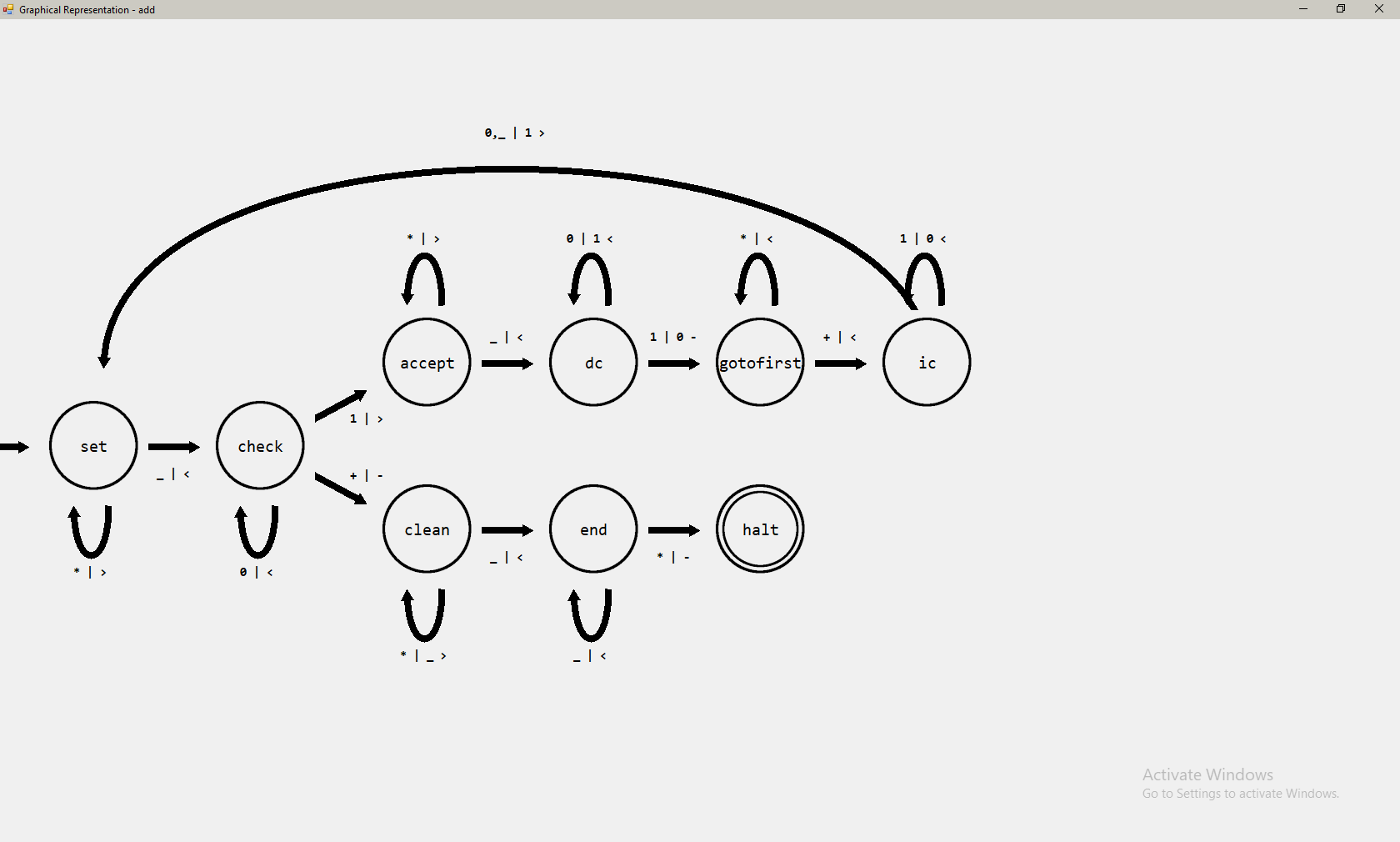
# Evaluation

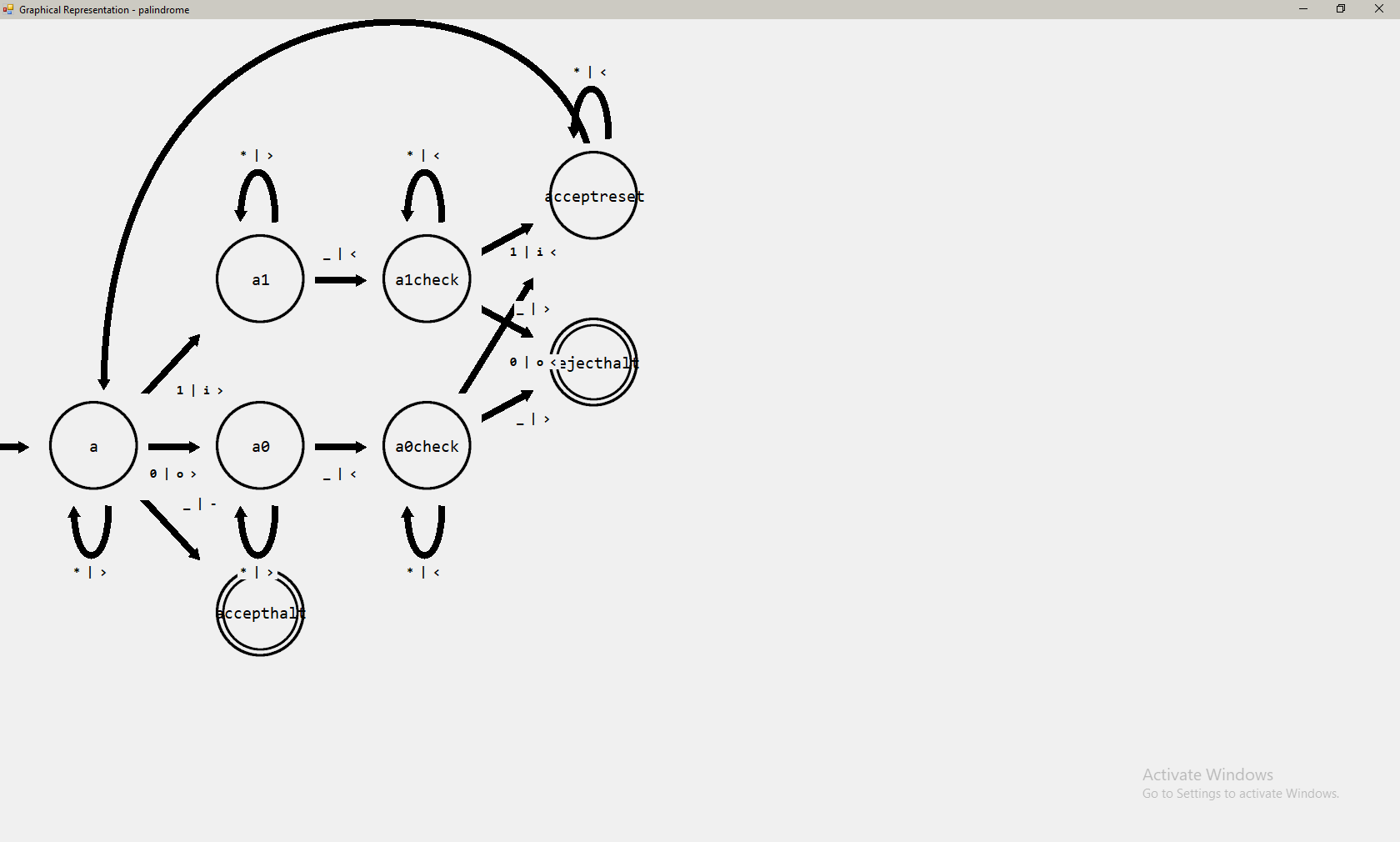
## General Review

Overall, my project has very effectively met the requirements of my objectives. The main aspect of the program, the simulation of the Turing machines, work exactly as laid out by the objectives. In addition, the simulator is able to be run at a very high speed using Full Speed mode, performing tens of thousands of steps in only a few dozen seconds. This shows that the machine is well optimised and can be used to perform very complex calculations like long Universal Turing machines.

The Finite State Machine feature of the program was slightly disappointing in terms of performance. When dealing with simple Turing machines, such as my Addition example machine, the algorithm used represents it as an FSM very cleanly:



However, when considering more complex machines, the appearance of them were sometimes less aesthetically appealing and sometimes more difficult to follow:



This was likely due to my algorithm being quite basic in the placement of the states – they did not account for the proximity of a state relative to other states and this can sometimes cause edges to overlap and make the labels difficult to read. This issue is particularly noticeable in the highlighted area above.

