

Automata and LogicEngineering 1/2

ALE1/ALE2 Report

|  |
| --- |
| Automata and Logic Engineering 2  Katerina Nicolaidou Louttchenko |

|  |
| --- |
|  |

|  |
| --- |
| Version 3.0  ALE2 Report |
|  |

|  |
| --- |
| Eindhoven, December, 2021 |

Contents

1. [Introduction 5](#_Toc93259806)
2. [Parse & Check DFA 6](#_Toc93259807)

[Process 6](#_Toc93259808)

[parse 6](#_Toc93259809)

[Check DFA 10](#_Toc93259810)

[Working Results 11](#_Toc93259811)

[Parse 11](#_Toc93259812)

[Check DFA 12](#_Toc93259813)

[Conclusions 13](#_Toc93259814)

1. [Accept String – Word Check 14](#_Toc93259815)

[Process 14](#_Toc93259816)

[Working Results 15](#_Toc93259817)

[Conclusions 17](#_Toc93259818)

1. [Regular Expression 18](#_Toc93259819)

[Process 18](#_Toc93259820)

[Working Results 25](#_Toc93259821)

[Conclusions 26](#_Toc93259822)

1. [Finite 27](#_Toc93259823)

[Process 27](#_Toc93259824)

[Working Results 29](#_Toc93259825)

[Conclusions 30](#_Toc93259826)

1. [NDFA 31](#_Toc93259827)

[Process 31](#_Toc93259828)

[Working Results 31](#_Toc93259829)

[Conclusions 31](#_Toc93259830)

1. [PDA 32](#_Toc93259831)

[Process 32](#_Toc93259832)

[Working Results 32](#_Toc93259833)

[Conclusions 32](#_Toc93259834)

1. [Software Design 33](#_Toc93259835)

[System Overview 33](#_Toc93259836)

[System Architecture 34](#_Toc93259837)

1. [GUI 35](#_Toc93259838)
2. [Testing 36](#_Toc93259839)
3. [Conclusions and recommendations 37](#_Toc93259840)

# Introduction

I chose to work with C#, since I used it for my previous project for ALE1 and so I can follow similar principles for my code structure. I also feel the most comfortable using Visual Studio and building a Windows Forms application as I’m familiar with the GUI toolset and IDE.

# Parse & Check DFA

## Process

### parse

First, I needed to parse the values given from a text file, by creating objects to hold them to follow an object-oriented structure.

The simple type of text files looks like this:

Graphical user interface

Description automatically generated with medium confidence**Alphabet**: The ‘language’ indicating the letters within it

**States**: All the states involved in the graph

**Final:** All the states that are final/accepting states

**Transitions:** All the transitions involved in the graph

Example: **q0, a --> q1**

**- q0** 🡪 State **from** where the transition **starts**

**- a** 🡪 **label/value** of the transition

**- q1** 🡪 State **to** where the transition goes

**DFA:** Indicates whether the graph is a DFA (y = yes, n = no)

**Finite:** Indicates whether the graph is finite (y = yes, n = no)

**Words:** Lists words to run through the system along with whether they are accepted (y = yes, n = no)

Therefore, I created the following object classes to represent the **Graph**, its **states** and **transitions,** shown in the figures below.

Text

Description automatically generated

Figure 1. Source code of Graph object and its properties

A screenshot of a computer

Description automatically generated with medium confidenceText

Description automatically generated

Figure 2. Source code of State and Transition objects and their properties

Before parsing, the file needs to be read, preferably by letting the user browse a file from a file explorer dialog to get the path, and then use the IO library to read all the lines of the text file:

 A screenshot of a computer

Description automatically generated with medium confidence

Figure 3. Button which opens a File Explorer dialog box to access a file to parse

Next, the graph would host the operations for parsing the file into the objects with the following method in pseudo:

public string ParseDefaultFile(string[] contents) {

// foreach **line** in the text file

// if line starts with ‘#’, ignore it (it’s a comment)

// if line starts with **alphabet**

// save the alphabet string to the corresponding string property of the Graph object (GO)

// if line starts with **states**

// initialize a string array and store each state separating the line based on the ‘,’ char

// foreach state name in the string array

// create a State object (SO) with the state name as the label, that it is not a final state, and pass null for transitions which will automatically initialize a new list of transitions

// Add the SO to the GO’s list of states

// if line starts with **final**

// initialize a string array and store each state separating the line based on the ‘,’ char

// foreach state name in the string array

// if the name matches a state label in the state list, set that state’s isFinal property to true

// if line starts with **transitions**

// for every line in the file’s lines starting from the next line

// if it is equal to ‘end’, break the loop

// split the line’s contents:

// Based on ‘,’ = Gets FROM state

// Based on ‘,’ and ‘-->’ = Gets VALUE

// Based on ‘,’ and ‘-->’ = Gets TO state

// Create SOs for the from/to states to create a Transition object (TO)

