2B NOR !2B

That is the question.

Abstract

This document covers the complete process of creating the application for which this document is named after. The application is a learning tool for GCSE and A-level Computer Science students to improve their understanding of logic gates and Boolean algebra.

Andreas Mullen

2024

Contents

Analysis

Background

Logic gate circuit diagrams (Logic diagrams) are commonly used to represent Boolean expressions in a simple and understandable manner. The diagrams are also used to show how states are transmitted within a Boolean expression when the expression is evaluated.

Due to their widespread use, sections within both the A-level and GCSE Computer Science specification are dedicated to logic gates and Boolean expressions to make sure students can understand and use them within context. This is shown in the AQA A-level specification, section 4.6.4, where students must "Draw an equivalent logic gate circuit for a given Boolean expression". Along with this, students must also be able to interpret truth tables and perform Boolean algebra to simplify Boolean expressions that are given within questions.

The Bishop's Stortford High School is a state school with a focus on Maths and Computer Science. Within the school, Computer Science, and more specifically logic gates, are taught in two main stages of increasing complexity being GCSE and A-level.

Key definitions

Term	Definition
State	A Binary value. Examples include 'true', 'false' and 0/1.
Element	Any component that makes up a logic diagram. These can be logic gates, input/output pins or wires.
Logic diagram	A visual representation of a Boolean expression. It is made up of logic gates, input/output pins and wires.
Logic gate	A visual representation of a logical operator, such as AND. The gate carries out its operation with its input and gives out an output. The inputs are provided either by other gates or an input pin, which is shown through the state of the connected wires.
Output pin	An element that receives the output of the last operation of the diagram. It indicates the output state of the Boolean expression for a given input state.
Input pin	An element with a toggle able state. Depending on its state, the pin takes a Boolean value. This can be used as the input of other gates.
Wire	A visual component that shows the Boolean state between elements within the logic gate diagram. The state of the wire is dictated by the input element.
Truth table	A table that maps all possible inputs of a Boolean expression and their respective outputs. Intermediary steps are also added to show how states change within the sub-expressions to produce the output.
Karnaugh map	A method for minimising Boolean expressions that works by forming groups of minterms and identifying valid subexpressions to produce those groups. These sub-expressions are then simplified to produce the minimised result.
Quine-McCluskey algorithm	A method of minimising Boolean expressions by find prime and essential prime implicants from an expressions' minterms.
Petrick's Method	An extension to the Quine-McCluskey algorithm which allows expressions to be minimised even if the essential prime implicants do not cover the entire Boolean function.
Minterm	The denary value of the input columns within a truth table that result in an expression evaluating to one. Given an AND gate, two would be a minterm as '11' evaluates to true when inputted to an AND gate.
Prime implicant	A grouping of binary numbers that cover a series of minterms of a Boolean expression. It acts as a general form for the binary representation of denary numbers.

Essential prime implicant	A Prime implicant that is the only implicant to cover a minterm. This means the prime implicant is the only one of the same form as the binary representation of the minterm.
Boolean expression / algebra	A logical statement that results in a binary value. Boolean expressions have their own form of algebra which allows the expressions to be manipulated.

Current System

There are a few different ways of drawing logic gate diagrams within the school.

- Drawing by hand

This is the most common method of drawing logic diagrams and is used by both teachers and students. The process that someone would take drawing a logic diagram by hand is generally the same for both teachers and students. This is done as follows:

- Draw and label the inputs of the logic diagram on the left side of the page/whiteboard and then add the outputs on the right side, these would also be labelled. Generally the labels for the inputs are capitalised and alphabetical starting from 'A', which is assigned to the top-most input on the page/whiteboard. Common labels for outputs include 'Q', 'S' and 'R'.
- Draw the required logic gates onto the page/whiteboard from the question/Boolean expression etc. It is then up to the teacher/student to place the gates in a position where each gate is clear and spaced out from other gates to result in a diagram that is good-looking and neat/clear.
- The teacher/student would then connect the inputs, outputs, and gates with wires (straight lines) to produce a completely connected diagram. Again, it is the teacher's/student's responsibility to make sure that the wires are neat and straight. It is also noted that line segments that make up a wire cannot be diagonal, meaning that line segments are perpendicular to each other.
    - Arcs are also drawn where needed. These are added when a wire segment intersects with another segment of a different wire.
    - Shaded circles are also added to show a wire junction. These are added when a wire segment intersects with another segment of a wire going to the same input.

- Using Software

The school also uses software, such as Logisim, within lessons for students to construct larger and more complex logic diagrams. Students can create logic diagrams from Boolean expressions by simply inputting them into a dialog box. Students can also interact with their diagram using the 'hand' tool to see outputs of their diagram based off of the input. After creating the diagram in Logisim, students can also generate a truth table of the circuit. Software is also used by the teachers to quickly demonstrate new concepts to their students such as the Boolean identities for expression simplification. Once the student/teacher is complete they can save, load, or export the diagrams that they have created. It should also be noted that a teacher must dedicate some lesson time to teaching students how to use the software.

Clients/users

My client is the Head of the Computer Science department within the school and hence, oversees the operations of the department. They are also responsible for bringing new software for teaching within the department. Logically, they are my client as they have the knowledge of the qualities that software must possess to be effective teaching/learning tools.

Prospective users

There is a wide range of prospective users for this system; the students have been broken down into their key stage so that they can be considered separately. This is because some parts of the Computer Science specification for logic gates' differ between each age-group:

- Teachers (referring to any Computer Science teacher)
- Students
    - GCSE students (referring to any Computer Science student in year 10 or 11)
    - A-level students (referring to any Computer Science student in year 12 or 13)

Teachers will primarily use the system for demonstration of key concepts within logic gates by drawing logic diagrams that represent simple Boolean expressions, the employ the key concept being demonstrated. They would also need to create simple truth tables with the diagrams that they have drawn, for teaching the students about truth tables. Teachers will also be required to deal with Boolean expressions and how they interact with truth tables and diagrams. Due to a Teacher needing to show their students how to interact with software, they require a simple user interface. This is to make it as easy as possible to show students how to use the system.

Although GCSE students do not cover Boolean expressions in much detail; they are expected to be able to draw simple logic diagrams and be able to interpret and fill in given truth tables, which are the major use-case of the program. Students should also be able to trace through logic gate diagrams and find outputs from them, which is also a requirement for the program. It must be noted that GCSE students only cover the AND, OR and NOT gates.

A-level students will also use the program in a similar manner; however, they will also make use of minimisation as they are expected to be able to simplify expressions within exams. A-level students must also be aware of NAND, NOR and XOR gates which builds upon the GCSE specification. They must be able to draw more complex diagrams involving, more layers of logic gates, larger number of inputs and potentially more than one output.

It should also be noted that both students of all age groups and teachers will use the saving and loading portion of my system as they are general-purpose features. That are required for all groups of prospective users.