This is not a huge issue in terms of the usability of the program, as this feature was intended mainly for the teaching part of this program. A teacher can easily produce an FSM from a Turing machine which, when teaching at A Level, will not be a very complex machine. The likelihood of a Turing machine being used that is even as complex as the Addition example machine in an A Level lesson is very low. Therefore, while this section of the program does not cater to high complexity tasks like other sections, it still performs its role satisfactorily.

The Universal Turing machine performs far beyond my initial expectations of it. The use of three tapes aided the algorithm immensely. Initially, I was using a single tape, with the instructions, state, read/write head position, and tape all on the one tape. This caused the algorithm to simulate the machine run incredibly slowly, going back and forth for each comparison. Changing to a three-tape machine allowed me to perform comparisons that could have taken a hundred steps in the old system to just one step in the new.

Teaching the user how to create Turing machines was an important part of my project. There is a plethora of example machines available for the user to look at and test, and there is also an information page built-in to the program. These features help to improve the teaching capability for an end user that is researching Turing machines in their spare time – they have a tool that is able to simulate any algorithm in a number of different machines that also explains to them how those machines function not just practically but also theoretically. This section of the program is not very complex in terms of implementation – the most complex part reuses code from importing Turing machines. However, the implementation of it provides a lot of assistance to the end user trying to make sense of the program.