// Add the TO to the Transition list of the GO

// if line starts with **dfa**

// Store bool value based on the line split by ‘:’, true for ‘y’ and false for ‘n’ for the file

// Call to method that will check from parsed values if graph is a DFA and store the bool value

// if the file and the method have the same truth value, indicate it on the GUI

[continue in next page…]

Figure 4. Pseudo-code for Parsing a default text file (Part 1)

[…continued]

// if line starts with **finite**

// Store bool value based on the line split by ‘:’, true for ‘y’ and false for ‘n’ for the file

// Call to method that will check from parsed values if graph is Finite and store the bool value

// if the file and the method have the same truth value, indicate it on the GUI

// if line starts with **words**

// for every line in the file’s lines starting from the next line

// if it is equal to ‘end’, break the loop

// Split the line based on ‘,’ and store the word as a key

in the GO’s string dictionary of words along with its truth value

// Return a method which returns a string in DOT format (for Graphviz) based on the parsed states and transitions

}

Figure 5. Pseudo-code for Parsing a default text file (Part 2)

And this is the method that creats a DOT string for the parsed values:

Text

Description automatically generated

Figure 6. Method that returns a string in Graphviz DOT format, based on the parsed values.

### Check DFA

To check whether a graph is a DFA, it needs to follow these conditions:

* No empty characters
* Has an outgoing transition with every letter in the alphabet

Text

Description automatically generated

Figure 7. Source code for checking DFA.

First, we check if any transitions have an empty label. Then we get each letter from the alphabet uniquely and then check if there are any states with a number of outgoing transitions that is **not equals** to the number of unique letters.

If either of these conditions is true, then the graph is NOT a DFA.

## Working Results

### Parse

Using the debug output from C# diagnostics library, I was able to display and check the parsed values:

Graphical user interface, text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

Figure 8. Debugging parsed values, showing textfile contents and the DOT format equivalent.

Diagram

Description automatically generated

Figure 9. Successfully displaying graph using Graphviz.

### Check DFA

The figures below show the GUI showing a ‘tick’ or an ‘x’ when a graph is a DFA or not respectively, and have a gray background if the file is wrong.

**Figure 10. a DFA graph and showing whether it is one in the GUI.**

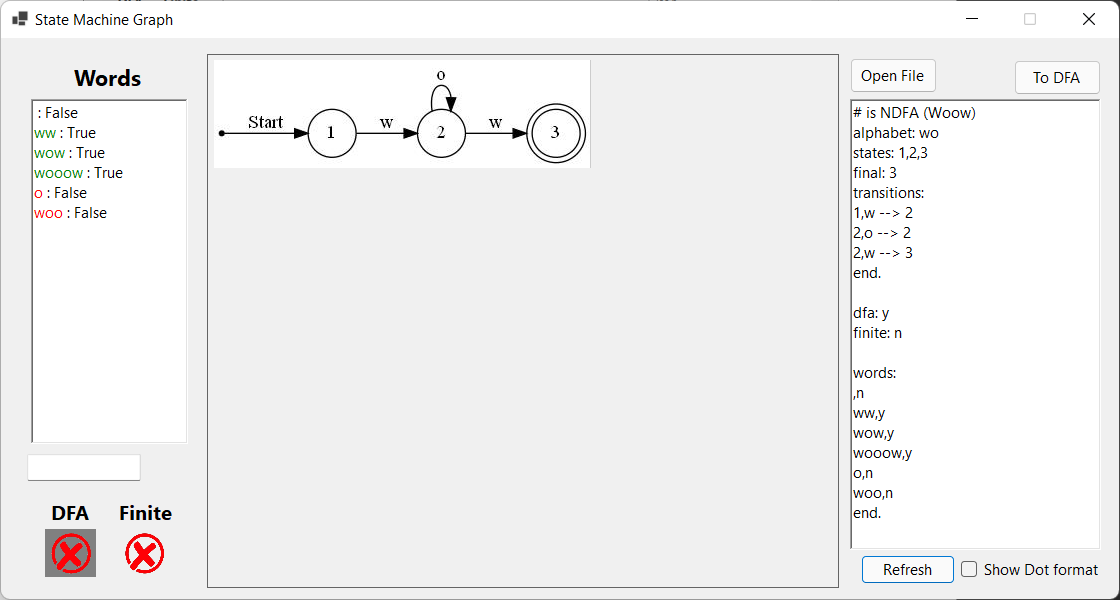
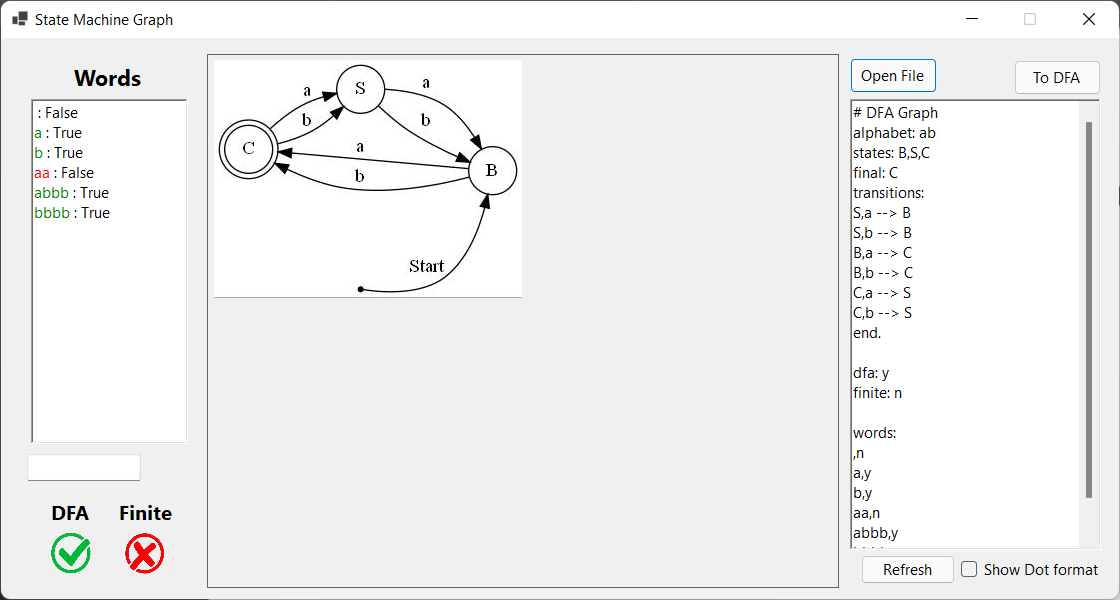


