Analysis

Introduction

Turing machines (TM) are theoretical machines described in 1936 by Alan Turing. They are Finite State machines (FSM) with a theoretically infinite tape memory divided into discrete cells, represented by characters. A read/write head moves along the tape, giving input from the tape to the ‘control unit’, which decides what operation to perform on the tape.

The control unit uses states to analyse the input from the tape. Each state has a different definition of what to do when it reads a certain character from the tape. These definitions involve changing the contents of the ‘cell’ currently being read by the read/write head, what direction the read/write head should move after doing this, and the new state of the machine.

A one tape Turing machine can be defined as *M = {Q, , b, , , q­0, F}*, where:

* *Q* is a finite set of states;
* **is a finite set of alphabet characters;
* *b* is the symbol to be recognised as a blank space;
* ** is the set of input characters for the tape;
* **is the set of transition functions that define how the machine reacts to the symbol read in based on the state of the machine;
* *q0* is the initial state;
* *F* is a finite set of halting/accepting states.

*Tape*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |  |

*Read/Write Scanner Head*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ***Q*** | | | | | |
| *Table of * | *Qa* | | | *Qb* | | |
| ****** | Write | Move | State | Write | Move | State |
| *blank* | *blank* | *stay* | *Fe* | *blank* | *stay* | *Fo* |
| *1* | *1* | *right* | *Qb* | *1* | *right* | *Qa* |
| *0* | *0* | *right* | *Qa* | *0* | *right* | *Qb* |

*Control Unit*

*Write Out*

*Move Out*

*Read In*

Example Turing Machine

The following example is a Deterministic Turing machine can be defined by these parameters:

States (Q): [a, b]

Alphabet (): [0, 1]

Blank (b): \_

Input Characters (): 100111011

Transition Functions (): [a \_ → e \_ S,

a 0 → a 0 R,

a 1 → b 1 R,

b \_ → o \_ S,

b 0 → b 0 R,

b 1 → a 1 R]

Initial State (q0): a

Halting States (F): [e, o]

The Turing machine uses a set of transition functions to run, and they determine what happens when a certain character is read from each state. As a table, they can be represented as:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Current state a** | | | **Current state b** | | |
| **Character Read** | **Write** | **Move** | **New State** | **Write** | **Move** | **New State** |
| **\_** | \_ | None | e | \_ | None | o |
| **0** | 0 | Right | a | 0 | Right | b |
| **1** | 1 | Right | b | 1 | Right | a |

This set of transition functions is designed to calculate the parity bit (whether there are an even or odd number of 1s) of the set of input characters. Whenever a ‘1’ is read in any running state, the new state is the other state. This means that the initial state, a, is selected when an even (or zero) number of 1s has been read by the tape, and the other running state, b, is selected when an odd number of 1s has been read by the tape.

The read/write head in this example is set to the first character in the series of input characters. Therefore, the first time the machine reaches a blank space, it will know it has reached the end of the binary sequence it is calculating the parity bit for. Therefore, whatever state the machine is at this stage tells us what the parity bit is (a for even, b for odd). Therefore, the machine moves into the two halting states e (for even) and o (for odd) to show the user what the parity bit is.

There are many different types of Turing machine. The example given above is the most common kind, a Deterministic machine. Each variation of Turing machine uses variations of the parameters given above (see Objectives – Point 7 for more).

Problem

My program is intended to simulate six different kinds of Turing machine. The user should be able to create their own Turing machine by typing transition functions into a text box, edit the contents of their tape, and edit other parameters of the Turing machine. After this, they should be able to press a button and run their Turing machine for as long as it takes to halt.

The program should be able to represent a Turing machine as a Finite State machine. TMs and FSMs are very closely linked, as the control unit of a TM works very similarly to an FSM, only with two output values (write and move). This feature will enable my users to view their TMs in a graphical representation, which will be easy to understand for students and easy to explain for teachers.

I also intend to simplify the design of my Turing machine as well. Many parameters in the Turing machine are superfluous and can be automatically assigned (or not assigned at all). For example, the blank character, b, does not need to be editable by the user, as my program will keep it as a space, which is the most logical character to use for a ‘blank’ cell. In addition, the alphabet of the machine can be derived from the contents of the tape and the transition functions written by the user. Characters that are not used in either of these will not be present in the machine, and thus do not need to be accounted for. In fact, out of the seven parameters used to define a Turing machine, M, only two, the input characters (*∑*) and the initial state (*Q0*) need to be specified by the user in my program.

There should be a GUI (Graphical User Interface) in an easy-to-use layout. The purpose of this tool is to teach people about Turing machines, and therefore the GUI should be designed in a way that gives the user all the tools to do this without overwhelming them.

Audience

This program is aimed at teachers and students. Teachers can use this program to show their students examples of Turing machines in order to improve their understanding of this rather abstract concept. Students can also make good use of this program as a revision tool for Turing machines and Finite State Machines.

There will also be a number of extra-curricular subjects in my program that can be accessed by students. This could be given as extension work by teachers, and would help students to solidify their knowledge of the topics taught on the A-Level curriculum.

End User

My end user will be Mr Barnes, my Computer Science teacher. My program will be a teaching application that he can use when teaching students theory about Turing machines. This program can simulate Turing machines, so he can demonstrate the operation of them, and he can also use the built in FSM conversion algorithm to show the class about how TMs and FSMs are linked.

Interview with End User

This interview with Mr Barnes established some key features that he would like to see in a teaching application, and how they would be an improvement over his current method of teaching the subject. Some sections of the questions and answers in this interview have been reworded for clarity.