Formal method of analysis

To gain insight into the necessary qualities of the system from a teachers' point of view, I have conducted an interview with the client. The interview has been recorded and a transcript has been written which can be found in appendix A. In addition to this, I have created a questionnaire for students to gain their insights, as they are the primary users of the system. The questions and results can be found within appendix B. The conclusions of the questions are given below:

Conclusions from the interview

My interview with the client gave some useful insight into some of the problems with the current way of teaching and also some of the tools within my system that would make it effective as a learning tool. My conclusions are discussed below.

1. Students should have the same interface as the teachers so that they can follow demonstrations more easily. Currently the interface within current systems (such as Logisim) make it very difficult for students to follow the teacher, and hence, make it more difficult to learn the taught material as the student spends time trying to find the correct menus and buttons and not understanding the material.
2. The program should not obscure features that are above a particular student's ability because able-students should be able to access content that is above their age group. This will allow for more able students to learn more in-depth and around the subject which will further their understanding of the material on the specification.
3. The user interface of Logisim, the main tool used by the client, is far too complex because lots of features are visible to the user at once, this can quickly cause users to become confused when using the software. This is partly due to the unintuitive menu-design and the support of many features that are beyond the A-level specification. Features within Logisim are also nested between many layers of menus. This makes it difficult for an inexperienced user to find the feature they would like to use, creating further confusion as a result. To solve this problem:
    a. Few layers of menus should be used, and features/buttons should be clearly labelled on what they do.

   b. The main window of the application should be simple in its design so as to avoid overloading the user with many different features which they may be unaware how to use them or what they do in the first place.

4. The program should be made up to the current A-level specification and not include content that is beyond it. This is because it is very easy for student to find information for logic gates that are beyond their current level. This could unnecessarily confuse the student further making it more difficult for them to understand the material they are trying to learn.
   a. This means that program testing should be extended to allow the client to conduct a review of the program to ensure that it covers all of the necessary bases for a students' learning.
   b. All of the content within the program should be moderated to ensure that it is of the right difficulty for the students it pertains to.
5. More help should be given to students when they are drawing logic diagrams, this is so a student can produce much more visually appealing diagrams which combats the disadvantages of drawing logic diagrams by hand. Some methods to aid students could include:
   a. Faint lines that guide the students as to where they should place their logic diagrams' elements and wires.
   b. A feature that automatically aligns a diagram and redraws the wires so that they are as clear as possible.
   c. A focus should be taken on making the program as clear as possible, meaning everything must be labelled and easy to understand. This is to make the program as easy to use as possible which means it is easy for students to follow a teacher's demonstration.
6. The program should be as versatile as possible when it is exporting logic diagrams, this is to make the program as useful as possible for both students and teachers. This includes:
   a. The possibility of exporting logic diagrams with transparent backgrounds in the most common file formats such as .png and .jpeg so that they can be placed into documents easily and without creating formatting problems within the document.
   b. The ability to add truth tables to exported diagrams. This, however, is not an essential feature of the program, as tables can be made in other programs very easily and there it is not essential for a student/teacher to save and record tables.
7. Questions are not the most essential tool for learning logic gates. Although practice questions are very important, these tasks should be completed by hand with pen and paper. This means that the program should focus on the techniques that a student would be required to replicate within an exam, such as drawing diagrams and creating truth tables.
   a. If questions are within the program, they should be as varied as possible to test the students as much as possible. Possible question types include:
      i. Filling in incomplete diagrams
      ii. Finding Boolean expressions from diagrams
      iii. Filling in a truth table from a diagram.
   b. Questions should be a separate section or even a different program. This is to make the program as simple as possible for students. If this branch of features is added to the program, it makes it easier for students to become confused and so the menu option should be clearly marked and easy to close, if opened.

Conclusions from the student questionnaire

This questionnaire has been answered by all of the subgroups of students and my extracted conclusions are discussed below. The goal of my questions was to highlight the issues that students had when drawing circuits with a variety of different methods, I also wanted to gain insight into what they would like to see in a new system.

1. Students prefer a more minimalistic design because it makes the software more accessible and clearer (see question 2 in appendix B). This means that one of the program's objectives is to be accessible as possible.

One of the methods of doing this is to reduce the number of clicks a user has to make. This can be done with more buttons in each window and fewer sub-menus.

2. Drawing logic diagrams by hand is generally preferred by students as it is faster, however:
    a. Logic diagrams are messy and untidy. Gates within the diagrams are also very inconsistent making the diagram difficult to read.
    b. Errors within hand-drawn logic diagrams are difficult to remove.
    c. Drawing / using multiple, large, or complex diagrams can quickly become very time-consuming and boring. When using the new system, a focus should be placed on making diagram drawing as easy as possible, which is a major objective of the program.
3. Logisim is used by the students but it does have some flaws:
    a. The GUI is cluttered and extremely complex from excessive buttons and menus. This makes some features less accessible for inexperienced users, making the program more difficult to use.
    b. The software provides no aid to the user when they are using their diagrams (such as how to generate a truth table). If the program provides resources to the user, they should be easy to find, and comprehensive.
4. It will be useful to have areas where help can be given to the students. This makes the program easier to use and will improve the user experience. This is because a student is less likely to get frustrated when using the program as 'things just work'. Some examples in which help can be delivered to the user include:
    a. Help pages on that describe specific logic gates. This can be extended to prebuilt circuits and Boolean expressions for A-level students. This would help the student understand the material they are dealing with which helps them make full use of the programs' functionality.
    b. Sections that explain how to use the software, such as where particular features are and how to use them. This will help users that get stuck when using more complex features within the program, such as diagram minimisation.
5. Simpler / lower level material, such as the basic AND, OR and NOT gates, are not covered as well as higher level material such as Boolean expressions. This was mentioned briefly in the interview with the client (see appendix A) but also within question 7 of the student questionnaire (see appendix B):
    a. Younger students have a more difficult time with current systems as the resources they have access too do not cover the fundamental concepts well enough. This means that, aside from asking the teacher, it is difficult for them to find a useful reference. This means that a requirement of the program is to provide detailed resources on the fundamentals of logic gates, to alleviate this problem.

Problems with the current system

There are several issues with drawing logic diagrams when looking at the two current methods of producing them.

Drawing by hand

- Diagrams that are drawn by hand are not neat unless lots of time is taken to draw them. This becomes a problem when a student or teacher must draw lots of diagrams within a lesson or for revision. This means that a student or teacher must decide if they would like to draw fewer logic diagrams that are neater or sacrifice neatness for a higher quantity given the same amount of time.
    - There is also not a standardised method of logic drawing diagrams by hand, meaning that when a teacher/student draws logic gates these can be different sizes and shapes. This results in a messier logic diagram, which does not look as visually appealing.
    - Onus is also placed onto the student/teacher to make sure that the diagram is aligned and nicely fits onto the page.
    - If a student/teacher makes a mistake while drawing their diagram, these can be difficult to remove with a rubber when using a sheet of paper. This is easier on a whiteboard, but can be difficult for large diagrams.

