TuringSim - An Universal Turing Machine Simulator for iOS

Cristian Stanescu

1. Introduction

The Turing Machine is a great model for designing general purpose algorithms. Due to its simplicity and flexibility, such a model offers an easy way to design and test any computable algorithm.
Often Turing Machines are represented by their instructions table, which is a good way for understanding each single instruction, but is hard to read and understand effectively *what* these instructions really do. Turing Machines models are finite state machines, which can be represented using a graph diagram too, where connections are interactions between different states, thus describe the instructions executed by the Turing Machine itself.
The present document will describe how the software allows the user to define such a model and how the defined Machines are run by the software.

2. Why iOS

The iOS platform allows a great user interaction with the device's user interface, allowing complex and otherwise unfriendly actions to be taken through simple gestures on the screen. The purpose of this project is to allow the user to create diagrams that represent a complete Turing Machine model with ease and clarity, without loosing any expression power given by such a model. iOS allows this kind of creation on a common use device like a smartphone (or tablet) and the frameworks given by the operating system allow the user to easily share, store, reuse and distribute the documents created in the simulator. The gestures, which have become an intuitive way to give an input to the using device, allow the user to easily customize, modify or enrich the current diagram he is designing. The rich UI is a perfect way to represent the current state of the currently designing diagram without loosing in expression power of the Turing Machine model.
There is no freely downloadable Turing Machine simulator present on the AppStore®, therefore this project aims to give a free solution for educational or experimental purposes for iOS users.

3. The Turing Machine

A Turing machine is formally defined as a 7-tuple $T=\{Q,\Gamma,b,\Sigma,\delta,q_0,F\}$ where

- Q is a finite set of states (non-empty)
- Γ is a finite set of the tape alphabet (non-empty)

- $b \in \Gamma$ is the blank symbol (“ ”)
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of input symbols
- $q_0 \in Q$ is the initial state
- $F \subseteq Q$ is the set of accepting states (final state included).
- $\delta : Q \setminus F \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ is the transition function, where L is left shift, R is right shift. (the presented implementation adds a third element, ‘End’ for defining the final transition)

4. The software’s main concepts

4.1 The user interface

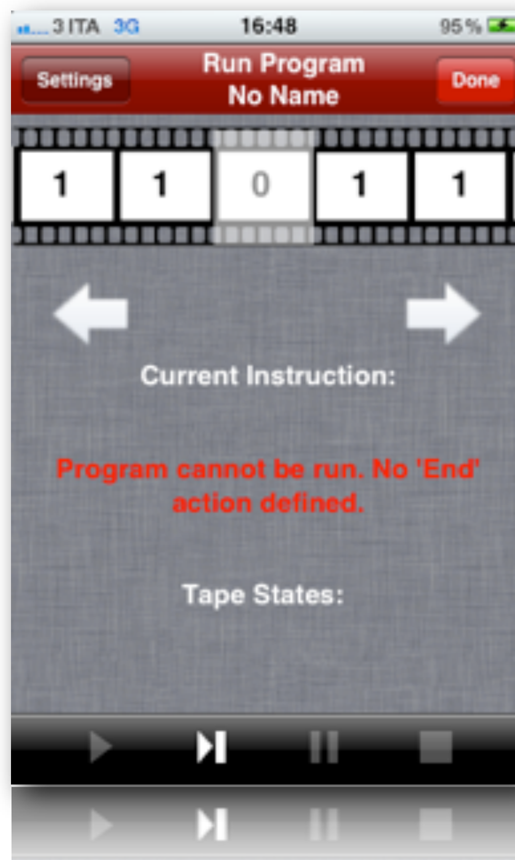
The simulator presents three different sections to the user. The first section is dedicated to the design and execution of the Turing program. The second offers a web repository where each user can publish the programs that were designed and download those made by other users. The third section is an informative one about Alan Turing, the main ideas of the Turing Machine model and an option for the user to communicate directly with the developer for suggesting new features or reporting bugs.
The main section is the core of the presented application.

By tapping the “Load” button the user can choose between two options: loading a saved program or starting to design a new one, which is the initial situation presented to the user.

The green action button allows the user to perform different actions related to the designing of the Turing Program.
The following menu is presented once the action button was tapped.

The “Add State” button allows the addition of a new state, represented by gray circle containing the label associated to the added state (“S{state index}” is the default label).
The “Settings” button brings the user to the settings pane for the current App, shown in the picture below.