## Does my Program Meet my Objectives?

|  |  |
| --- | --- |
| **Objective** | **Evaluation** |
| 1a | A text box is used in my final program as the element to edit the transition functions of the machine. This reduces the amount of time the user needs to take to produce machines – once they learn the syntax, they can very quickly produce lines of code that would take longer if using a series of drop-down boxes for a function, for example.  The instructions are parsed in the order specified in this objective as well, so this objective has been met. |
| 1b | All of the elements mentioned in this objective were implemented into the program. This can be seen in test 1, where the UI is loaded. Labels used in this pane are updated whenever the relevant object’s values are updated. For example, when adjusting the speed of the machine, the label with the value of the speed updates as the slider is being moved, which is shown in test 81 and 82 when the slider is moved. |
| 1c | The tape exists above the text editor and options pane, as specified in the objective. The text box that contains the contents of the tape is editable when the machine is not running, and when it is, it is read-only. This helps with the simplicity of the program, as writing to a separate text box (pg. 8) to specify the tape’s contents.  Below the tape’s text box is a text box containing the read/write head. This has buttons below it that allow it to be set to a specific location before running the program as shown in tests 14 and 15.  Each tape in a multi-track machine has a set of labels specifying the current state and step count of the tape. As intended, these labels are not shown in other machine variants, with these machines using a single set of labels above the options pane, which are hidden when using a multi-track machine. This is shown in tests 25 and 26.  Similarly, the drop-down boxes for the initial states of each tape in a multi-track machine will all be visible. In other machines, only the option for the bottom tape will be visible, as all tapes in these machines share the same state. This is shown in test 19 and 20.  The tapes can be closed by the user with a button attached to the object. This option is displayed correctly for each machine, as written in this objective and proven in tests 19 and 20. |
| 1d | The debug text box has a couple of options below it. The option to disable inputs to it is used, as is the option to clear its contents. They both work as intended as shown in tests 128 and 129.  The debug box uses a number of different formats to print the information from the tape(s) after they halt.  Single tape and Universal machines take the final tape, contents, and step count and print them.  Multi-tape machines take the final tapes and produce a list of them, listing the final state of the machine and step count afterwards.  Multi-track machines take the final tapes, final states and step counts and produces a list of them for each tape.  These formats are verified by tests 63-65. |
| 1e | The top menu contains a number of miscellaneous options as well as duplicates of on-screen options. The duplicated options are designed to act more as an ‘index’ of the shortcut keys that can be used in the program. Additional features are placed here to reduce the visual complexity of the base program, but still give the user easy access to these options. |
| 2 | The guide produced contains all of the desired topics of discussion. The “Transition Functions” section needed to be split into two different tabs to be slightly less confusing for a beginner but each section conveys the information that it needs to. |
| 3a | The text editor section of the program uses a TabControl object to contain multiple tabs. Each tab contains a text box that can have the code for that machine written in. Changing tabs will change what machine is selected, mapping the new values to all of the UI elements in the program. This gives the user the ability to have multiple machines open at once as seen in test 7.  When right clicking on the selected tab, a context menu is displayed with additional options. The user can delete the machine from here, rename the machine, or duplicate the machine  Deleting the machine from here does nothing different from the “Delete Machine” control in the options pane.  Duplicating the machine, as seen in test 6, produces a carbon-copy of the machine, ensuring that a new, cloned object reference is created for the MachineData tag to prevent the two machines from being linked.  Renaming the machine, which is tested in test 8, opens a window that allows the user to enter the new name for their machine. Provided what is entered is not blank, the machine will be renamed – which is shown as the label of its tab.  One part of this objective I would have expanded on if I had more time would be the context menu. I would have devised a method to select the tab that was right-clicked over. Doing this would give the user greater and faster control over managing the machines in their program. For example, if “machine2” was selected but I right click over “machine1”, this machine is selected and its context menu is shown. However, because this is a rather minor feature, this was cut for the sake of time. |
| 3b | Using a RichTextBox control as opposed to a TextBox for my editor, most features for formatting were included automatically. One section I needed to edit was the pasting function, which needed to remove any images from what would be pasted in to the text box (test 29).  One part of this section that does not fully follow the objective is the shortcut for the Redo button. The RichTextBox control maps Ctrl-R to “align right”, which aligns the text to the right side of the control. It uses Ctrl-Y for Redo. I decided to not create my own Redo shortcut because most other programs use Ctrl-Y as their Redo shortcut. Because I wanted to make the design as intuitive as possible, I used shortcuts that were used in other programs (e.g., F5 for run program, Ctrl-N to open a new machine) and decided to keep Redo as Ctrl-Y for this reason.  In addition, whenever the text box’s information is parsed, it checks for lines beginning with ‘//’ and ignores them (tests 52 and 53). This allows the user to comment out lines of code that they do not yet want to implement, as well as add spaces between sections of code and add descriptions. |
| 3c | When editing the text box, the states that can be used by the machine are being constantly updated, as seen in test 11. Therefore, every time the text box is edited, the contents of the text box is taken and the initial states of each line is added to a list, which can be used by the initial state drop-down boxes to select the initial state(s) for the machine. |