Figure 11. a NDFA graph with the file wrongly saying it is a DFA

## Conclusions

# Accept String – Word Check

## Process

All the words that have been parsed need to be iterated through and call a recursive method to check each word.

Text

Description automatically generated

Figure 13. Source code of CheckWords() method that calls the recursive method CheckWord().

public bool CheckWord(string word, int letter\_index = 0)

{

// If starting new word, clear stacks and push first state

// Check possible transitions from current state (state on top of stack)

// If there are possible transitions

// foreach transition in possible transitions

// push the state the transition is point to, to the global state stack

// if the letter is the last one in the word

// if the state from the stack final?

// return true

// else

// pop stack to previous state

// else

// Call CheckWord() with next letter (letter\_index++)

// if it returns true

//return true

// else

// pop stack to previous state

// decrement letter\_index (letter\_index--)

}

Figure 12. Pseudo-code of recursive method CheckWord().

## Working Results

Displayed the process in debug diagnostics:

Text

Description automatically generated

Figure 14. Debugging of word checking in output terminal.

Diagram

Description automatically generated

Figure 15. Shows when a word is correct by the word’s text color and shows red color on truth value when file is wrong.

The user is also able to check a single word they input in a textbox below, which checks the string inputted every time the value in the box changes. It indicates whether it is accepted or not using a ‘tick’ and ‘x’ image next to it:

A picture containing graphical user interface

Description automatically generatedA picture containing graphical user interface

Description automatically generated

Figure 16. Checking if user input is an accepted word.

## Conclusions

This was slightly tiring, as a lot of little things needed to be taken into account to make sure it covers as many cases as possible.

# Regular Expression

## Process

The operators used in these regular expressions are shown in the figure below:

Text

Description automatically generated with medium confidence

Figure 17. Regular expression and their ASCII equivalent.

The come with these equivalent state machine layouts:

Diagram

Description automatically generatedDiagram, schematic

Description automatically generated

Diagram

Description automatically generated

Figure 18. Regular Expression State Machine Layout examples.

Due to my previous work in ALE1, I used a similar structure of a **NodeRegexManager** that controls the nodes, with node **OperantRegex** and **OperatorRegex** classes implementing an **INodeRegex** interface class.

Text

Description automatically generated

Figure 19. INodeRegex interface properties.

Text

Description automatically generatedText

Description automatically generated

Figure 20. OperatorRegex and OperantRegex classes and their properties, methods.

Text

Description automatically generated

Figure 21. NodeRegexManager class and its properties, methods.

The pseudocode for the parsing of the nodes from a regex formula is:

private void AddNode(string formula){

// for every character in the formula

case '(':

// Add children

AddNode(formula);

case ')':

// Last parent has no more children to add so remove from stack

this.parent\_stack.Pop();

return; // to exit from child method (of recursion)

case ',':

// Just skipping works

case: // Operants/Operators

// Add the nodes to their respective list (Operants/Operators)

// Add this node as child to parent in global ‘parent stack’

// Push this node in parent stack if operator

// Add it to the global list of nodes

}

Figure 22. Pseudocode of AddNode() method.