- When the student/teacher has completed drawing their diagram, using it can be very as time-consuming (such as producing a truth table for a particular diagram). A student/teacher would want to see the table immediately and not spend their time tracing through the diagram which is what would be required.
    - Drawing diagrams by hand also have the drawback of not updating the users' inputs in real-time. This means that a student/teacher must trace through the diagram every time they want to see an output for a set of inputs for a particular diagram. Unless they record all intermediary stages of the trace. This is impractical for when someone wants to change the states of the logic diagram often.
- The same can be said for Boolean algebra and logic diagram simplification. This is very time-consuming and complex to do, whilst also being error-prone. This is because some students find it difficult to find the correct Boolean identities to use, to achieve a correct final answer. There is also no set process for simplification making it possible to achieve incorrect answers that are simply incomplete and not incorrect logically.
    - Being able to quickly simplify Boolean expressions will be useful for users as they can quickly check whether or not they got the correct answer. This will aid the students in their practise of Boolean simplification which will improve their ability to answer questions within the exam.
- Drawing diagrams by hand makes it very difficult for students/teachers to share, edit and save work. It is impractical for students/teachers to keep every single sheet of paper that they have used, whilst also sharing it around a classroom.

Using software

- Using Logisim is too complex because it is made for use cases beyond the scope of A-levels.
    - There are components within Logisim that are not for Logic circuit design such as RAM and ROM. These clutter the program and make it more confusing for inexperienced students; making it more difficult to draw logic diagrams.
    - There is a very large menu hierarchy within Logisim due to the number of features that it has. This means that it is very easy for important features to be hidden from the user (such as diagram minimisation). This means that an inexperienced user would not be able to access the full functionality of the program without spending a significant amount of time learning the software. For teachers and students this is not an option as it takes away from teaching and learning time.
    - The GUI of Logisim is simple and minimalistic to point where it makes trying to find tools and components very difficult. A system that is more clearly labelled would it make it easier for a student to find what they want.
    - Since October 2014, development of Logisim has been suspended indefinitely. This creates issues such as the program having an outdated user interface which makes the program more difficult to use as it gets older. Also, as the GCSE and A-level Computer Science specifications get updated, there is a potential for Logisim to not have the required features within it. This creates a need for a newer program that reduces the age difference between the specification and the program itself. This will keep software as a helpful tool for both students and teachers and ensuring it is relevant to the specification.
- Software does not provide help guide students when they make a mistake. This makes it difficult for them to see their errors and ultimately learn from them. A system that could help students making their logic diagrams would reduce the chances of making a mistake and could potentially make the process more enjoyable as the student spends less time "fighting" with the software and more time making diagrams.

Modelling of the problem

Current system

The data flow diagram below shows the process that a student would take when drawing a logic circuit diagram from a given Boolean expression. This is a common exam question for some GCSE and all A-level students.

The main issue with this method is that there are multiple steps that a student would carry out using trial and error, which could result in drawing a logic diagram multiple times. An example of this is when a student is drawing the

wires to connect the gates in their drawn diagram. This could result in a situation where the student believes that they are correct, only to realise this is not the case and so they have to redraw the logic diagram, which is time-consuming and creates a messier logic diagram due to them having to either erase or cross out the incorrect element.

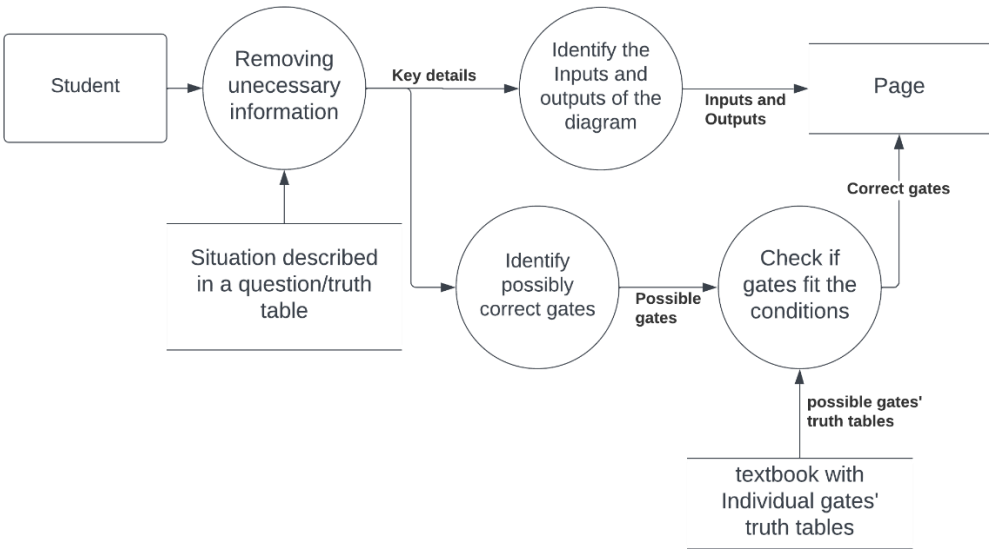


Figure 1: Drawing a logic diagram by hand from an exam question with a Boolean expression.

The data flow diagram below shows the process of drawing a logic diagram from a Boolean expression within Logisim. This method has no real issues and is quite effective to students/teachers once they know how to use it. I would like to implement something similar to this where my implementation, makes it simpler for the user to draw logic diagrams from expressions.

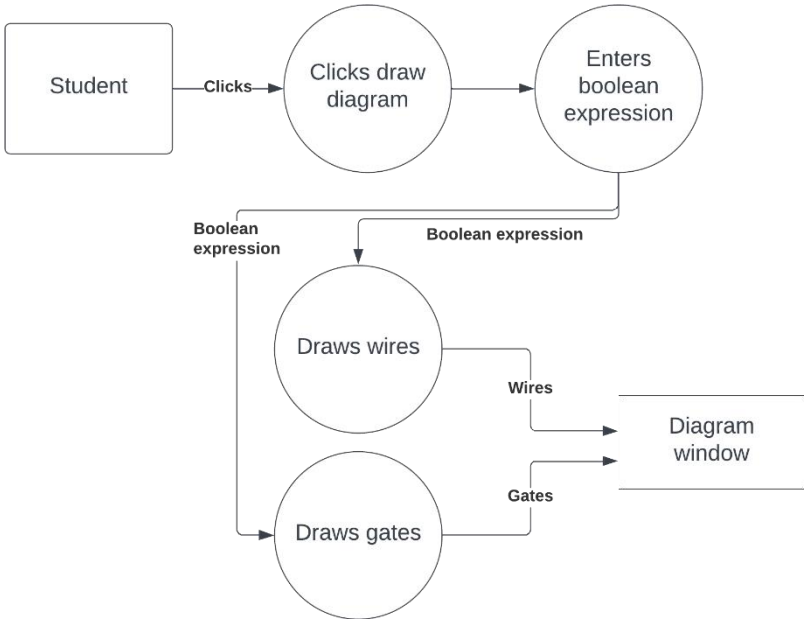


Figure 2: Drawing a logic diagram within Logisim.