**How do you currently teach students about Turing Machines?**

*Currently, I use diagrams of Finite State Machines and tapes to explain how the machine outputs to the tape. I also hand out worksheets that have a small pointer printed onto a piece of paper with them that they can move along a tape to help them visualise the process of moving the read/write head. This is quite difficult when I ask students to create their own Turing machine diagrams, as I have to mark the diagrams state-by-state.*

**What issues are limiting factors for this method?**

*When talking about conceptual subjects like Turing machines, it can be quite difficult for students to picture the tape’s head shifting along it after transitions, and keeping track of what state the machine is in. Also, when the students answer questions on it, there’s no real way of finding out if they’re right except if they trace through the machine’s operation one step at a time which isn’t very efficient. This is true for me, too. If I have to go through a machine one state at a time, there’s a greater chance of me marking it incorrectly.*

**What features would be important to you in my program?**

*There would need to be a way to control the speed of the machine. It’s not very useful running a Turing machine at full speed if it finishes in only a few seconds. Controlling the speed would let me explain the operation of a machine step-by-step to the class to ensure they understand what is happening in the machine.*

*It would also be useful to include a number of example programs. These would be a useful tool to show to the class when they begin to understand the basics of Turing machines so I can introduce them to some more complex algorithms.*

*There should be a way to save Turing machines written in the program. This would be very useful when marking work students do in the program – they could write their machines and save them, then send them to me via e-mail. I could easily open them up and run them, where I can see if they’ve completed the task correctly.*

*I would also find it useful to include additional features that extend beyond the A-Level curriculum. A number of students in my class are interested in topics covered in class beyond what is taught, and they ask me where they could find out more. Including extra-curricular materials in the program would help me easily guide them to a resource with these materials in.*

**Do you think my program could benefit your students’ learning? If so, why?**

*I do think so. Having an animated representation of the Turing machine running would help them visualise how the machine processes each step. Also, having the ability to write your own Turing machines in the program would improve their logical thinking skills – an important part of their exams.*

*Also, the idea of including extra-curricular material in the program sounds useful. After students grasp the concepts taught in the curriculum, I can set them work in the program beyond the syllabus as an extension. This extension work would challenge them and would help to solidify their understanding of the subject.*

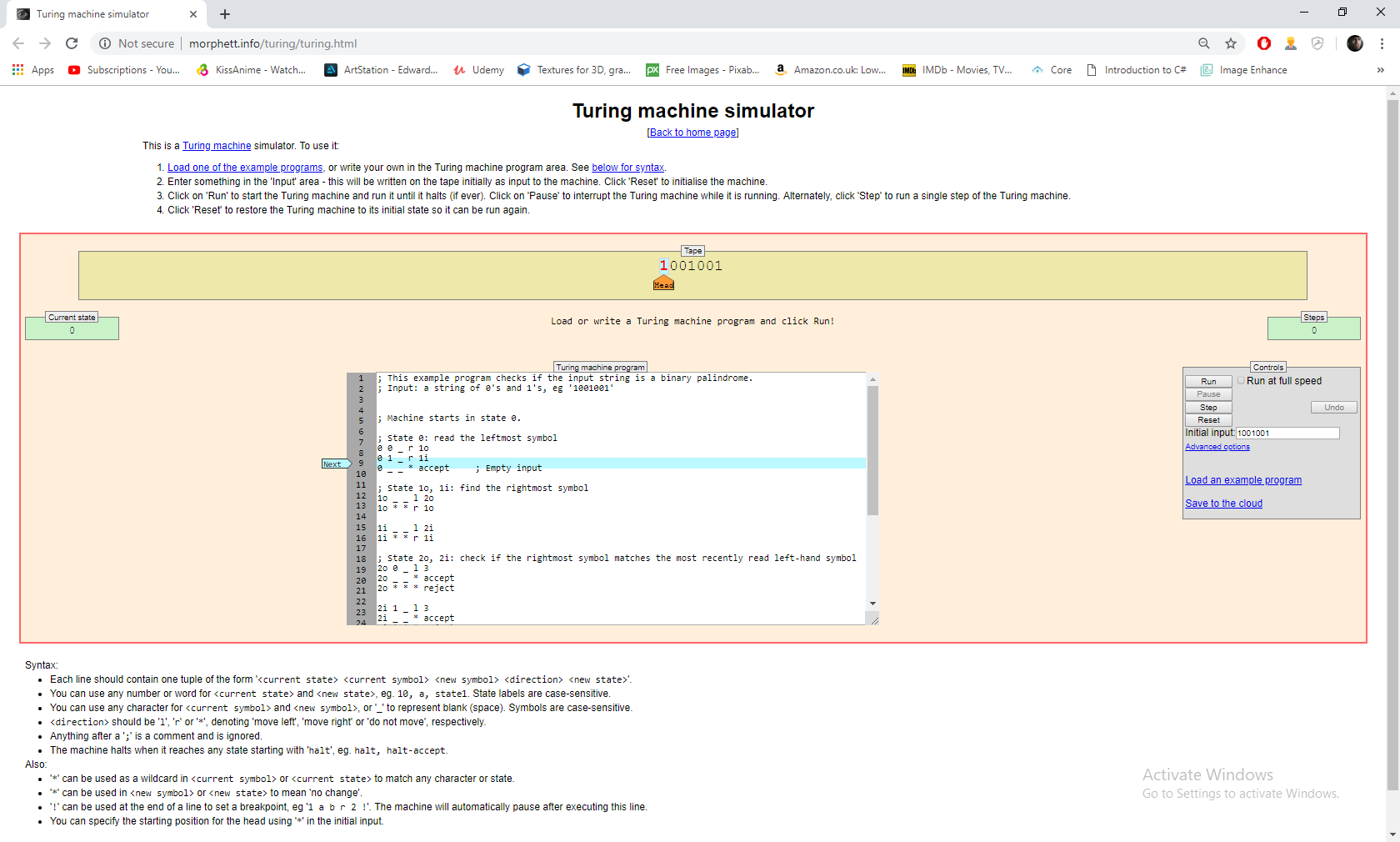
*Having the ability to display a Turing machine as an FSM would be very beneficial. This feature would help link their work to the kind of diagrams they would need to draw in their exams, which would then improve their ability to read FSMs and understand what function they would perform.*