| 3d | A rather logically complex section of the program, this objective required the program to produce lists of what read keys can override the default case function in a machine. This section was relatively simple for the single tape and multi-track machines – the priorities of functions were binary – fully defined and high priority, or default case and low priority.  For multi-tape machines, the system was much more complex as there were now different priority levels – each read key field could contain any number of default case characters. My implementation of the objective involved producing a list of all sequences of characters from the transition functions of the current state, except for the default case function. These sequences’ priority depended on how fully-defined they were. Therefore, when checking if a function is of a high enough priority, its read keys were compared to this list of sequences to determine if it is the highest priority matching function.  This objective is proven to be correctly implemented over a number of tests, from test IDs 66 to 79. |
| 4a | The program correctly opens text files in the format specified in this objective (specific syntax and examples given at pg. 36) as seen in test 4. If this format is not correct, exception handling prevents the program from crashing and tells the user that this file cannot be opened.  If I had more time to produce my program, I would have changed the opening procedure of the program to preserve the formatting of the text files to be opened. Currently, any blank spaces in the transition function set are ignored and not read in. This is not so important due to the commenting feature, which allows users to simply produce a line with ‘//’ in and nothing else – effectively producing a blank space. This feature was of a rather low priority which is why it was not included in the final program. |
| 4b | The user is able to choose the name of their saved file and where it is stored. The format of the file is consistent with the files to be opened, so the user can open the Turing machine whenever needed. This is proven by test 5.  The validation function is run before saving machines, to ensure that the saved machine’s format is valid to remove the risk of the file not being openable after saving. |
| 4c | The example machines in the program allow the user to see how algorithms are written in the program easily. We can see this feature is working by test 119. This feature works well with objective 2, as they are both good tools for teaching the user how to produce their own Turing machines.  If I had more time, I would have produced a few more complex examples for more advanced users to use and learn how to produce more difficult algorithms. |
| 5a | In the top menu, under the “Edit” tab, is the settings feature. This opens a window that lets the user enter personalised settings for certain parameters of the program. We can see the Settings menu opening correctly in test 89. |
| 5b | The settings menu allows the user to change:   * Maximum and minimum speeds of the machine * Default case character * Interval at which the machine runs in full speed mode   When the user confirms their choices for the customised parameters, the program validates them and ensures they can be used in the program. The rules followed are the same as specified in the objective. If these rules are broken, the user will not be allowed to use these values. These validation rules are shown to be working in tests 82 to 86. |
| 5c | If the customised settings are accepted by the program, they are saved to the “config.ini” file in the Debug folder. This saving process is shown by test 88.  This file is opened every time the program is run, its contents is read in and the parameters are set. This lets the user keep their customised settings persistent even after closing the program, as proven in test 89.  If this file is not found, the default settings are used and a new file is created containing the default settings. This removes the risk of an error occurring. This is shown by test 90. |
| 6a | When the Full Speed option is turned on, each time the machine ticks, a large number of steps are completed. We can see this working in tests 123 and 124. |
| 6b | The quantity at which the machine increments is consistent with the value specified by the user in the Settings menu. We can see this is true by tests 123 and 124. |
| 6c | The Full Speed option in the program allows the user to run the program at a much higher speed than normal. This is reflected in the time-to-compute when this option is turned on. This is shown by test 125. |
| 6d | Turning on the Full Speed option disables the ability to change the speed of the machine (test 122), as this value is not used when Full Speed is enabled. The label showing the speed is also disabled to indicate this. When Full Speed mode is turned off, these controls are enabled once more. (tests 126 and 127) |
| 7a | The Deterministic machine is the default setting of the program. It consists of a single tape, with a single state. The transition functions of this machine are in the standard form defined in the Analysis section (pg. 4).  The operation of this machine is covered by tests 33 to 39. |
| 7b | The Read-Only machine is identical to the Deterministic machine in function except for the fact that the tape is not written to. Because of this, its transition functions do not contain a definition for a write key, as it is unnecessary in this machine.  The movements of this tape is implemented identically to the DTM, and this is tested in tests 36 to 39. |
| 7c | The Non-Deterministic machine also functions similarly to the Deterministic machine. The only difference in its function is that it allows for multiple functions with the same state and read key. The machine chooses between which function to execute randomly with equal probability for each function.  This fair randomness was shown in test 40.  The formal definition for a Non-Deterministic Turing Machine (NTM) was not followed during the implementation of this program. The definition given implied that each transition function would allow a list of write and move keys to be used, and one set would be chosen randomly from them.  This was changed in the final program to use a transition function for each set of write and move keys:    In the format proposed in the Objectives section, this set of functions would be expressed as:    While this format is not too difficult to read on a small scale, I realised that if the user wanted this function to branch more than a few times (e.g. 5 or more times), the function would quickly become difficult to read.  I also realised that, if the user were to have states that were more than a single character long (which is allowed in my program), there would be issues with splitting the final states up. I would need to introduce a separator character in-between these states, which would mean the user needs to learn even more formatting to be able to use this program efficiently. Therefore, I decided to change the format of this machine to the first example. |