Proposed system

The data flow diagram below shows the outline of my proposed system. It covers the fundamental diagramming portion of my solution. This involves generating diagrams, saving/loading diagrams, and also exporting images of logic diagrams. Other processes such as generating truth tables and expression minimisation have also been included.

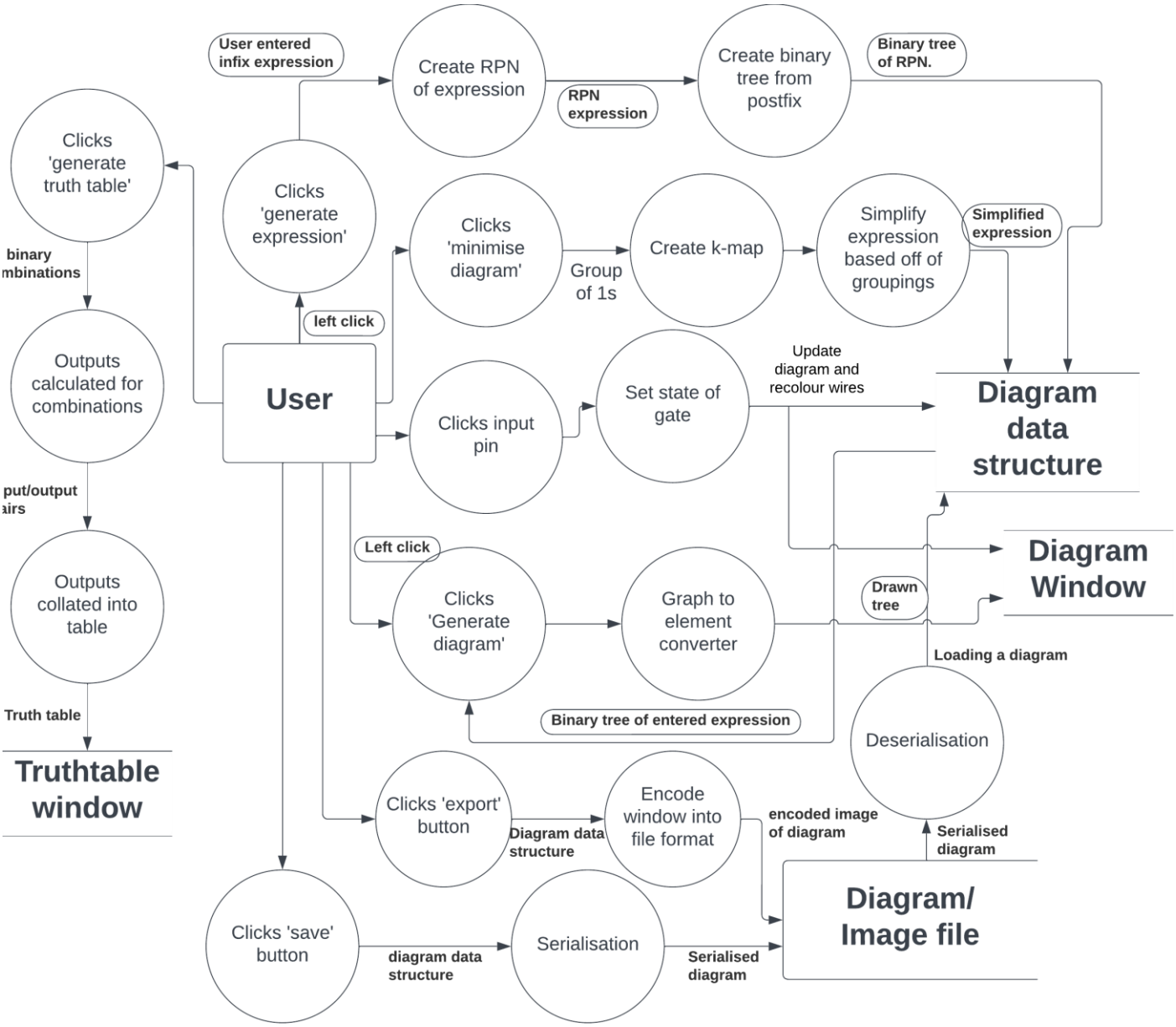


Figure 3: Data-flow diagram of the core of my proposed system.

Objectives

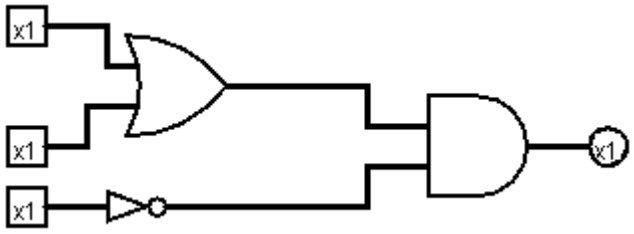
Core Objectives

1. Every feature should be no more than three clicks away.

To ensure that the system is fully utilized by its users, every feature should be no more than three clicks away from the main diagram window. Three clicks is my cut-off point for this objective's success as it will allow enough space within the system's menus for an extensive range of functions whilst also being quick to navigate.

2. A new user should be able to create a diagram and its respective truth table from a Boolean expression in less than 2 minutes.

A good test to see the usability of my system is to time how long it takes a new user to construct a simple diagram and its truth table from a given Boolean expression. An example is shown below. This is a common task in lessons and is fundamental to my project. 2 minutes is enough time for a user to learn the system if it is well designed, as items and buttons should be clearly laid out and labelled for the user. Hence, this is my cut-off time for the success of this objective.



a	b	c	x
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

3. The program will not lag on a reasonably powerful computer.

Due to the program being used within a school setting, it must be well optimised for their computers (which are reasonably powerful) to reduce lag. If the program lags, then this will make it difficult to use for both students and teachers which disrupts lessons and revision. An appropriate bound for performance would be ≈60fps. This is because if the program performs at this framerate, then it can be assumed that the program will run well on a majority of hardware and will not cause lag.

4. The program will not error regardless of Input.

To ensure that the program does not fail whilst it is being used, it will have extensive validation and exception handling. This will ensure that erroneous inputs such as calculating truth tables with erroneous expressions cannot cause the program to crash. Alongside preventing crashes using exception handling, the program will also have validation to notify the user on the nature of the error that has occurred. This will minimise the chances of the program breaking due to the user making an error. To measure the success of this metric, rigorous testing will be conducted to find any flaws within the program's code. Testing will also be done by the users to ensure the largest range of expressions are being tested, which are not accounted for within my own testing of the program.

