     tp->read()
//============================================================
int   copy ( tape *tp );
//============================================================
int main ( int argc, char *argv[] )
{
   int       ret;
   tape      tp;

   ret = tp.scan();
   tp.print();
   printf("\n");

   ret = copy(&tp);
   return 0;
}
//============================================================
int   copy ( tape *tp )
{
   int   c;
   int   ret;

   TM("R.01");
   while ( (c = READ) != '_' )
   {
      TM("W_.01");
      TM("R_.01"); TM("R_.01");

      if ( c == '0' )   TM("W0.01");
      else              TM("W1.01");

      TM("L_.01"); TM("L_.01");

      if ( c == '0' )    TM("W0.01");
      else               TM("W1.01");

      TM("R.01");
   }
   TM("L_.01");
   return 0;
}
```

Here we have enclosed the entire copy machine into one subroutine called *copy*. We pass a tape pointer into this routine and the routine will copy the string on the tape. There are a few key features used in this file:

- We must include "tape.h" so that the tape object can be implemented.
- We call

  ```
  tp->scan()
  ```

  to scan the input tape from stdin.
- We call

  ```
  tp->print()
  ```

  to print the scanned tape to stdout.
- We make repeated calls to

  ```
  tp->scan_tm(‘‘tmstring’’)
  ```

  where "tmstring" is the name of a file which contains the specification for a simple Turing machine. By making this call, we load the Turing machine and then execute the Turing machine on the current tape configuration. Notice that the Turing machine head is denoted by a * on the tape and hence the "head position" travels with the tape object and not each individual Turing machine object.

Notice the use of macros (defined at the top) to simplify the calling mechanisms for loading Turing machines from file and for reading the symbol currently under the head. Using these macros presupposes that your subroutine has an int named *ret* and a pointer to the tape named *tp*.

After creating *simple.cc* we build the unit with: *make*. You may have to adjust the makefile if you want to call your source file something other than *simple.cc*.

We then execute the machine as:

```
simple <tape1 >out
```

where *tape1* is a file that holds the desired input tape and *out* holds the output of the program computation. The output will show the syntax for each machine being loaded and then will show the transitions each machine makes on the tape – it is a lengthy output file.

The *out* file from above is not shown here (it is 900 lines long) but can be found on the website.

8 Examples and notes

There is a website for this simulator at:

`www.denison.edu/~kretchmar/tm/`

At this website, you can find the following:

tm The basic executable for the formal Turing machine syntax.

instantiate The template preprocessor.

tm.ps This document.

tm.tar.Z The makefile, object files, and some source files. These are necessary for the development environment (simple.cc).

stuff.tar.Z A library of simple Turing machines. Each simple machine is a textfile following the formal notation. Also included are a few tape files and two alphabet files.

Practical Limitations

Although an idealized Turing machine can have arbitrarily large (but finite) state sets, alphabets, and transition functions, we must place artificial limitations on the amount of memory allocated to a tm object. Below is a list of these restrictions. They can be expanded, if necessary, by modifying the appropriate #define in the corresponding header file and then recompiling the executables.

- The tape length limit is set to 100.
- Each set (states, alphabet, ...) is limited to 100 elements.
- The transition function is limited to 500 elements (rules).
- Turing machines halt after a maximum of 100 execution steps – to prevent infinite loops.