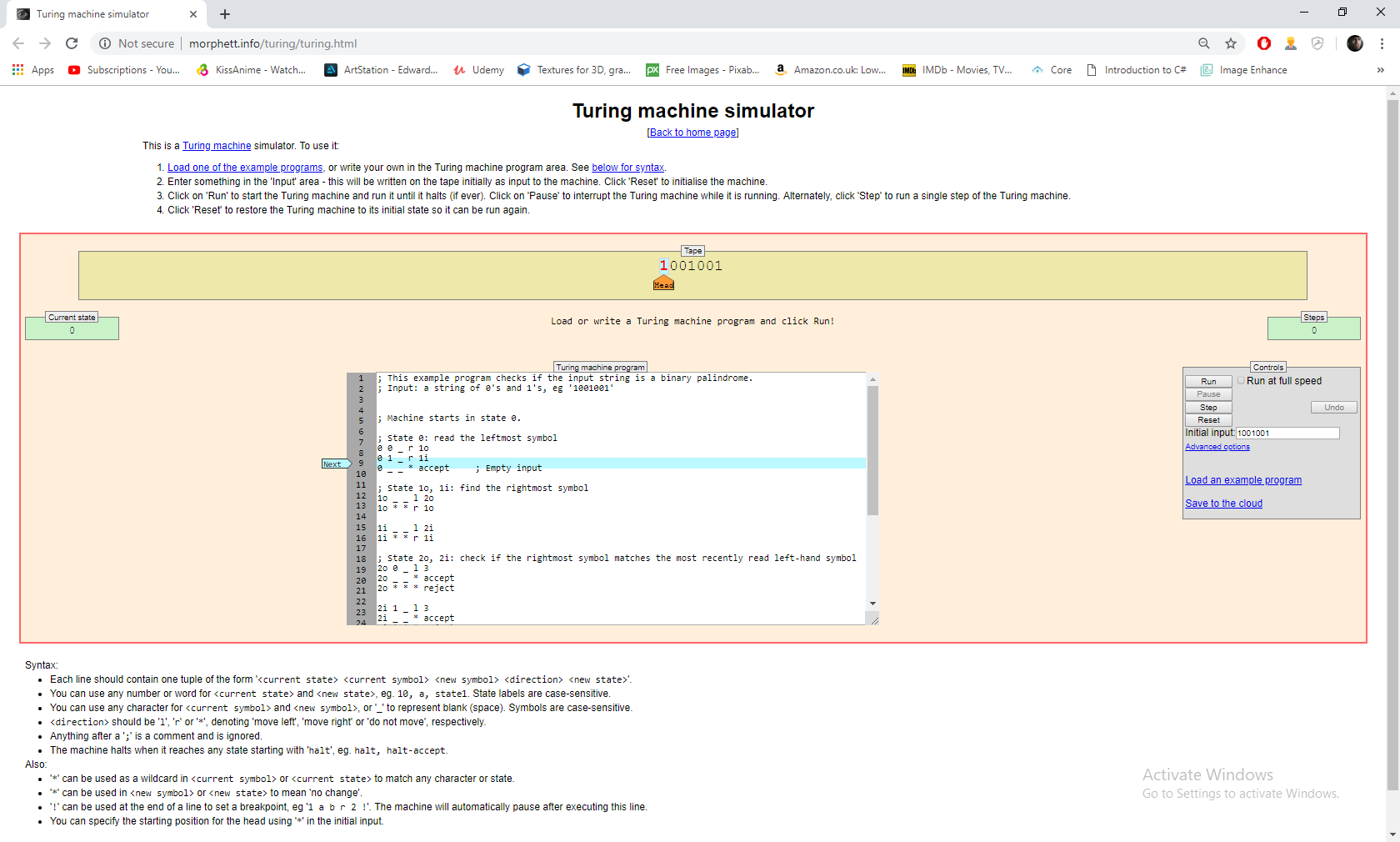
**What acceptable limitations are there for this program?**

*I usually wouldn’t use very long or complex machines – a machine that checks the parity of a sequence would be the typical complexity of a machine. Students don’t need to write long FSMs or Turing machines in their exams, so for the curriculum the complexity of machines do not need to be great. When talking about the extra-curricular aspects of the program, it would be very useful if the program could handle large quantities of data – that’d help people who want to challenge themselves to make complex algorithms.*

Existing Software

There currently exists a number of Turing machine simulators that are open source and free to use. These are typically online simulators that use a web-based HTML GUI to allow the user to simulate these machines. An example of a commonly used simulator is <http://morphett.info/turing/turing.html>:





The simulator exhibits the design that I want to emulate. I will be, however, changing certain aspects of the program such as the spacing between option panes and adding fields for further adjustments to the program. I will also be changing the default colour scheme to monochrome white.