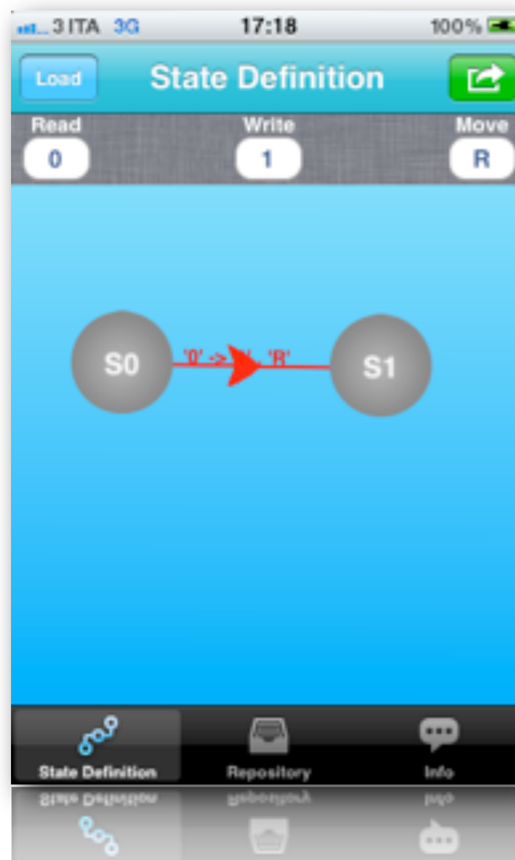
Here the user is able to assign a custom name to the current program, modify the alphabet it is using, the initial tape status and define any debugging options.
The current program can be saved by tapping on the “Save” button, sent as an e-mail attachment by tapping on the “Send” button, printed using the “Print” button (if a compatible printer is available on the current local network). The “Publish” button allows the user to send the current program to the community’s web repository, allowing other users to download, reuse and rate the current program.
The “Run” button brings the user to a different view shown in the following picture.



In this view the user is able to see the initial tape status, move left or write to place the header on the desired tape cell and control the program's execution flow (run the program, pause it, stop execution or execute step by step). The view gives the user information about the program's executability and eventually the errors found during the program execution. While the program is being executed the tape changes dynamically moving left or right.

4.2 Designing a program

After tapping on the "Add State" button, a gray circle representing the newly added state is drawn on the screen. After adding a second state one can connect them by holding the initial state button down until a white image fades in and the state circle becomes green, this means the state is ready to be connected to another state. The connection is done by selecting the destination state. A black arrow will be drawn indicating the function associated with the corresponding action. Tapping on the arrow allows the user to redefine this function, like shown in the picture below.



In the upper part of the screen the user is able to define which symbol should be read for the action to take place, which symbol should be written on the tape and the direction to take after the writing operation is complete. For each of these three settings, various options are presented to the user. For the reading action the user can select between all the symbols in the program's alphabet or the "*Any*" option, which means that the action will take place regardless from the symbol read on the current tape cell. For the write action only symbols present in the dictionary are presented to the user. The move options are the two basic options ("Left" and "Right") and the "End" action, which means the next state is the ending state ("Halt" instruction). The user can select the starting state as the final one, thus creating a recursive action. Double tapping a state allows the relabeling of the state, presenting a keyboard to the user to define any name he prefers to the selected state. Tapping on an empty area on the screen deselects any previous selection, while a double tap selects all the elements in the state diagram, allowing the user to move the whole diagram. If a larger area is needed for the diagram designing, the user can pinch and on the screen to zoom in or out. If an instruction table view of the program is preferred or needed, the user can view it by entering the program settings view, where such a view is presented and directly modifiable.

When selecting a state or a connector, by tapping the action button, the corresponding element can be deleted. If a state is selected the user can also choose to insert a program in the selected state. Inserting a program means that once the execution of the program arrives to the selected state, the inserted program will be automatically executed and then the moving action will take place. More than one program can be added to a state, their execution order can be then managed and each program can be removed from the

selected state. This feature can be seen as a subroutine in procedural programming languages.

5. The main implementation

The simulator has some simple structures to interpret and coherently represent the described model and the UI.

Every program is a *dictionary* structure, containing 5 key-value pairs:

States - A mutable array containing all the states defined by the user during the program definition. Each state is a dictionary structure itself, containing three key-value pairs - “Name”, which defines the label for the state; “X” and “Y”, which contain the state position on the screen. Optionally a state can contain a fourth key-value structure, named ***Program***, which contains a complete program structure, representing the inserted program.