Using the layouts, I was able to build a recursive method that parses the graph properties (eg., alphabet, states, transitions) using the parsed nodes from the regular expression:

public string ParseRegexFile(string[] contents)

{

// Create a NodeRegexManager Object (NRM)

// foreach line in string array contents

// if line starts with ‘**#**’

// ignore

// if line starts with **‘regex’**

// split the string by ‘:’ char and pass the string representing the regex formula to AddNode in NRM to parse the nodes

// Call a recursive method to parse the nodes into the Graph object (GO)

// In regex, always set the last state in the list as the final state

// add the state to final states list

// foreach operant in the operants list in NRM

// parse letter into GO’s alphabet string

// if line starts with **‘dfa’**

// Split the string by ‘:’ char and pass true if ‘y’ or false if ‘n’ and store what the file says.

// Call method to check if graph is a DFA (CheckDFA() method) and return a bool.

// if the file does not agree with method result

// show it in the GUI

// if line starts with **‘finite’**

// Split the string by ‘:’ char and pass true if ‘y’ or false if ‘n’ and store what the file says.

// Call method to check if graph is Finite (CheckFinite() method) and return a bool.

// if the file does not agree with method result

// show it in the GUI

// if line starts with **‘words’**

// for every line in the file’s contents starting from the next line

// if its equals to ‘end’

// break loop

// Split the string by ‘,’ char for each word

// Split the string by ‘,’ char for each word’s

truth/accepting value: true if ‘y’, false if ‘n’

// Return a DOT format based on parsed GO values

}

Figure 23. Pseudocode of ParseRegexFile() calling ParseRegexContents recursive method.

**ParseRegexContents() - Initialization**

Text

Description automatically generated

Figure 24. Source code of recursive method ParseRegexContents() (Part 1)

**ParseRegexContents() - Concatenation**Text

Description automatically generated

Figure 25. Source code of ParseRegexContents() (Part 2 - Concatenation)

**ParseRegexContents() – Choice**

Text

Description automatically generated

Figure 26. Source code of ParseRegexContents() with left child operations (Part 3.1 - Choice)

Text

Description automatically generated

Figure 27. Source code of ParseRegexContents() with right child and rest of operations (Part 3.2 - Choice)

**ParseRegexContents() – Loop**

Text

Description automatically generated

Figure 28. Source code of ParseRegexContents() (Part 4 - Loop)

## Working Results

Example: **.(a,|(b,c))**

Diagram

Description automatically generated

Figure 29. Regex graph result of concatenation with choice right child.

Example: **|(|(a,b),c)**

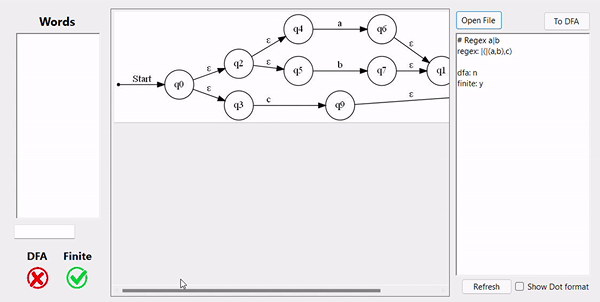


Figure 30. GIF of Regex graph result of choice with choice left child.

Diagram

Description automatically generated

Figure 31. Regex graph result of loop.

## Conclusions

Regex was easier to implement than the rest so far but had some issues with passing the right parameters within the recursive method at first, until I realized I had to pass an ‘open’ and ‘final’ state for each recursion to know with what state to start growing the transitions from and where to end them.

It also made me realize something missing from my word checking algorithm. It would check for empty transitions until it found the desired letter, but if that state was not final, the word got rejected. However, it needed to check for more empty transitions from that state that might lead to a final state.

Text

Description automatically generated

Figure 32. Added code to fix word checking due to regex graph.

# Finite

## Process

For a state machine (SM) to NOT be finite, it needs to follow these conditions:

* If SM **has a self-loop with a transition to a final state**, it is infinite.
* If **a final state is a part of a cycle** (states that are all connected through each other)

Thus, I added these properties in the Graph object:

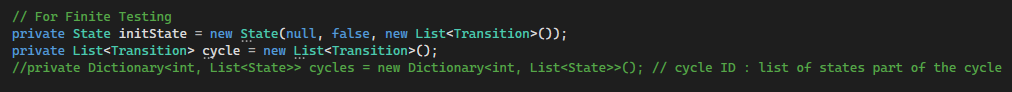


Figure 33. Graph Finite related properties.

The first figure below shows the **CheckFinite()** source code:

Text

Description automatically generated

Figure 34. Source code of CheckFinite() method.

The following figures show each method that returns a bool value based on whether the finite condition is true or false:

Text

Description automatically generated

Figure 35. Source code for method checking if a state has a path to a final state.