5. The system's output will be versatile for students/teachers.

One of the major use cases of the program is generating logic diagrams and adding them to work (such as a teacher producing a work sheet for their students). This mean that the program should be able to save, load (in a custom file format) and export generating logic diagrams. This will allow students/teachers to come back to their work at a later point and make any necessary changes. To ensure, the program's exported diagrams are compatible with other external software, the program should be capable to export multiple file formats. If it can do this then this objective is deemed to have been satisfied

6. The program should provide help to users if they encounter difficulty using the program.

If a user were to get stuck using the program (such as not remembering the truth table for an AND gate), the program should provide resources and guides on the logic gate, so that they student can use it as a reference for their learning. This objective can be measured using a user survey asking about the usefulness of these resources and whether or not they helped the user's particular issue. Another way this objective will be tested is by using a new user. They will be asked to complete a difficult task, such as drawing a complex logic diagram and minimising it. The only resource they will have access to will be the references within the program. If they cannot complete this task within 5 minutes, then this objective will not be satisfied.

7. The program should cover all elements within the GCSE and A-level specification.

This objective will be tested in two ways. The first way will be with a variety of students from all of the different age groups of students. They will be asked to explore the program and 'mess about' with it. After 3 minutes, they will then be asked if they found the program to be fully featured for their needs. Their responses can be used to evaluate the success of this objective from the users' point of view. The client using the specifications themselves will conduct

a more thorough examination. Their feedback will be used in tandem with the students to evaluate the success of this objective.

8. The program will be written to the current coding conventions from Microsoft.

To ensure that the program will be readable, easy to understand and consistent, a coding style will be used. The program will be written to the current coding conventions that have been created by Microsoft and can be found on their website for C# .NET. The objective will be tested using a simple checklist which can be found within the technical solution of this document. More details of the convention and the reasons behind it can be found in the technical solution.

9. The program should minimise a complex logic gate diagram.

This is a concept that is not covered within the A-level specification but is very useful amongst logic diagram creators. This is something that an able A-level student may have learnt about and is interested in exploring and extend their knowledge of Boolean algebra, and the application of Boolean identities. It would also help reduce the size of created diagrams and present interesting challenges for students. This objective could be tested by timing its speed of execution. If the program takes more than 5 seconds, then this objective is not satisfied.

Extension Objectives

10. The program could have automatic diagram alignment to produce more visually appealing diagrams.

One of the issues with current systems is the lack of help when creating a diagram. This generally causes users to make diagrams that are not always the most visually appealing. If the program could either suggest wire and gate placements or automatically align diagrams, then this would help produce more visually appealing results. This objective can be measured using an arbitrary scoring value. This value would be given by the following formula below:

$$\text{Score} = 2 * ((\text{x}) + (\text{y}) + (\text{z})) + (\text{c})$$

$$\text{Where x} = \text{number of intersections between wires}$$

$$\text{y} = \text{total length of wire}$$

$$\text{z} = \text{number of wire turns}$$

$$\text{c} = \text{number of gates}$$

A diagram with a lower score would be a more visually appealing one. The scores should be compared between the diagram aligned by the program and the original diagram. A mean average of all of the scores would be taken for many different diagrams. If the objective is successful, then the program should produce diagrams with a lower mean score given by the formula above.

11. The program could provide questions to the user about simplification and logic diagrams.

As mentioned previously, logic questions are a useful way of revising logic gates and helping the user to consolidate their knowledge and understanding. The program could generate different types of questions that could test the user's knowledge of logic gates. Some of the types of questions that it could ask are listed below, further question examples are also given:

- Diagram questions
  - The student will be shown an incomplete logic diagram and asked to complete it by dragging and dropping the gate from a side-panel.
  - The student will be given a Boolean expression or truth table and asked to draw the corresponding logic gate diagram from it.
- Truth table questions
  - The student is given an incomplete truth table representing a Boolean expression. They will then be asked to fill in the table.

- The student will be asked to identify a particular gate / Boolean expression from its truth table.

- Boolean algebra questions.
  - The student is given an expression and asked to simplify it. This can be checked using the programs' minimisation functionality.
  - The student is given a simplified expression and asked to find the set of expressions given in a list that simplify the one that they are given.

To test whether this objective has been met, Computer Science students with a range of abilities will be asked to complete 10 questions generated by the program where each question type is being tested. The students will also be asked to complete a very short survey about the questions and whether or not they found the questions helpful. If 90% of the test group say that they found the objective helpful, then this objective will be met.

12. The program could show the algebraic steps when minimising.

When using Petrick's method (when using the Quine-McCluskey algorithm), one of the processes within it is using the distributive law and other Boolean identities to simplify the Boolean expression entered by the user. A-level students must be aware and able to use these laws and identities within questions. Therefore, showing the steps that the program is taking to simplify the expression would be very helpful for students and would ultimately, help their understanding of Boolean algebra as they can see a practical example of the laws being applied.

Data volumes

When the program is running, there is a variety of data that needs be stored to represent logic diagrams and truth tables. The following section considers a reasonable limit for the volume of data that needs to be stored by the computer, whilst the program is running.

Logic diagrams

The table below stores all of the components that would be required to make a logic diagram. The NOT gate has been separated from the other gates as it needs one input whereas, the other gates need two inputs. The size of strings have been calculated in accordance with Unicode, where one character is 16-bits (2 bytes). It should be noted that the objects are going to be stored within lists in the program, however, for the benefits of this exercise, a table is a more effective way of calculating the storage requirements. The ID properties below are used to replicate the indexing used in a list.

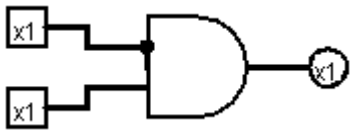
Component	Property	Datatype	Size (bytes)
Input pin	State	Boolean	1
	Label	Char	2
	inputID	Integer(byte)	1
Output pin	State	Boolean	1
	Label	Char	2
	outputID	Integer(byte)	1
AND / OR / NAND / NOR / XOR gate	IsConnectedPoint1	Boolean	1
	isConnectedPoint2	Boolean	1
	input1Wire	Integer	4
	input2Wire	integer	4
	IsConnectedOutput	Boolean	1
	OutputWire	integer	1
	gateID	Integer	4
NOT gate	isConnected	Boolean	1
	InputWire	Integer	4
	isConnectedOutput	Boolean	1

	OutputWire	Integer	4
	gateID	Integer	4
Wire	isConnectedPoint1	Boolean	1
	isConnectedPoint2	Boolean	1
	Point1Gate	Integer	4
	Point2Gate	Integer	4
	ID	Integer	4