| 7d | A Multi-Tape machine contains any number of tapes. The machine only has one state, and the tapes are controlled by a single control unit. Because of this, each transition function needs a list of read keys, write keys and move keys. The list should account for what is being read on each tape at once, what to write to each tape and how each read/write head should move after writing to the tape.  This machine uses a similar format as the NTM’s defined format (pg. 18), which I changed for the implementation of my code:    I decided to keep the format of the Multi-Tape machine the same as what was defined because this was a very efficient way to implement this transition function. There were no issues with having to separate multiple possible final states, and therefore the only programming needed was to separate the input and output fields into their characters.  The additional complexity added to this machine was relatively small – provided a reasonable number of tapes (e.g. 1 to 5) were used, the functions are still very easy to read.  The method of writing to the tapes in this machine is the same as in the DTM, just repeated multiple times. Having identical implementation means that tests 33 to 35 are valid for this machine. The same is true for movement, which is shown in tests 36 to 38 as well as transitioning to the final state in test 39. |
| 7e | Multi-Track machines contain any number of tapes. Each tape has its own state, and is controlled by its own control unit. This means that its functions are defined identically to a Deterministic machine, as each tape works independently of the others. One way to describe a Multi-Track machine is multiple Deterministic machines put together. This definition matches my implementation – the Multi-Track algorithm uses the same algorithm as the Deterministic machine but repeated for as many tapes in the machine. |
| 7f | This objective was followed for the most part. However, a slight deviation was made. In point 7fi, I specified that only Deterministic machines should be converted into UTMs. However, because the conversion process was also very simple for Read-Only machines, I made these machines also available to be turned into UTMs.  This is shown by test 117.  The Universal Machine functions the same as the Multi-Tape machine. However, its tape count is locked at three, to ensure all of the required information is on the tapes.  The algorithm to produce the UTM code functions correctly. While going through and proving this line-by-line wouldn’t be feasible, it is clear that the algorithm works as intended as it is capable of simulating any machine with the same parameters as the machine it was produced with.  The operation of UTMs are shown to be correct in tests 113 to 118. |
| 8a | The program successfully permits only the machines specified in this objective to be produced as FSMs  (tests 104 and 105). |
| 8b | The states in the FSM are produced using a depth first algorithm. This kind of algorithm reduces the different types of lines that need to be drawn, as explained in the Documented Design and shown in test 91.  A state in the FSM that does not have any transition functions coming from it is marked as a halting state with an additional ring around its label. |
| 8c | The different types of lines drawn onto the FSM are correctly decided according to the specifications of the objective.  The only issue with the drawing of these lines is the drifting of arrows from the final states if using a lot of branches, which can be seen in test 95. For example, if four different states branch from one state, the arrows pointing towards each of these states stop too soon. This issue would have been fixed given more time, however, as the size of Turing machine to be represented as an FSM is intended to be relatively small, this was not the highest priority to fix. |
| 8d | The labels to go on each transition are correctly computed and placed in the correct place. Each label is also simplified where possible to make them easier to read.  The production of labels is seen in tests 93 to 101.  The simplification of the contents of labels is proven by tests 106 to 110. |
| 8e | Before displaying a Turing machine as an FSM, it is checked using the validation algorithm in point 9. This removes the risk of the program crashing due to an error in the Turing machine in this feature. |
| 9a | The validation algorithm used by the program follows the rules specified in this objective closely. If any line breaks these rules, the machine is not allowed to run.  These rules are proven in tests 43 to 48. |
| 9b | If a tape contains the default case character and the user attempts to run the machine, the user’s attempt is refused and an error message is given to specify this issue.  This rule is shown to be working by test 49. |
| 9c | Because the machine needs an initial state to function properly, if any of the required instances of an initial state is not filled, the machine cannot run. The user must fix this issue before being able to run the machine.  This rule is shown by test 41. |
| 9d | The tapes on a UTM must follow strict rules for a UTM to be allowed to run. This is because the automatically generated set of transition functions are programmed to work only if the enforced structure is followed. Therefore, the machine cannot run if the instruction and state tapes are not correctly formatted.  These validation operations are shown in tests 50 and 51. |
| 9e | When an error in the transition functions of the machine is encountered by the algorithm, a set of comparisons are performed with regular expressions to determine the cause of the error. These expressions are broken down sections of the main regular expression to help determine the specific cause of the error. When the error is found, it is displayed to the user along with the line number the error was found in.  The line number given is shown to be correctly working in test 42. |
| 10a | When the machine is run, all controls are disabled except for the Pause and Speed options. The tape and read/write text boxes are not specifically disabled, as they need to remain active to keep their colour and prominence in the program. However, the tape is set to read-only to ensure that it is not edited when the machine is running.  This is shown by test 60 and 61. |
| 10b | If the machine is paused, the parameters of the machine must not be edited, so most controls remain unusable. However, the options to restart the machine (Run and Step) are enabled, as well as the option to Reset the machine. Because the machine is already paused, the Pause button is disabled as it does not serve a purpose at this stage. The ability to run the machine at full speed once restarted is made available at this stage as well, as the runTimer counter is disabled so no issued would be caused if the user wanted to switch to Full Speed mode.  This can be seen in test 122. |
| 10c | After resetting the machine, the parameters of the machine can be edited once again. Therefore, all of the states of the controls are returned to normal; most controls are re-enabled. However, the Pause and Reset buttons are disabled, as they do not serve a purpose when the machine is not compiled.  The controls can be seen to be enabled again in test 59. |