Text

Description automatically generated

Figure 36. Source code for method checking if state is part of a cycle.

## Working Results

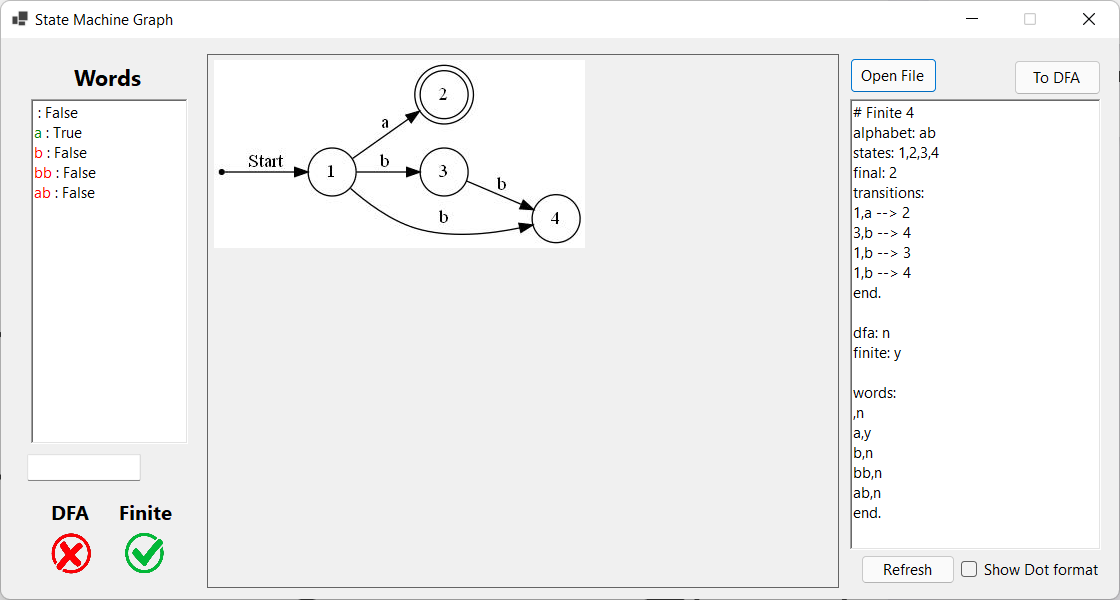


Figure 37. GUI showing Finite graph result (correct file).

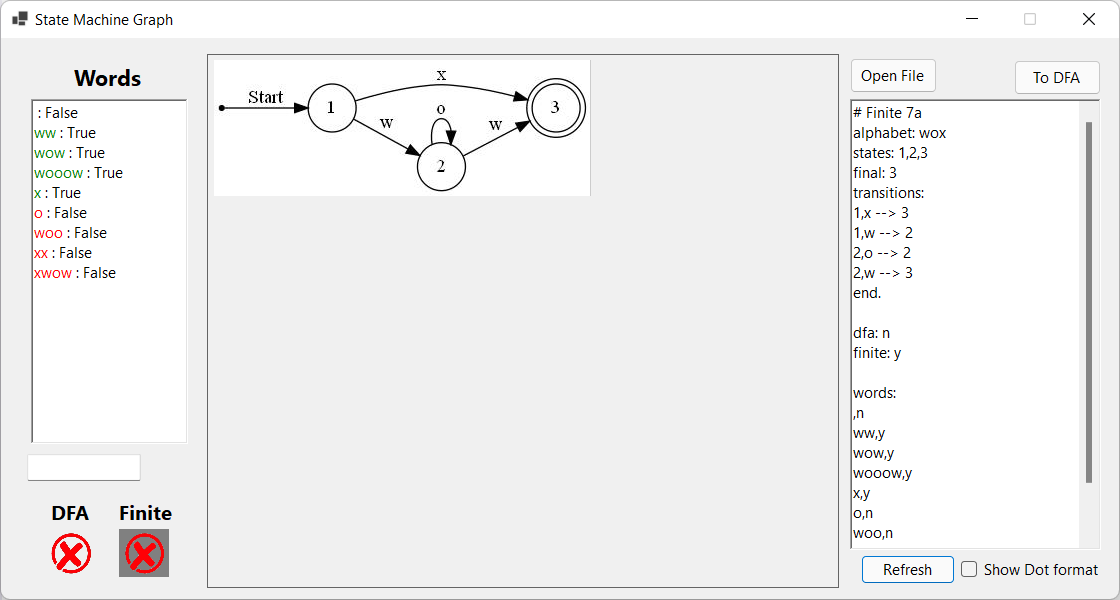


Figure 38. GUI showing an Infinite graph result (wrong file).

## Conclusions

This seemed to be one of the hardest tasks to complete, but when conditions are this clear and when the code is organized well, it becomes a lot simpler to map the process in your mind and implement it.