Truth tables

Property	**Type**	**Size (bytes)**
Number of Rows	Integer	4
Number of Columns	Integer	4
Input-Output Pairs	Integer[,]	((Number of inputs * 4) + 4) * (2 ^ number of inputs)
Headings	String[]	(Number of inputs * 2) + 2

If we use the diagram below as an example circuit:



Then the minimum storage required for the diagram would be equal to:

$$3 * (1 + 2 + 1) + (1 + 1 + 4 + 4 + 1 + 1 + 4) + 3 * (1 + 1 + 4 + 4 + 4) = 3 * (4) + (16) + 3 * (14)$$
$$= 70 \text{ bytes}$$

If we then generate truth tables for this diagram (shown below) this would be equal to:

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

$$5 + 3 + ((2 * 4) + 4) * (2 \text{ ^ } 2) + (2 * 2) + 2 = 5 + 3 + (12) * 4 + 4 + 2$$
$$= 8 + (48) + (6)$$
$$= 62 \text{ bytes}$$

This means that the minimum required space for a logic diagram and its respective truth table is equal to $70 + 62 = 132$ bytes. This is insignificant amount for modern day computers.

Boolean expressions

Within the program, Boolean expressions will probably be represented as strings. Due to their being infinitely long expressions that can be made, it is impossible to calculate the storage requirements for these. However, if we assume that the longest expression is 32 characters long, which still allows for a huge range of expressions to be inputted, then the storage requirements would be equal to $32 * 2 = 64$ bytes for the expression itself. If this expression was converted to a diagram, more storage will be taken up. An example is shown below:

If the expression is $(\text{A} \,.\, \text{B}) + (\text{C} + \text{D}) \,.\, (\text{A} + (\text{C} \,.\, \text{B})) \,.\, (\text{A} + \text{B}) \,.\, (\text{C} + \text{D})$, this can be made with 4 input pins, 1 output pin, 10 gates and 20 wires $= 460$ bytes, using the table above. This amount is easily handled by computers.

Karnaugh maps

Property	Type	Size (bytes)
Row Heading	String	4
Column Heading	String	4
Row values	Integer[]	4 * (2 ^ (number of inputs / 2) [rounded down])
Column values	Integer[]	4 * (2 ^ (number of inputs / 2) [rounded up])
Table of values	Boolean[,]	Number of row values * number of column values

If we assume that the largest Karnaugh map that will be handled by the program will contain 4 inputs, then the size required to store the map is as follows:

$$(8) + (4 * (2 \text{ ^ } (4 / 2)) + (4 * (2 \text{ ^ } (4 / 2)) + (4 * 4) = (8) + (16) + (16) + (16) = 56 \text{ bytes}$$

This amount can be easily stored in memory by modern computers.

Data sources and destinations

Input	Role
Clicks 'Export' button	Opens a dialog that finds the file path of where an image is being saved. The contents of the diagram are then encoded and saved in the users' selected file format.
Clicks 'Save' button	Finds the users desired file path and saves the contents of the diagram window in a custom file format.
Clicks 'Open' button	User selects file to open and the file is then deserialised and displayed on the canvas.
Clicks 'Clear'	Clears the diagram window.
Clicks input pins	Toggles the input pin on or off, allowing the user to interact with the circuit.
Clicks 'Generate truth table'	Data is taken from the diagram window and a truth table is generated.
Clicks 'Generates Diagram from expression'	The string value of the expression is taken from the input window, and it is converted into the diagram's elements. This is then displayed within the diagram window.
Clicks 'Generate expression from Diagram'	Objects in the diagram window are read and then converted to a string representation. This is then returned to the user.
Clicks 'Minimise Diagram'	The diagram is read from the diagram window and a Karnaugh map is generated. This is used to generate a minimised Boolean expression which is then converted back to a diagram and displayed to the user.
Clicks 'Minimise expression'	The expression is read from the expression window and a Karnaugh map is generated. The map is then used to produce a minimised expression which then displayed to the user.

Output	Description
Image file	An exported image of the logic diagram. This will follow custom user preferences such as optional grid lines.
Diagram file	A custom file format that will allow diagrams to be saved and then edited at a later point.
Truth table	The program will produce a truth table when instructed by the user. This will be an image that can be exported.
Diagram	The program will produce logic diagrams from Boolean expressions that are inputted by the user. They can be produced either in their minimised or non-minimised form.
Wire states	When a user clicks an input pin of the diagram, the wires within the diagram change colour to reflect the updated states within the diagram.
Label dialog	A dialog that comes up when either an input or output pin is added. This will ask the user to select the label from a list of characters.

Edit label	A window that is used to change the name of a particular input/output pin. A change is not expected hence this is not a dialog.
Error message	A window showing the user that an error has occurred within the program. This message will contain details of the error and solution to it.
Expression	Boolean expressions either in their minimised form or not. This will be displayed in a separate window.
Karnaugh map	A variant of the truth table that will be displayed to the user. This will then be used to produce minimised diagrams and expressions.

Research

Online material

Online resources have been very useful in my understanding of how logic gates behave. Some notable ones include:

- https://www.youtube.com/watch?v=QZwneRb-zqA

This video provided a comprehensive walkthrough of logic gates and how they operate it covered many aspects from switches to adders and two's complement. This video was helpful because it improved my understanding of how logic gates work and the nature of the concepts that are being applied within my program. This will allow me to program my project more effectively as I have a deeper understanding of the concepts in use.

- https://isaaccs.org/c/sys_bool_logic_gates?examBoard=all&stage=all

This is a series of references that contain all of the required content within logic gates for the GCSE and A-level specifications. This is useful for me as I can make sure that I fully understand the material, within the program. It also means I can make sure that the entire specifications are properly covered.

Other resources have also been helpful into understanding other techniques that are being used with my program.

- https://www.youtube.com/watch?v=3vkMgTmieZI

This the introduction to a series of videos related to Karnaugh maps, which are used for minimising Boolean expressions. This series was extremely useful due to the number of examples demonstrating key concepts and the summary at the end of the video establishing the rules for grouping.