## Feedback from End User

After giving my program to Mr Barnes, he sent me an email in which he highlighted the main positives and negatives they found while testing my program (Note: Formatting has been edited for the following list to improve clarity):

Overall, I think that your program accomplishes your main goals well. The program is very good at showing a user how to make Turing machines. The main positives and negatives I found while using your program were:

Positives:

* **There are a lot of tools for teaching –** Having the ability to produce an FSM automatically with the click of a button is very useful. While it isn’t as refined as would be desired when dealing with high-level machines, it’s very good when I need to show my students how Turing machines and FSMs are linked. Also, the option to control the speed of a machine is useful because I can take the students through the operation of the machine at my own speed.
* **The user interface is well designed** – This lets both teachers and students understand how to use the program easily. The toolbar at the top gives me a list of most of the features in the program. The others can be found easily in the main window. I also like that, when switching machines or activating certain options, certain buttons and controls are disabled/enabled or hidden/shown. It improved my understanding of the program because anything unnecessary to my current machine was taken away so I was left with only the controls I needed at that moment.
* **There are a wide variety of unique machines** – The variants of machines that can be used in the program are easy to choose, and they offer useful insight into different aspects of computation. For example, the Universal machine is a good theoretical model for a CPU – data and instructions are stored on tapes, where they are both read in as data. This machine type is briefly mentioned in the syllabus, as well. Having this in the program would be useful for teaching this to my class – even though they do not need to know about algorithms used to run one.
* **A number of features for extra-curricular learning –** The extra machines included in the program are one feature that allow users to make more complex algorithms. For example, the multi-tape Turing machine lets users make very complex algorithms with multiple tapes. Also, Universal Turing machines, as mentioned above, are mentioned briefly in A-Level but can be investigated further using your program. Including these in the program can teach users about algorithms as well as our ideas about information – specifically how we deal with certain types, instructions and data.
* **Open/Save feature is useful –** When preparing a lesson, I can use the save feature in the program to save the Turing machines I want to use in lessons. This helps because I can open the file easily during the lesson without having to rewrite it, which would take up time. Also, I can set my students work in your program so they can practice their problem-solving skills, which they can save and send to me. I can open the files and mark them more easily than when they would be drawn as FSM diagrams.

Negatives:

* **The FSM feature doesn’t handle large machines very well** – When using machines with a lot of states and many branches, the FSM produced can become quite messy. For teaching, this doesn’t affect me that much as I would use simple TMs and FSMs to teach my class, but for high-end users this may be a drawback.
* **The text is too small, as is the main program window** – The UI’s design is good. However, it is quite small. When using this program in lesson, I need to make sure everyone in the class can see the contents of the board clearly. This means the text on screen should be large. It would be good if a feature to increase the font size of the text was added in the future. Also, the window does not maximise horizontally, meaning the size of the window is limited. I think that adding a feature to let the user make the window full-screen would also help me when using this program with a class.

## Analysis of Feedback

The feedback provided by Mr Barnes was very useful. It highlighted the strengths and weaknesses of my program, and what would have been useful to include in retrospect.

The main aim of this project was to produce a program that could:

* Teach users about how to define Turing machines;
* Give A-Level teachers a tool to use to improve their lessons;
* Allow A-Level students to research extra-curricular materials outside of lessons to further improve their knowledge of the subject.

Judging by the feedback given from my end user, I would say that these aims have been met well.

The program includes documentation about how to use my program and explains how it correlates to the theory of Turing machines. This is useful for teaching the person using my program about Turing machines. My end user thinks that the FSM feature of the program also helps in this regard, as it conveys the nature of the link between the two types of machine. As stated by Mr Barnes in his email, the user interface presents the subject material clearly; in a way that is easy for a student to understand.

Furthermore, the fact that the program can save and open Turing machines from text files aids the teaching process as well. Mr Barnes said that he could use this feature to have his students send their work in the program to him, where he could check it much more easily by simply simulating the machine instead of having to mark paper-based diagrams.

The program also achieves its aim of providing a teaching tool for A-Level teachers reasonably well. Mr Barnes’ criticism of the program not being able to go full screen, and the issue of text being too small are important features that should have been added in retrospect. These are definitely ways my program could be improved in the future. Functionally, my end user thought that the program does a good job at providing the tools for him to slowly talk their students through the operation of the machine. The ‘Step’ button and speed controls that allow my end user to talk the class through each step of operation of the machine at his own speed is a good example of this.