# NDFA

## Process

Added the following Graph properties for this:

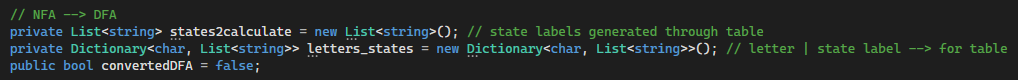


Figure 39. Graph properties added related to NDFA to DFA conversion.

public string NDFA2DFA()

{

// Store a temporary copy of the final states

// Call method CheckForEpsilonClosures() that checks for epsilon closures and reforms the states labels

// Initialise groups dictionary var (based on letters from alphabet)

// Add the initial state to var holding the calculated states to start the table (label)

// Call the recursive method NDFA2DFA\_CalculateState() and return a bool value to show that it has completed successfully

// If completed successfully

// Clear all state and transition lists

// foreach string state in newly calculated states

// Set states to final if they contain a state in their label that was previously a final state

// Initialize a new State object (SO)

// if the string state is equals to ‘SINK’

// create a state with the name SINK and create self-loop transitions with each letter of the alphabet to this state

// else

// create a state with the string state label

// if the new SO is not null

// if it’s final

// Add to final\_states list

// Add to states list

// Create/Add appropriate transitions between the states

// foreach letter in the alphabet

// foreach state from the states list

// Point to the state with the same label from the letters\_state dictionary

// Create a transition for that letter

// return CreateDotFromParsedFile(), a string in DOT format

}

Figure 40. Pseudocode of NDFA2DFA() method which calls recursive method NDFA2DFA\_CalculateState().

private bool NDFA2DFA\_CalculateState(State curr\_state, int tb\_index = 0)

{

// foreach letter in the alphabet, fill in the table row for the current state

// if state has a multi-state label (e.g., {q1,q2,q3})

// Check transitions for all the sub-states

// else

// Check transition for single state

// Once we know all possible transitions, form the states as needed

// foreach transition in possible transitions

// Check if state was already written (avoid duplicates)

// Add the state the transition is pointing to, to the new state's label as sub-state

// Add state to letters\_states dictionary under the letter

// If not possible transitions, add ‘SINK’ under the letter

// Check for newly formed states in letters\_states

// foreach letter and its list of strings in letters\_states dictionary

// If there are no states with the same label as the state we're checking

// Create new state (for future checking) // Add new state to table (states2calculate)

// call this method again with new state looking at next table index (based on number of states in states2calculate)

}

Figure 41. Pseudocode of NDFA2DFA\_CalculateState() method.

Text

Description automatically generated

Figure 42. Source code for CheckForEpsilonClosures() method.

## Working Results

I messily show on the output terminal the ‘drawn’ table based on the conversion:

Graphical user interface, text

Description automatically generated A picture containing text, device, meter, gauge

Description automatically generated

Figure 43. Debugging NDFA to DFA table on output terminal.

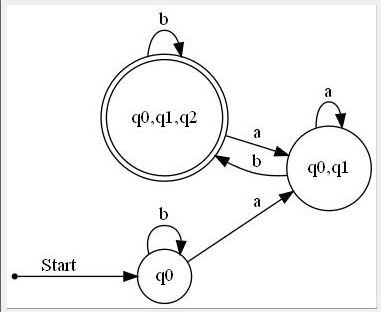
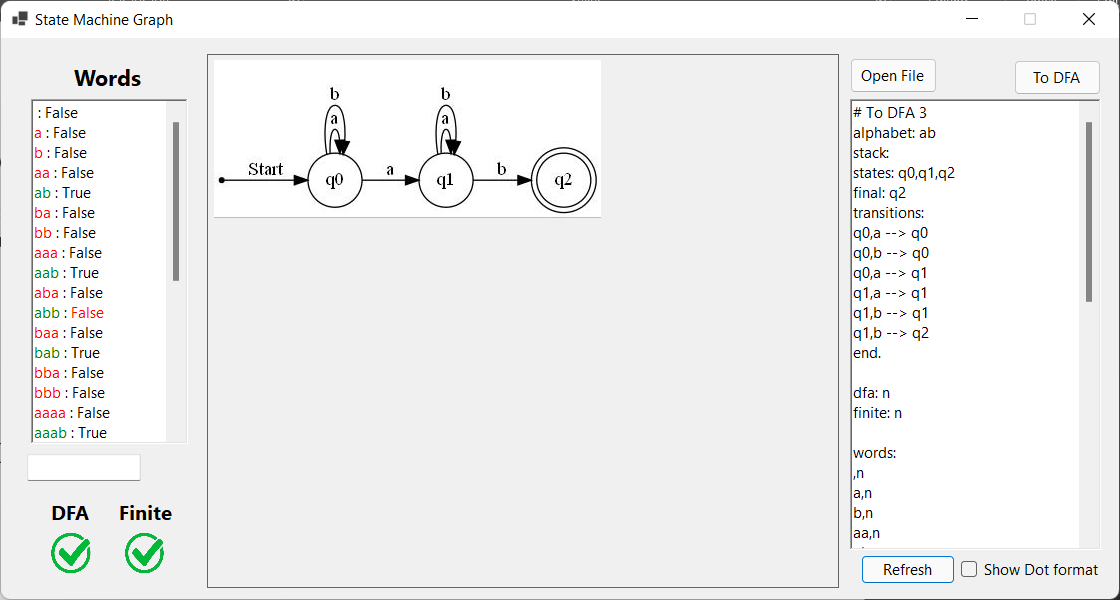


Figure 44. GUI showing conversion of an NDFA to DFA (no SINK).