One drawback of this simulator is that it has a limited number of machine variants. The simulator allows for users to use a ‘standard’ Deterministic machine, a Semi-Infinite machine, and a Non-Deterministic machine. This limits the number of programs that can be written easily in this simulator. It is correct that any algorithm can run in this simulator, but it would take more time without the convenience of more machine variants. For example, performing multiple addition operations at once would be much easier using a multi-track machine rather than a single-tape Deterministic machine. Another example would be the ability to emulate other Turing machines. Using a multi-tape machine for the template for a Universal Turing machine would be much more useful for the user, which is what my program will do. There will also be an algorithm to automatically generate the very lengthy code required to run a Universal Turing machine successfully in my program which is not available in other simulators.

Additionally, the feature to produce a graphical representation of the Turing machine as a Finite State Machine (FSM) does not exist in the current system. With my system, there will be an integrated feature allowing a user to quickly produce an FSM from the Turing machine they have written. This improves and speeds up the workflow of people using my system, and also provides an easy way to visualise their Turing machine.

Research Sources

The following websites were used to gather information on how certain Turing machines work and how others have gone about abstracting them:

Wikipedia – Turing Machines: <https://en.wikipedia.org/wiki/Turing_machine>

Wikipedia – Universal Turing Machine: <https://en.wikipedia.org/wiki/Universal_Turing_machine>

Cambridge University Computer Laboratory – What is a Turing Machine?: <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html>

Coding Concepts

For the creation of the program, I intend to use C# with the Windows Form extensions to allow me to create a GUI for ease of use for the end user. I will use Microsoft’s documentation of their coding languages as well as tutorials on sites such as YouTube to help me create the program.

I will use Object Oriented Programming (OOP) to help me code the program, as well as advanced Regular Expressions. I will also be creating methods to save and read data to and from text files in the computer’s storage.

Data Volumes

As a teaching tool, I would not expect the program to be utilised for large programs. Usually teachers would use somewhere between 5 and 10 states for an example machine, with a tape size of around 10.

However, I intend to make the program able to perform programs with large quantities of states. This is to facilitate the creation of high-complexity algorithms to aid students researching extra-curricular materials using my program. In addition to having a large state capacity, I also want the tapes to be able to store a large quantity of information. This is to allow a large number of instructions to be stored when using a Universal Turing machine and transition functions are placed on the Instruction Tape. Having a large tape capacity will allow for complex non-Universal algorithms to be used as well.

One section of my program I do not intend to have a high data volume would be the Finite State Machine. This section is mainly intended to be used for teaching and showing students the relationship between TMs and FSMs. Ideally, this section would be able to produce complex FSMs but for its purpose it is not strictly necessary. This section can handle any number of states, just like the Turing machine simulator itself, but it is unlikely a large number of states will be automatically represented in a clear and easily understandable way.

Acceptable Limitations

* Turing machines have a theoretically infinite tape size. This will not be possible as a computer has a limited memory. Therefore, there will be a limited tape size in my simulation.
* In addition to this, there will be a limited number of states that can be used in a Turing machine, as these will also need to be stored in memory which is not infinite.
* Because a Universal Turing machine also contains instructions on its tapes, and the states need to be an alphabetic character, therefore there can only be 52 (26 upper and 26 lower case characters) different states on the instruction tape of a UTM.
* The limitations of creating a program as opposed to a web service is that there is no easy access to cloud saving. The current system has a way to save Turing machine files to the cloud, enabling the user to quickly retrieve them from any device. While my system will be unable to do that, it can allow users to save Turing machines to text files on their system, which they can transfer between machines.

Planning

1. Produce a console application for a Turing machine.
   1. Begin by producing an application that can run a single-line machine.
   2. Expand the function of this application by letting it run more than one line.
   3. Create a system to import a text file and parse the parameters for the Turing machine from it.
2. Produce a Human-Computer Interface that allows the user to interact with the Turing machine graphically.
   1. This step will likely not all be done at once. Additional interface elements will be added over the course of development as new sections of the program are added.
3. Implement the code from the console ‘skeleton’ program into the HCI.
4. Expand the utility of the Turing machine. Allow for the use of default case functions and speed controls.
5. Produce algorithms for the other five Turing machines.
6. Create an algorithm that can convert a Deterministic Turing machine into a Universal Turing machine.
7. Make a module that can take a Turing machine and represent it as a Finite State Machine in a new window.
8. Add additional functionality to the program that make it more intuitive, useful and robust.
9. Perform rigorous bug-testing on all sections of the program after completing each section. Ensure that all functions are working as intended and not producing any logic as well as runtime errors.

System Requirements

* The user will need to be able to enter information into the program and will need to be able to edit the settings of the program. Therefore, they will need a mouse and keyboard attached to their machine.
* The user will not need an internet connection to run the program but must be able to download it from a GitHub repository to get it onto their machine.
* As the program will be coded in Visual Studio, the user will need the appropriate Microsoft Visual C# Redistributable Packages to allow them to run the program. Because the program is written in a Microsoft programming language, the user will require a Windows operating system to use this program.
* The hardware requirements for the program will not be very high. However, if the user tries to use the machine at a very high frequency of steps per second the hardware requirements will rise along with this frequency.

General Objectives

1. Have an intuitive and interactive Human-Computer Interface.
2. Produce a built in ‘Info’ section that provides the user information on how the program works and what it can do.
3. Include example Turing machines for the user to look at in order to understand how to produce their own better.
4. Have a text box that can use a number of formatting tools, including commenting out lines of code.
5. Let the user save Turing machines to text files and open them again.
6. Have preferences the user can modify to their liking, which will stay persistent even after the program closes.
7. Have the ability to run Turing machines at a high speed, making complex calculations feasible.
8. Have six different Turing machines, each with their own properties, which allows the user to produce a wide variety of algorithms easier than when using only a single type of Turing machine.
9. Give the user the option to represent their Turing machine as a Finite State machine.
10. Perform validation on the Turing machine when necessary, to ensure no errors are encountered when saving the machine, running the machine, or displaying it as a Finite State machine.
11. Ensure that no issues can be caused while the machine is running by removing the user’s ability to edit the parameters of the Turing machine while it is running.