- https://www.youtube.com/watch?v=l1jgq0R5EwQ

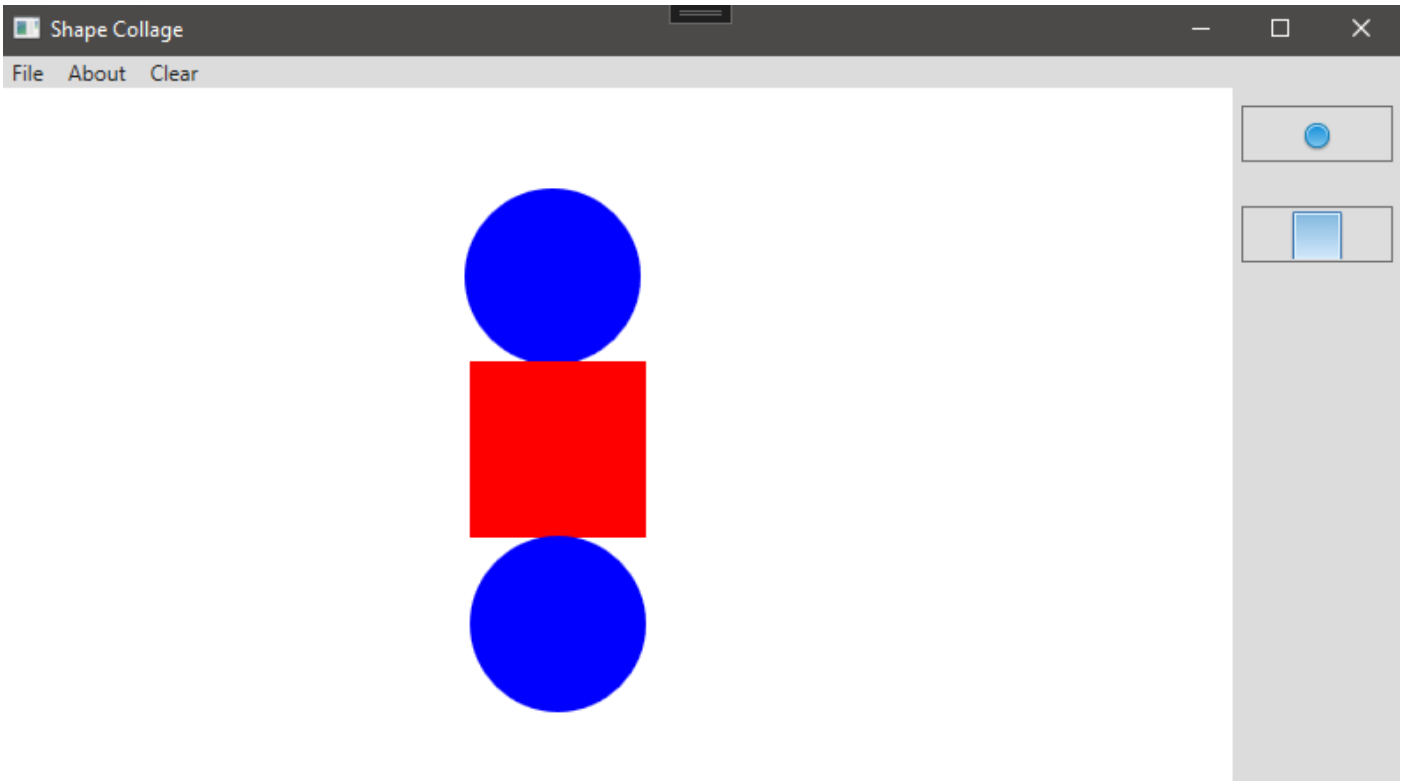
This video shows a demonstration of the Quine-McCluskey algorithm. This is another method that can be used for minimising expressions, that is functionally equivalent to use Karnaugh maps. This video is useful as it clearly shows the process that would need to be carried out by the computer, making the algorithm much simpler to understand in comparison to using Karnaugh maps.

- https://www.youtube.com/watch?v=z9s8A8oBe7g

This video explains the generation of Boolean expressions from diagrams. It contains some useful information such as Boolean operator precedence and also many examples on how to form boolean expressions.

Programming

Shape Collage

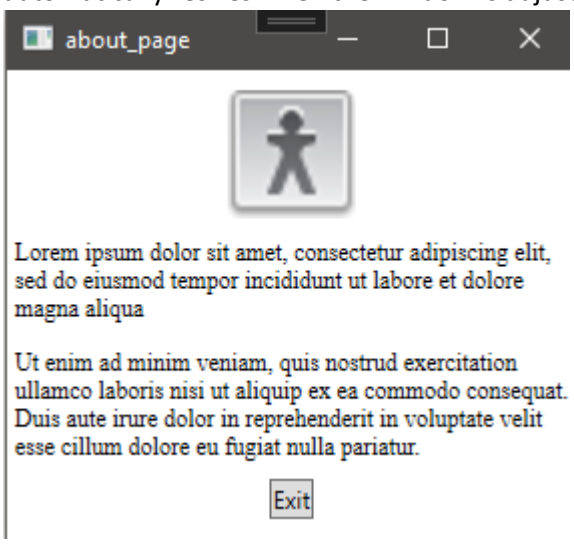


This program allows the user to create simple pictures, using circles and squares. The program allows the user to add shapes to the canvas using the side buttons. They can also reposition the shapes by dragging them around the canvas. The program also allows the user to save, load and export the contents of the canvas using serialization, deserialization and encoding respectively.

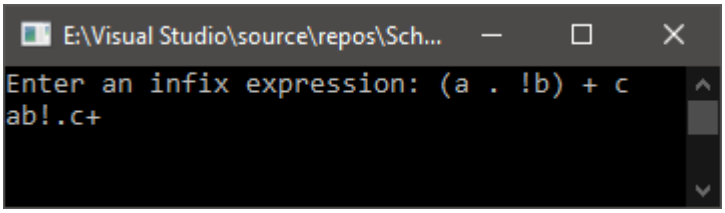
This program was really helpful for my understanding. This is because, aside from it being my first complete project within WPF, it included lots of techniques that would be required for my program. This allowed me to iron out any problems with these common techniques. It also give me insight into how the UI and 'non-logic gate related' portions of my program could look like. Elements of the program that were helpful include:

- The program uses custom userControls to represent the shapes in the canvas. The main benefit of using these is that it is very easy to make complex elements that can be used within the main window. This is because a custom userControl has its own set of XAML. This makes it very easy to reuse code and its XAML to create intricate and detailed interfaces. Due to the code-behind file it is also easy to add functionality to the user control. This was helpful because userControls underpin WPF, this allowed me to work with this and familiarize myself with how to use them.
- Serialisation and Deserialization of the window. This technique was new to me when I started this program. However, it would be essential for saving and loading diagrams. This program would create a custom file format (.ajm) and then serialise the custom userControls within the canvas. This was where the userControls were so powerful because it allowed me to quickly write a class that could store the necessary data and write it to the file. This was helpful because serialisation and deserialization were completely new to me. This project allowed me to explore it and how it fits into a visual program.
- Exporting the contents of the canvas. The program currently exports images of the window (.png). It does this by encoding the canvas as a bitmap and then saving it using a memory stream. This was useful because I have never worked with saving images before this project, and I was unaware on how to do it. This portion of the program allowed me to learn this technique which would be very important in the main project as this is the basis for exporting logic gate diagrams.

- This program also helped me learn how to fully understand and utilise XAML. This is seen in the about page of the project (which can be seen below). This is because it allowed me to experiment with new windows and also some of the features of blocks within WPF. An example of this is the text within the window automatically resizes when the window is adjusted.