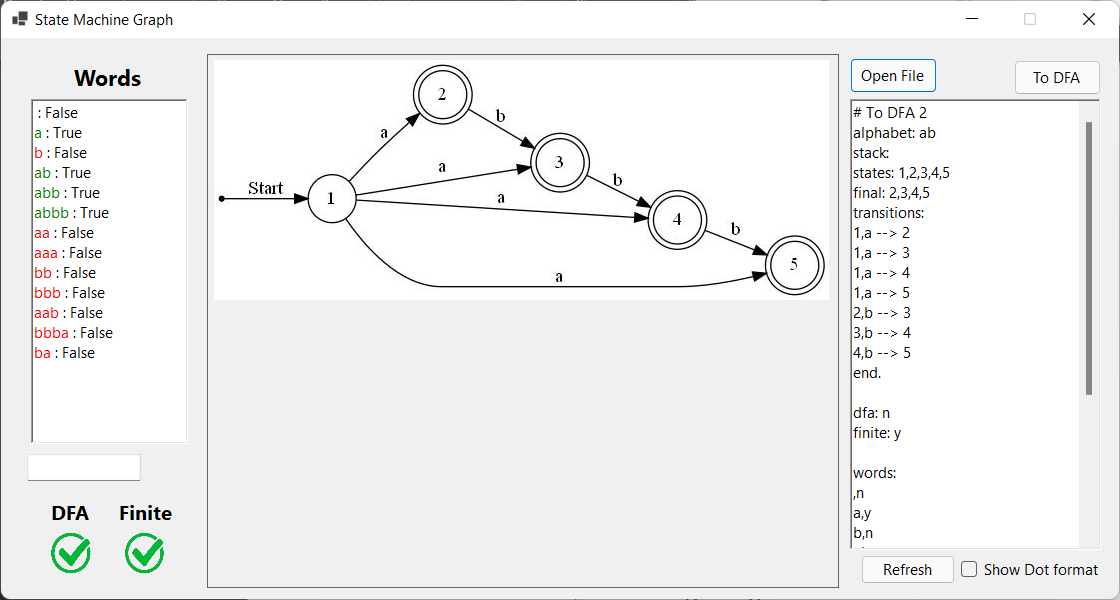
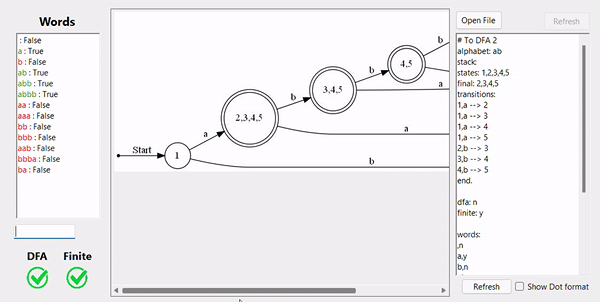


Figure 45. NDFA to DFA with SINK (GIF).

## Conclusions

This was the task that took the longest, mostly because building a table in recursion can become confusing. However, the logic on how to create the table was straight forward and eventually covered more cases than I thought I had time for.

# PDA

## Process

The following properties were added to the Graph and Transition object:

Graphical user interface, text

Description automatically generated

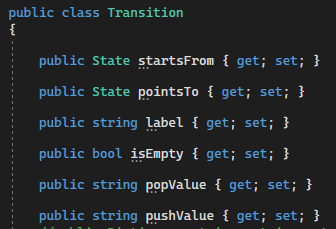


Figure 46. Graph and Transition properties added related to PDA.

public string ParsePDAFile(string[] contents)

{

// Set this graph as PDA (global bool var)

// foreach line in file contents

// if line starts with ‘#’

// ignore comment

// if line starts with ‘alphabet’

// parse string to Graph object’s (GO) string var alphabet

// if line starts with ‘stack’

// parse string to GO’s string var stack\_val

// if line starts with ‘states’

// parse each as a State object (SO) using the given label

// if line starts with ‘final’

// foreach state in states list

// set state as final if exists in parsed string

// if line starts with ‘transitions’

// for every line of the contents starting from next line

// if starts with ‘#’ ignore comment

// Split string based on additional characters ‘['

and ‘]’ to get additional pop and push values for each transition

// if line starts with ‘words’

// for every line of the contents starting from next line

// stop loop if ‘end’

// skip if line is empty

// Split the word and its truth value based on ‘,’

Char

// return a string by calling a method to get a string in DOT format

}

Figure 47. Pseudocode for ParsePDAFile() method .