Connections - A mutable array containing all the connections defined by the user during the program definition. Each connection is a dictionary structure containing 5 key-value pairs - “From”, defining the starting state for the connection; “To”, defining the final state for the connection; “Read”, defining which character has to be read for the action to take place; “Write”, defining which character has to be written on the current tape cell; “Move”, which defines the direction to take after writing the character on the tape cell.

Alphabet - A mutable array of characters which simply defines the alphabet of the program.

LoopWatcher - A Boolean value defining if the *LoopWatcher* is active or not.

Name - A string value containing the name of the program.

Each time a program is loaded, these values are populated by those saved in the program structure. These structures are created by saving a program, doing so the software adds the program to the ***Programs*** array, which contains all the saved programs, ready to be loaded at any time. If a user downloads a program from the web repository, the downloaded program is added to the ***Programs*** array.

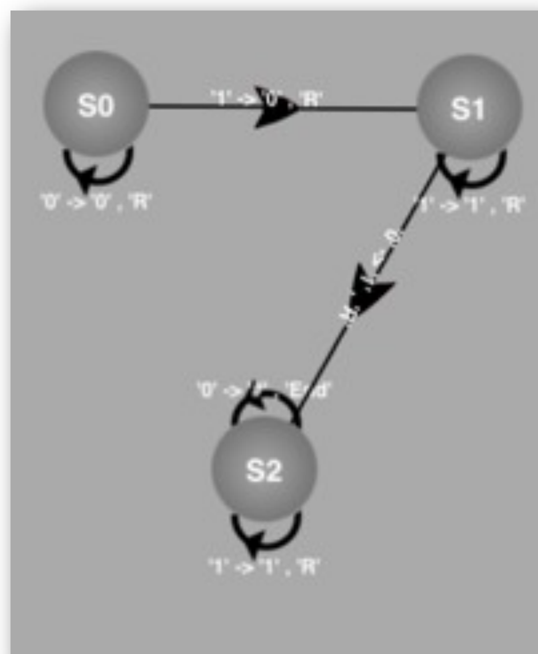
7. Setting up, running & debugging the designed program

Running a Turing program means nothing more then executing each instruction and follow the states diagram correctly. For modern machines this is an elementary task, but needs a robust and stable implementation to avoid infinite loops and incorrect execution.
Reading and writing of alphabet only characters is granted by the design phase, which allows the user to select only from characters part of the alphabet defined by the user himself, rather than letting him type the character to read or write on the tape cell. When the running view is loaded the program is automatically checked for having an '*End*' action ('*Halt*'), if it does not, the software does not allow the user to execute the designed program, informing him with a corresponding text message. If the program is correctly designed from an initial analysis point of view, the program can be run.
Executing the program itself is a quite trivial algorithm, based just on a timer, firing every one-second, which executes a "*doAction"* routine. In future versions the execution speed will be customizable by the user, by incrementing or slowing down the timer fire interval.
At each execution step the routine checks the current state the program is in, cycles through all connections attached to the current state, checks the current tape cell, simulating the '*Read*' action, and writes the corresponding character on the tape. Finally it sets the current state to the one described by the connector that was used. Obviously the tape is moved in the given direction ('*Left*' or '*Right*') or end execution if the next moving action is the ending one ('*End*').
If none of the attached actions contains a "Read" instruction that matches with the current tape cell, the routine informs the user by throwing an exception, signaling that no action is specified for the current character on the tape. If the *LoopWatcher* is enabled, the routine traces the current state. If the current state remains the same for 50 consecutive cycles, an exception is thrown, signaling that the program may be in an infinite loop.
The simulator allows the user to execute the program in two different modes. The first is a continuous execution, which simply runs the whole program without stopping (if no errors make the program halt) through the last instruction ('*End*'). The second mode is a step-by-step mode, which allows the user to execute one instruction at the time. The user is also able to mesh up the two running modes. This can be done in two ways. The first one is to pause execution while the program is running in continuous mode, by tapping the 'Pause' button. Once the program is paused, the user can choose if to resume execution in a continuous way, by tapping the '*Play*' button, or a step-by-step execution. The second way is to tap the '*Play*' button after an execution step was completed in step-by-step execution mode.
To pause the program execution the user is enabled to tap the '*Pause*' button, or to set a breakpoint inside one or more desired states of the program. This is done at design time, just by selecting the desired state and selecting '*Insert Breakpoint*' from the menu after tapping the '*Action Button*'. Removing the breakpoint is as easy as adding it, the App will recognize that a state has a breakpoint already inserted in it and the '*Insert Breakpoint*' option will be replaced by a '*Remove breakpoint*' one inside the menu shown by tapping