My program includes a number of features not covered on the A-Level syllabus. This includes the addition of a variety of different Turing machines from the typical Deterministic machine taught to A-Level students. For example, the Universal Turing machine is mentioned in the A-Level syllabus but is not described thoroughly. Including this in my program allows students to look at more complex algorithms than would typically be taught – which can help them to understand Turing machines in greater detail. The lack of a more advanced FSM generation method is a detractor from this aim. However, the FSM feature was mainly intended for fulfilling the purpose of teaching a class of students about A-Level Turing machines – though I would have altered its design to accommodate high-level users as well if I had the time to do so.

## Additional Feedback

I spoke to Henry Faull, a software development trainee, about the user interface of my program. The workflow of the program is quite similar to an IDE (Integrated Development Environment), as there is code which is written, compiled and run. He had the following things to say:

* **The included information and examples are useful –** Having a window with instructions on how to code the transition functions is helpful to new users of the program. Being introduced to a new coding format can be daunting but giving a guide on the syntax of the functions improves the learning curve. Putting in examples that I can read over to understand how to approach making my own algorithms helps as well.
* **The validation system could be improved –** At the moment, only the first error in the machine is recognised as an error before blocking the user from running the machine. It would be good if there was a list somewhere of all of the errors that are in the program, so that I wouldn’t need to try to run the program after fixing each error to see if there are any more.

## Analysis of Additional Feedback

Henry gave me useful feedback on the user interface.

The point raised about examples and documentation are in line with Mr Barnes’ opinion that they aid the learning process for new users, which is important as this fits with my aim to make this program an educational tool.

The idea of including a list of all of the errors in a machine would certainly be an improvement on the current system. To be able to see everything that is wrong with the code at once would greatly improve the user experience as they would not be forced to rerun the same machine repeatedly and only solve errors one at a time. For a large machine, this could be time consuming.

## How my Program could be improved

There are some ways that my program could be improved in the future.

One area of improvement would be the way of containing the compiled transition functions. Currently, multiple lists of transition functions are used (see Machine.cs) for objects that need it. One way of simplifying this would be to create a single list of a new class, TransitionFunction. All of the other classes for transition functions could inherit from this class, making this section of the program more compact and easier to read. Another way this section of the program could be improved, to reduce the time taken to compute the simulation of the Turing machine would be to implement a dictionary for the storage of transition functions. Each value of the dictionary would contain a list of transition functions for one state. Then, when evaluating functions to find a matching function, the program can get all of the transition functions correlating to the current state by referencing the current state as the key of the dictionary. E.g., Functions[state] would return a list of all of the transition functions that have the initial state “state” from the dictionary Functions.

There are a number of ways I could improve my Finite State Machine implementation. Firstly, I could devise a new way to place the states in the window such that they will be less likely to overlap. One way of doing this would be to make objects in the scene ‘repel’ each other, where each state ensures they are a certain distance away from any other state. Doing this would ensure no states overlap.

Another way of improving the FSM diagram would be to alter the way labels are placed on edges. The labels for edges could be rotated to be parallel to the edges. This would reduce the amount of space they take up, meaning they would be less likely to overlap with other edges, as they would be closer to their own. As a result of this, there would be less chance of mistaking what label corresponds to what edge.

Furthermore, I could add the ability to drag and drop states. The auto-generation method could run as normal, and afterwards the user could be given the ability to click and drag over the states to move them in the window. This manual control would be more reliable than the automatic method and would also allow the user to change the layout of the FSM to their personal preference, rather than what was imposed by the algorithm.

One way I could improve the teaching capability of the program would be to implement a full screen mode. Many projectors in schools may be of quite a low quality. Because of this, my program (which does not maximise) may be difficult to see from the backs of classrooms. Making the program full screen, with additional options such as font size for the tape(s) and transition functions would allow teaching staff to use my program more effectively to teach their students about Turing machines.

Another section that could be improved is my validation function. At present, the first error and the line in which it occurs is reported to the user. I could make use of the debug box at the right of the window to print off a list of all of the errors present in the current machine. Doing this would allow the user to go through the list and remove all errors with their program before running the machine again, improving their workflow.