Check words for PDA, added conditions:

* Accept word only if stack is empty
* Check pop and push values for every possible transition (pt)

Text

Description automatically generated

Figure 48. Source code for added code in CheckWord() method.

## Working Results

A picture containing diagram

Description automatically generated

Figure 49. GUI shows PDA graph.

A picture containing diagram

Description automatically generated

Figure 50. GUI shows PDA graph with file wrong about accepted words.

## Conclusions

This took the least amount of time to complete, probably because I am very familiar with stacks, since I’ve been using them for a lot of the ALE courses methods I made.

# Software Design

## System Overview

The project is a Visual Studio Win Forms program in C#. It’s formed using Object Oriented Design principles to store and manipulate its data.

## System Architecture

Diagram

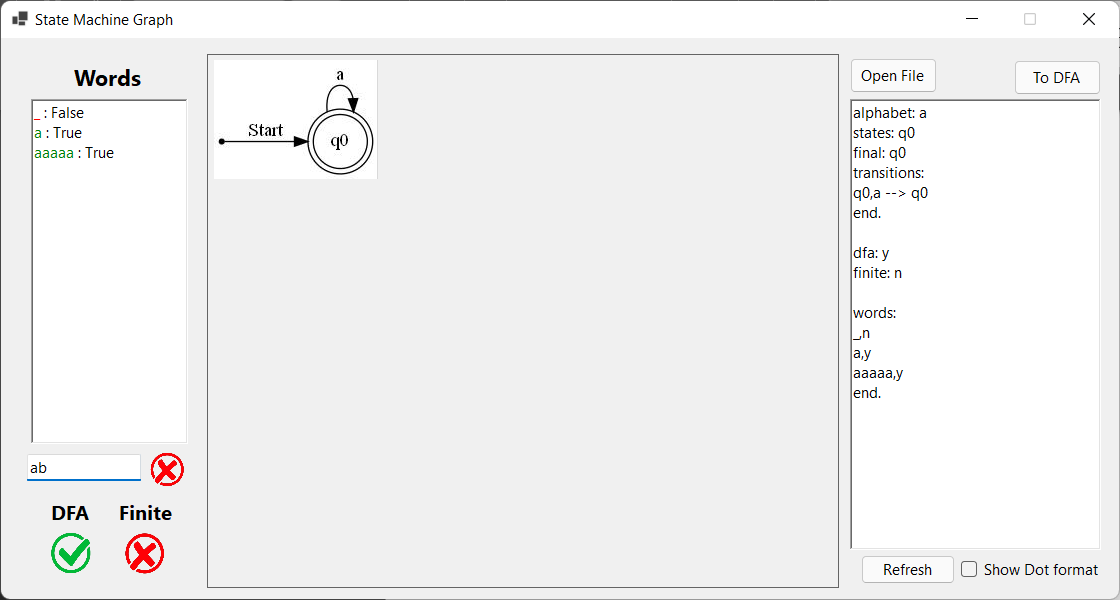
Description automatically generated

Figure 51. UML

# GUI

Below is an explanation of the graphical user interface:

Figure 52. GUI Explanation



Words from the file appear here with their truth states (accepted or not):

Green/Red Letter = accepted or not

Green/Red Truth State = File is right/wrong

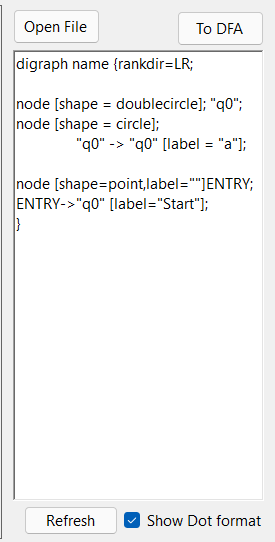
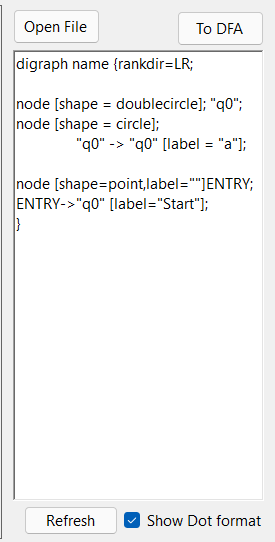
Graph

User can type custom words to check if they are accepted by the system.



Shows if it is a DFA and if it is a Finite system.

If the file is wrong, the image displays grayed out.



Shows contents of file

Opens a text file

Convert current graph into a DFA

Re-draw the graph with the current file contents

# Testing

# Conclusions and recommendations