the '*Action Button*'. The simulator will pause the execution once it arrives at the desired state, ***before*** executing ***any*** instruction.
If the execution is paused, the current state is saved to the heap, to allow the user to continue the execution afterwards or to continue execution in a step-by-step mode. If the execution is stopped by the user, the current state is reset to the initial one, but the tape does not reset to the initial state. It's the user's responsibility to reset the tape to the desired initial state by accessing the program settings.
During execution the tape is updated in realtime, avoiding any animation for performance reasons. The current action being executed is printed on screen in real time too, and the tape status is represented also in string form, not only in graphical way.
Once the program ends correctly the tape shows it's status in both ways, graphical and in string form. The string form divides each tape cell by a '**|**' (*pipeline*) character.

8. A simple Turing Machine Program

As an example of a program designed and run with the proposed software let's have a look at the adding program. Such a program sums two numbers represented by a sequence of '1'. For example the number '3' will be represented b the sequence '111'. The two numbers will be separated by a '0' in our example, which represents a white space. At the end of the execution the tape will contain a sequence of '1' which will represent the sum of the two numbers initially present on the tape.
The proposed Turing program is shown below. The state diagram was created with the simulator being presented in this document.



Three states have been defined. The state "S0" cycles on itself while the tape contains a '0' character on it, not changing anything and moving to the right on the tape. When a '1' is

read the program writes a zero on the tape, moves right and the program goes to the state “S1”. In this state the program cycles on itself again moving right while reading a ‘1’ and doesn't change the characters on the tape. When a ‘0’ is read, the program writes a ‘1’ instead and moves to the right on the tape, going to state “S2”. In this state the program cycles on that state while reading a ‘1’ and not changing any character on the tape. Finally, when a ‘0’ is read again on the tape, the program writes a ‘#’ character and ends. The instruction table for that program is presented below, this is a screenshot from the instruction table automatically generated by the simulator.

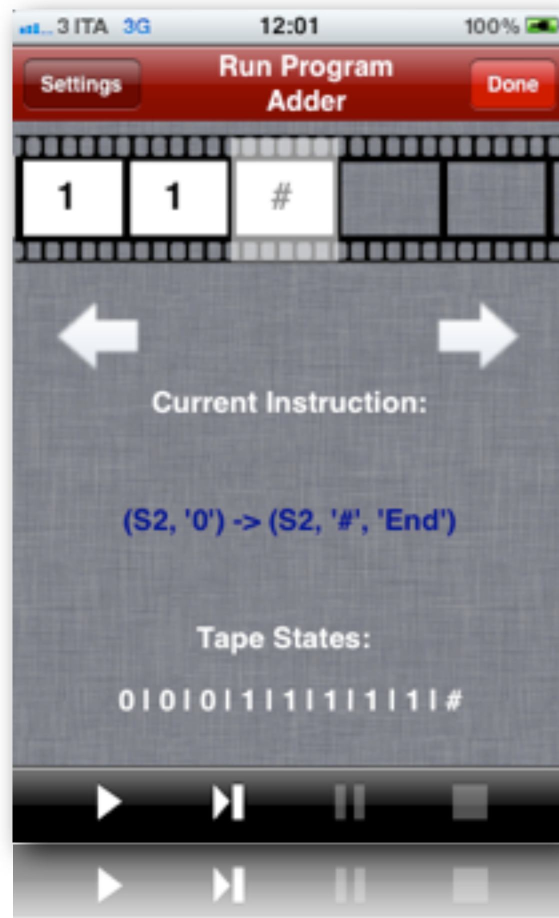
(S0 , '0') -> (S0, '0', 'R')
(S0 , '1') -> (S1, '0', 'R')
(S1 , '1') -> (S1, '1', 'R')
(S1 , '0') -> (S2, '1', 'R')
(S2 , '1') -> (S2, '1', 'R')
(S2 , '0') -> (S2, '#', 'End')

The initial tape state is shown below as presented by the simulator.

0
0
1
1
0
1
1
1
0

The designed program is now ready to be run and tested inside the simulator. Accessing the running view presents the current tape. The reading header was set at the beginning of the tape as shown in the screenshot below.