Specific Objectives

1. The GUI must be interactive.
   1. There will be a text editor to allow the user to enter in their Turing machine code.
      1. In order to simplify the process of making transition functions, I will employ a text-based method to produce functions, similar to the example application above.
      2. The user can type functions into the text editor without the need for interacting with any interface to produce their instruction set. This process is similar to producing program code in an IDE (Integrated Development Environment), which is useful as it is a workflow familiar to people who study and teach computer science.
      3. Each transition function has a set number of ‘fields’, properties that define it. Generally, these are:
         1. Initial state, the state that the machine must be in to accept this function;
         2. Read key, the character that the read/write head must be reading to accept this character;
         3. Write key, the character that is written to the tape at the read/write head if the function is accepted;
         4. Move key, the direction the read/write head moves after the function is accepted;
         5. Final state, the state the machine ends in after accepting the function.
      4. The only machine to differ from this rule is the Read-Only machine, which does not contain a write key field.
   2. There will be a pane to the right of this text box which will contain a number of options to edit the simulation. These will include:
      1. ‘Run’ will begin the simulation;
      2. ‘Pause’ will stop the simulation where it is until it is restarted;
      3. ‘Step’ will perform one command at a time. Can be used while paused;
      4. ‘Reset’ will reset the tape and stop the simulation;
      5. ‘New Machine’ will open a new tab with all blank values with the machine type set to default;
      6. ‘Close Machine’ will delete the currently selected machine;
      7. ‘Save Machine’ will save the machine to a text file or a Turing machine file;
      8. ‘Open Machine’ will import a machine from a text file or a Turing machine file;
      9. ‘Speed’ will be a slider that controls the speed at which the simulation will run;
         1. The speed value will be defined in milliseconds.
         2. The default values will be 1ms and 1000ms, however this can be altered by the user.
         3. There will be a label showing the exact value of the slider. This is to help the user fine-tune the speed they want the machine to run at.
         4. At lower values, there may be inconsistencies with the value selected and the actual speed of one step. This is caused by the computer not being fast enough to process one step of the machine in the amount of time specified by the user.
      10. ‘Run at Full Speed’ will be a checkbox above the speed slider. When this is checked, the machine will operate faster than it would do compared to normal operation;
      11. ‘Machine’ will be a dropdown box which allows the user to choose between a variety of Turing machines to simulate;
      12. ‘New Tape’ is an option that will appear when the machine variant selected allows for more than one tape to be used;
      13. ‘Generate Universal Code’ is an option that will be available when using a Universal Turing Machine (UTM). The purpose of this button is to automatically generate the complex code required to use a UTM correctly;
      14. There will be a label containing what line of the code is currently being selected. The purpose of this is to help the user quickly navigate to a specific line when they encounter a syntax error – the line number of an error is given which helps to locate it and fix it quickly.
   3. The tape will rest above the text box and options pane.
      1. The item containing the contents of the tape will be a text box. When the machine is not in operation, this can be written to directly, instead of needing to write to a text box in the options pane.
      2. The first tape (and all the tapes in the multi-track machine) will have a dropdown box for the user to specify the initial state of that tape.
      3. The tape should be able to remove additional spaces created during the operation of the machine.
         1. If the read/write head moves beyond the tape, spaces may be added onto the tape. After returning to the bounds of the tape, this would result in the tape not being centred. The program should have a procedure that removes blank spaces from the tape that affect the centring of the tape.
      4. When applicable, tapes have a button called ‘Close Tape’ which removes them from the program:
         1. Single tape machines do not have this option, unless if additional tapes are present when switching to this machine. In this case, the tapes must be closed using this button before running.
         2. Multi-Tape and Multi-Track machines give the user the option to close any tape above the first tape.
         3. Universal Turing machines do not give the option to close any tapes, as this machine must always have exactly three tapes.
      5. The read/write head for the tape will rest below it, and below the head will be buttons which can move the head’s initial position from left to right along the tape.
      6. Sets of labels will be present for the machine to display information about how many instructions have been performed and what state(s) the machine is in.
         1. For Multi-Tape machines, there will be a set of labels next to each tape, as each tape has its own set of values for these variables.
         2. For other machines, there will be a set of labels just above the options pane which will show the information for the whole machine – as these machines use only one set of values.
      7. For machines that use only a single state, a set of labels will be present just above the options pane to display the number of steps taken and the current state of the machine.
   4. To the right of the options pane will be a read-only debugging text box. This will print information about the machine once it has halted. Due to differences in machines, the format of this will differ slightly based on which machine variant is selected:
      1. Single tape Turing machines state the final contents of the tape, the final state of the machine and how many steps were taken by the machine to reach the halting state. Universal Turing machines are also printed this way. The final tape contents is taken from the Data Tape, the final state is taken from the State Tape, and the steps taken is found from the standard step label.
      2. Multi-Tape Turing machines give a numbered list of the tape. Below this list is the final state of the machine and how many steps were taken by the machine to reach the halting state.
      3. Multi-Track Turing machines give a numbered list of the final parameters of each tape. Each element in the list contains the final contents of the tape, the final state of the tape, and the number of steps taken by the tape before it halts.
      4. To give the user the option to reduce the number of elements on the screen, the debug box can be toggled using a check box called ‘Enable Debug Box’. Doing this will cause the box to not be written to after halting – reducing the visual complexity of the machine if the user wants to focus on running the machine.
      5. In addition to point iv, there will be an option to ‘Clear Debug Box’. This button will remove all of the information in the text box, allowing the user to reset it if they have performed many tests and it is very long.
   5. A menu will be at the very top of the machine.
      1. This machine will have a number of tabs to access certain sections of the program:
         1. ‘File’ handles aspects relating to opening, saving, creating and deleting Turing machines. In addition to this, there is the option to quit the program from this menu:
            1. New Machine (Ctrl-N)
            2. Delete Machine (Ctrl-W)
            3. Save Machine (Ctrl-S)
            4. Open Machine (Ctrl-O)
            5. Quit (Ctrl-Q)
         2. The ‘Edit’ tab contains options that are used to edit either the machine or program’s contents:
            1. Settings
            2. Convert to Universal
         3. The ‘Machine’ tab relates to running, pausing and resetting the machine:
            1. Run (F5)
            2. Pause
            3. Reset
            4. Step (F11)
         4. ‘View’ contains options that let the user view and run example Turing machines, view their Turing machines as Finite State Machines, and find information about how to create Turing machines:
            1. Info
            2. Display as FSM
            3. Examples:

Addition

Subtraction

Palindrome Check

Parity Calculator

* + 1. Some of these options contain keyboard shortcuts, as shown above. These allow the user to quickly perform simple tasks after they get used to the program.

1. Have a built-in guide for the program that helps the user understand how to use the program. The guide should contain a number of sections:
   1. General Information. This section should give the user a general description of what a Turing machine is, how they work, and what they are used for in computing.
   2. Transition Functions. This should tell the user the syntax used for each machine in the program, and what each field means.
   3. Universal Machines. This section includes information on the contents of the tapes of a UTM and how to properly write them.
   4. Default Cases. This should explain to the user how each machine uses the default case character in its transition functions.
2. The text box will take the Turing machine code from the user.
   1. Allow the user to have multiple Turing machines on the program at once, using tabs to switch the machine to be read.
      1. Changing tabs will cause the information on the tape, tape head, initial state and machine type fields to change to reflect the information saved in the newly selected machine.
      2. The user can right click on the selected tab to open a context menu which will contain the following:
         1. Rename Machine;
            1. Doing this will open a window that will get the new machine name from the user. Provided the name entered is not blank, the name of the tab page containing the machine will be updated to this new name when the user presses ‘OK’.
         2. Duplicate Machine;
            1. This will create an exact copy of all of the elements of the machine, including the tape contents and head position, etc.
         3. Close Machine.
   2. Can have complex formatting available.
      1. The text box should let the user use the Undo (Ctrl-Z) shortcut to undo mistakes.
      2. The text box should let the user copy and paste text blocks into and out of the machine.
      3. The text box should prevent anything other than text from being pasted into it.
      4. There should be an option to comment out lines of code by placing a ‘//’ at the beginning of the line. This lets the user produce lines of description for their code, and space it out along the text box without losing formatting after exporting.
   3. When the Turing machine is edited, the drop-down box used to select the initial state of the machine should be updated to reflect the new set of initial states that can be used.
      1. If the initial state that is currently selected by the machine is deleted from the text box, the drop-down box’s selected value should become blank.
      2. If a state is only referenced at the end of transition functions, it will be automatically identified as a halting state, and will not be included in this list.
   4. Have an ‘override’ feature for transition functions.
      1. This will be, by default, the ‘\*’ character.
      2. When placed in the write key field of a transition function, the machine will not write anything to the tape and leave the current key as it is.
         1. In a Multi-Tape (and Universal) machine, where each transition function has *k* write keys (where *k* is the number of tapes), only the tapes where the default case character is written will be not written to.
            1. E.g. for the write key field *‘\*11’,* only the first tape will not be written to, and the second and third tapes will have ‘1’s written to them.
         2. For every other machine variant, as their write keys are only one character long, whenever the instruction is parsed the tape will not be written to.
      3. When placed in the read key field of a transition function, the machine should use this as the ‘default case function’ for a state. If the currently selected key on the tape does not match any read key in the transition functions for the current state, then the machine will use this default case function as the instruction to perform.
         1. For single tape Turing machines, the override key will simply be executed if there is no matching transition function for the current state and read key.
         2. For Multi-Track machines, each tape’s operation is decided independently. Therefore, a tape will decide to use the default case function only if its own current state and read key do not have a function defined for it.
         3. Multi-Tape machines have 3 types of transition functions:
            1. Fully defined functions – highest priority. This function contains no instances of the default case character in its read key. E.g. *“101”.*
            2. Partially defined function – middle priority. This function contains a mixture of ‘real’ characters and the default case character. E.g. *“1\*\*”.*

A partially defined function’s priority is related to how many occurrences of the default case function there is. The more occurrences, the lower its priority as it is less fully-defined. For example, the read key field “*1\*1”* has a higher priority than the field *“1\*\*”*, as the first has a fewer number of default case characters in it.

* + - * 1. Default case function – lowest priority. This function contains the default case function in its read key for all tapes. This function will only be executed if there is no other transition function for that state that matches what is being read from the tape.
        2. These rules are also followed by the Universal Turing machine, as it uses the same algorithm as the Multi-Tape machine to function.