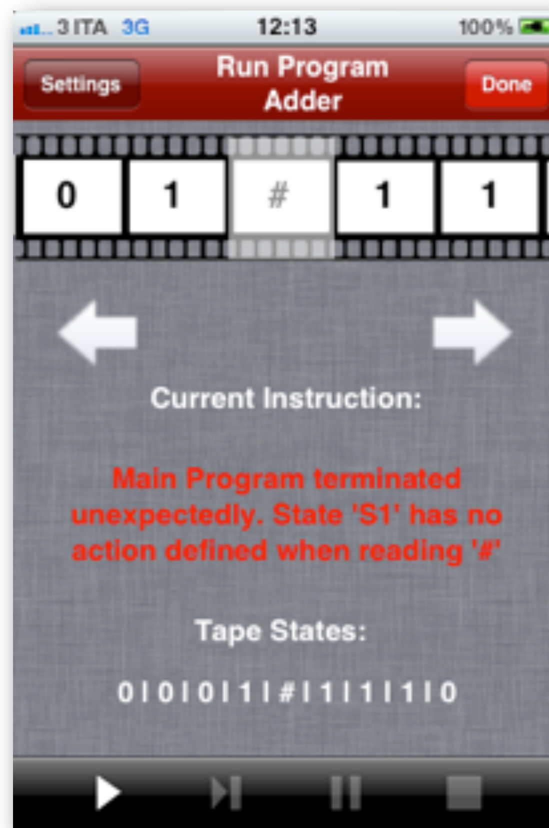
By tapping the ‘Run’ button the execution starts. After a few seconds the program ends correctly and the simulator presents the following screen:



The “*Current Instruction*” shows the last action taken by the program, which is to write a ‘#’ when a ‘0’ is read in state “S2”. Below the final tape state is presented in string form. Note that the addition has been done correctly, presenting five consecutive ‘1’ on the tape after the two initial ‘0’, which have been not modified, and another ‘0’, which was written in state “S0” instead of the first ‘1’. The ‘0’ separating the two sequences of ‘1’ was replaced in state “S1” by a ‘1’. The ending ‘#’ character has been correctly written on the tape instead of the last ‘0’.

This was just a simple example of designing and running a program with the presented simulator. As you can note there are situation that are not defined in the program design. For example what would happen if in state “S1” a ‘#’ character would be read on the tape? We modify the initial tape as follows:

Now we start the program again from the “Run” screen. When the program enters state “S1” and the ‘#’ character is read, it halts, presenting an error, as no action was defined for this situation at design time. The following screen is presented to the user. Note the red caption describing the halting reason and the reading header at the top of the screen still in the reading position of the ‘#’ character. At the bottom of the screen the tape is represented in string form. Note that the first ‘1’ character was correctly overwritten with a ‘0’ in state “S0”.



Thanks to this feature the user is encouraged to create a better design of the program and prevented from defining programs which are not aware of possible halting situations.
Obviously this was a very simple example of a program with very few states, but when writing a larger program with many different states and actions, this feature can turn out very useful as it's very common to forget some possible situations or not being able to fully predict the tape status in each state. Being informed of which state wasn't able to read which character allows the user to immediately correct the error or make the necessary corrections to the whole program definitions, which can become a tedious task when the state diagram contains 10 or more states and even more connectors between them.

9. What's next?

The Turing Machine computation model is very powerful. Having just three elementary actions to take at each execution step ("Left", "Right" or "End"), each state is almost deterministic. Thanks to its simple alphabet-based model the Turing Machine has a huge expression power.
Theoretically every computable problem can be solved by an adequate Turing Machine, for complex problems probably such a machine becomes very extended and complex. For real world purpose such a machine is even almost unimaginable. The software presented in this project allows the user to insert a whole Turing Machine program in a single state.

The aim for this choice is to give even more expressive and computable power to the Turing Machine model. A step further will be to allow the tape reader to interpret more complex objects on the tape than just characters, without abandoning the idea of simple "Read" and "Write" operations and continuing to allow just a "Left" and a "Right" move at each execution step. Another big gap in the Turing Machine model is the lack of memory. Emulating a heap or a stack using thew classic Turin Machine model is not trivial and in some cases even impossible. Adding a sort of '*Super Tape*', or '*Global Tape*' could be a work around for this problem and is one of the future objective of this project.

What we want to show here, is that a pseudo codification, based on the Turing Machine model, giving the final developer, the user, a small set of instruction, but a large set of objects to work with can bring to a very powerful developing strategy, which drastically reduces the generation of syntactic and logical errors through it's limited set of instructions and improves the expression power given by the Turing Machine model.