1. The program can import and export Turing machines.
   1. The program will read and write programs as follows:
      1. Machine variant ID;
      2. Number of tapes (if the machine can have more than one);
      3. Initial tape, tape head and state (if more than one tape, these are repeated, but a multi-tape machine will only have one initial state at the end where a multi-track will have an initial state for each tape);
      4. The set of transition functions which the machine will run.
      5. The name of the machine in the program will be the name of the text file the machine is imported from.
      6. If the file chosen to be opened is in an incorrect format, an error message will be displayed to the user, and the machine will not be opened.
   2. The user can specify the name of the file, as well as navigating to the save directory using the dialog box that opens when opening or saving a machine.
      1. When saving a Turing machine, the format of the machine must pass the same requirements as when attempting to run the machine.
         1. This is to prevent issues with opening the machine again, as if a saved machine has errors there may be issues with exporting it with the correct format, causing it to be un-openable.
   3. There are a number of example Turing machines that can be opened (see 1e) that can allow the user to learn the syntaxes of the program easily.
      1. These are stored in the program’s Debug folder and can be opened from the top menu. The method of opening these files is the same as opening any other Turing machine file.
2. Allow the user to customise certain settings in the program.
   1. An item in the top menu allows the user to open a “Settings” window.
   2. This will let the user adjust some of the settings of the program. They will be able to change:
      1. The minimum and maximum speed of the Turing machine.
         1. The minimum speed of the Turing machine must always be less than the maximum speed of the Turing machine. If this condition is not met, the program will not allow the user to confirm this choice.
      2. The character used for default case transition functions.
         1. The default case character cannot take the value of the blank character, which is represented by the machine interchangeably as an underscore ‘\_’ and a space ‘ ’.
         2. If the above condition is not met, or if this field is left blank, the program will not allow the user to confirm this choice.
      3. The number of steps performed by the machine before updating the GUI when running the machine at ‘full speed’.
   3. After modifying the parameters of the program, the updated preferences will be stored in a configuration file, which will be loaded in every time the program runs, and the user’s preferences will be set when the program opens.
      1. If the configuration file cannot be found in the program’s directory, a new configuration file will be created, saving the default values to it. The default values for the different settings are:
         1. Maximum speed = 1000
         2. Minimum speed = 1
         3. Default Case Character = ‘\*’
         4. Full Speed Interval = 50
3. Be able to run complex Turing machines quickly.
   1. The ‘Run at Full Speed’ option will set the machine to run based off a different timer. This timer is at a fixed value of 10 milliseconds. Each time the timer ticks, it will perform many steps of the machine at once, after which it will update the UI elements.
   2. The number of steps performed by the machine before updating is defined by the ‘Full Speed Interval’ value in the settings menu.
   3. This is designed to speed up the machine, as updating the UI elements after each step takes significant amounts of time. Reducing how often the UI is updated will reduce the overall computing time of the Turing machine.
   4. Enabling this option should disable the ability to modulate the speed of the program using the slider in the options pane – this option does not affect ‘Full Speed’ mode. When ‘Full Speed’ mode is turned off, the slider should be enabled once again.
4. The program should allow the user to select what kind of Turing machine to use. The program will load up a Deterministic Turing machine by default. However, the user will be able to choose between a few other Turing machines. The number of machines available will be determined by whether or not it is possible to create said machine in C#. For example, it would not be realistic to create a Quantum Turing machine. I will, however, make the following machines:
   1. Deterministic Turing machine (DTM);
      1. This will be the default setting. This machine does not allow for ‘duplicate’ instructions, has a theoretically infinite tape size and can move in any direction along the tape. It can also change the values on the tape and the program is independent of the tape.
      2. The formal definition of the transition functions in this Turing machine in my program is:

*δ : Q x  → (Q/F) x Γ x {L,R,S}*

* 1. Read-Only Turing machine;
     1. A Read Only Turing machine can only move along the tape and cannot edit its contents.
     2. This machine is defined using the same parameters as the Deterministic machine. However, the one difference is that its transition functions, δ, is defined without a read key:

*δ : Q x Γ → (Q/F) x {L,R,S}*

* 1. Non-Deterministic Turing machine;
     1. This machine can use instructions where the read key is the same as another instruction in the same state. When this occurs, the simulation will choose a random instruction to follow out of *n* instructions with the same read key (when the read key matches that of the key on the tape).
     2. The formal definition for the transition functions in this Turing machine is:

*δ : Q x Γ → ((Q/F) x Γ x {L,R,S})n*

* 1. Multi-tape Turing machine;
     1. This machine will have *k* number of tapes. Each tape has its own head for reading and writing. The tapes act together, with the machine having one overall state and each line of machine code having *k* write keys, *k* read keys, and *k* move keys, and each set informing the action of each tape.
     2. This machine is defined slightly differently from the other Turing machines. Because the machine has multiple tapes, it is written as:

*M = {Q, , b, k, , q­0, F}*

* + 1. This is because there are *k* tapes, thus our tape parameter, ∑, must be a list of length *k*. The transition functions for this machine are also defined differently:

*δ : Q x (Γ)k → (Q/F) x (Γ x {L,R,S})k*

* + 1. Each tape has its own input and output fields, and this is reflected as such in the transition functions, with the input and output being lists of length *k*.
  1. Multi-track Turing machine;
     1. This machine also has *k* number of tapes. However, the tapes work independently of each other. This means that each tape has its own state and runs off code with one write key, one read key and one move key.
     2. The Multi-Track Turing machine is defined differently from other Turing machines. Similar to Multi-Tape machines, it has *k* tapes. It also has *k* initial states. Each tape acts independently, and therefore can start from whatever state independently. Therefore, the machine is defined as:

*M = {Q, , b, k, , q­0k, F}*

* + 1. The transition functions for this machine are defined identically to Deterministic Turing machines. This is because each tape acts independently, and gives and receives to and from the control module independently.
  1. Universal Turing machine (UTM).
     1. A Universal Turing machine is a Turing machine that takes other Turing machines as its input, and evaluates them. For simplicity, the only Turing machine that can be simulated on this Universal Turing machine are Deterministic Turing machines.
     2. This is a Multi-Tape machine consisting always of three tapes. The tapes will store different information about a DTM, and will work to emulate it. From top to bottom, the tapes of the UTM will contain:
        1. The tape of the DTM
        2. The current state of the DTM
        3. The transition function set of the DTM
     3. There will be a built-in algorithm to convert a DTM into the tape inputs for a UTM.
        1. Only Deterministic machines can be changed into UTMs.
        2. The algorithm will set the machine type to “Universal”, set the number of tapes in the machine to three, and produce the following:
           1. Instruction Tape;
           2. State Tape;
           3. Data Tape.
        3. The instruction tape contains the set of transition functions of the DTM.
           1. A single character is used to represent each state, and the ‘\_’ characters are converted into ‘-’ characters to avoid confusion for the UTM’s transition functions.
           2. If the DTM to be converted uses default case functions, the machine will not be allowed to be converted. This is because a tape cannot contain the default case characters as its presence would cause confusion when dealing with default case functions.
        4. The State Tape will contain the single character representation of the initial state of the machine from the DTM.
        5. The Data Tape contains a copy of the tape from the DTM.
     4. There will be also an algorithm to produce a set of transition functions to simulate the DTM.
        1. This set of transition functions will be able to be used for any DTM that shares the same alphabet, running states and halting states as the DTM it was generated for, regardless of whether or not the instructions themselves are the same.
     5. This machine is defined using the following:

*M = {Q, , b, I, , , q­o, F}*

* + 1. Here, *I* is the contents of the Instruction Tape, *∑* the contents of the State Tape, and *∆* the contents of the Data Tape. Because the tapes of the Turing machine work together to achieve a result, they can be defined using the transition function definition of a Multi-Tape machine where *k* = 3:

*δ : Q x (Γ)3 → (Q/F) x (Γ x {L,R,S})3*

1. The user should be able to produce an FSM representation of their Turing machine.
   1. The following machine variants can be displayed as Turing machines:
      1. Deterministic Turing machines
      2. Read-Only Turing machines
      3. Multi-Tape Turing machines
      4. Multi-Track Turing machines.
   2. The states in the FSM should be represented by labelled rings. The halting states should contain an additional ring in the picture to identify them.
   3. For a transition function, an arrow, or ‘edge’, should point from the initial state of the function to the final state of the function. There should be different ways of drawing these functions based on the final state’s position relative to the initial state’s position:
      1. A standard, straight line should be used if the final state is to the right of the initial state;
      2. A loop should be used if the final state is the same as the initial state;
      3. An arc should be used if the final state is to the left of the initial state. This arc should travel high, as it should not intersect with any looping lines that may be between it and the final state.
   4. The edge of a transition should be labelled with the information of how the function is accepted (what its read key is) and what occurs when it is (what its write and move keys are).
      1. Because some states may have more than one function travelling to the same final state, an algorithm to combine these functions into one label should be made. Once these have been combined, they should be simplified down using the following rules:
         1. If all combined functions share the same write key(s), then the combined label only needs one instance of this.
         2. If all combined functions share the same move key(s), then the combined label only needs one instance of this.
         3. If none of the functions to be combined result in the tape being written to, then the write key portion of the label can be removed entirely.
   5. The machine should only be able to be displayed as an FSM if the machine is valid and can be run by the program. That is to say that it must meet all of the conditions set forth in Point 9.
2. The program will ensure the code entered by the user is suitable and will not cause any errors. The machine will be checked when the user presses ‘Run’, ‘Step’, ‘Save Machine’, or ‘Display as FSM’. This will ensure there will be no crashes relating to the parsing of information from the user.
   1. Each transition function in the text editor is evaluated using a series of regular expressions and selection statements. The following criteria must be met by every function in the machine:
      1. The write, move, and read keys are not characters, but strings;
         1. This is true for all machines except the Multi-Tape machine. This machine allows for multiple read, write and move keys on each function. However, the length of all of these fields must be the same as the number of tapes in the machine.
      2. Each transition function must have the correct number of fields:
         1. Most machines require all five fields to be valid functions.
         2. The one exception to this rule is Read-Only machines, as they do not contain a write key field.
      3. The number of write, move and read keys are not appropriate for the machine to be run;
         1. The write and move keys can be any character except for the ‘space’ character. In transition functions, the blank character is represented by an underscore ‘\_’.
         2. The move key can only be three characters, ‘r’ for right, ‘l’ for left and ‘\_’ for stay.
   2. If the tape contains the default case character, to avoid confusion when dealing with default case functions, the machine will not be allowed to run.
   3. If the initial state setting on any of the tapes are blank, the machine will not be allowed to run.
   4. If the machine is a UTM, and the tapes do not match the required syntax for this machine variant, the program will not allow the machine to run.
      1. Each transition function placed on the Instruction Tape of the UTM must be separated by a ‘|’ character.
      2. The State Tape must contain only one character.
   5. If there is an error in the machine, a specific error message is given and the line number in which the error occurred is given. This helps the user quickly fix any issues with their code.
3. The program will ensure that when running, the machine’s data cannot be edited to ensure no errors can occur by the user changing data during operation.
   1. When the machine is running, all functionality is disabled except for the following elements:
      1. Pause;
      2. Speed Slider and its labels (if not using Full Speed mode).
   2. When the machine is compiled, but paused, these elements should be usable:
      1. Run;
      2. Reset;
      3. Step;
      4. Run at Full Speed;
      5. Speed Slider and its labels (if not using Full Speed mode).
   3. After resetting the machine, all of the controls should be enabled again. This excludes the Reset button, as this is not a usable feature if the machine is not compiled. Also, the Speed Slider may not be enabled, depending on what state the ‘Run at Full Speed’ checkbox is in.