Infix to Postfix Boolean expression converter

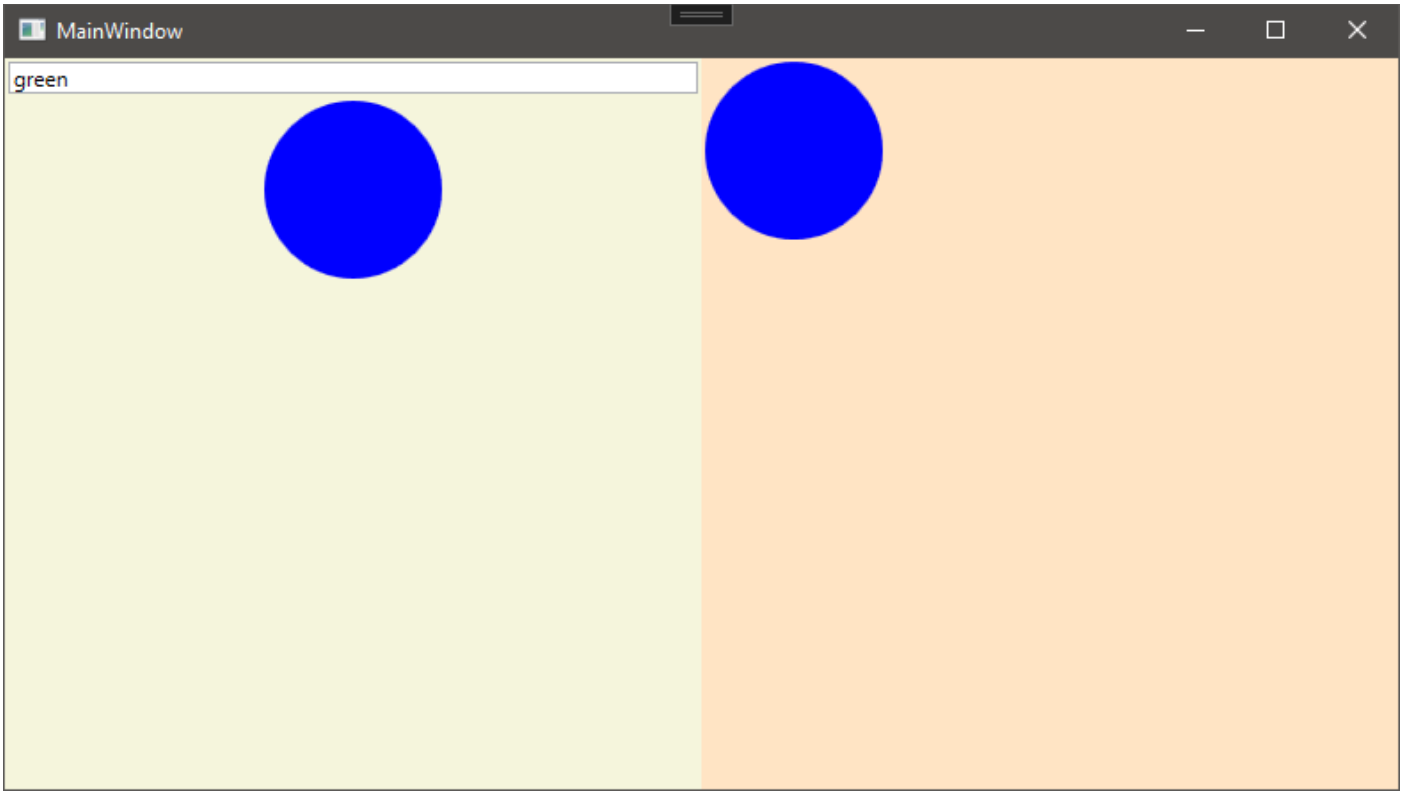


This program does as the title states, it converts an infix Boolean expression to the postfix (RPN) expression. The program makes use of the 'Shunting yard' algorithm to convert the expression. It also uses a linear search to ensure that Boolean operator precedence is followed.

This program was helpful for a few reasons:

- It allowed me to visualize what the expressions will look like. This is helpful for when I design the solution to this project as I am more aware of the nature of the data I am dealing with.
- The program also allowed to make use of a stack. Prior to this, I have never programmed using a stack and had only seen them conceptually. This program allowed me to apply my knowledge and become more familiar with using this data structure.

Data transfer using drag-and-drop



This is a program that implements drag-and-drop for data transfer. This allows the user to either move the circle data or copy it to the other panel. To make this tutorial I have followed a tutorial from Microsoft, which is linked below. This is to become more familiar with the different uses of drag-and-drop within visual programs:

https://learn.microsoft.com/en-us/dotnet/desktop/wpf/advanced/walkthrough-enabling-drag-and-drop-on-a-user-control?view=netframeworkdesktop-4.8

This program was helpful because it allowed me to more to work with drag-and-drop for transferring userControls from one panel to another within WPF instead of just visually moving the userControl. This is extremely useful, because all WPF windows are built upon panels. This will be useful for my program as it will allow me to create a more visually appealing user interface as I can visually represent the components within my program.

This program also allowed me to work with the grid panel within WPF. This is very valuable as the grid is extremely versatile, due to its scaling capabilities, which allows for advanced UI's to be created.

Constraints / limitations

Program limitations

- The program will not contain any content/techniques in logic gates or Boolean algebra that is beyond a reasonable expectation of an A-level student. This content is defined by the A-level specification.
- The diagrams produced by the program will conform to the ANSI/IEEE standard 91-1984.
- When drawing diagrams from expressions there are some cases that will not be drawn by the program:
    - Diagrams with more than 13 unique inputs will not be drawn.
    - Diagrams that contain multiple output pins cannot be drawn as these diagrams cannot be represented with a single Boolean expression.

User limitations

General Limitations

- Users are expected to be able to input a Boolean expression into the system.
- Users should be able to recognise logic symbols (such as an AND gate) without needing the name written.

Students

- It can be assumed that any student using the system will be studying Computer Science or have experience with using a computer.
- It can be assumed that the system will be used under the supervision of a teacher, either within a Computer Science lesson or an after-school club/revision clinic. It can be assumed that the teacher is competent with using the program.
- A student is expected to have complete knowledge of the content within their age group. They are, however, able to use features that are of a higher age group than their own.

Teachers

- Any teacher is assumed to have complete knowledge of the OCR GCSE Computer Science and AQA A-level Computer Science specifications.
- A Teacher is also assumed to have complete proficiency with using a computer.

Proposed solution

Language

Some of the programming languages that I could have used include C# and Python. I have ultimately chosen C# as the language that I will be using for my program. This is because, it is also a compiled language. This means that it runs more quickly in comparison to Python which is an interpreted language. I also feel that because C# is statically typed, it produces more readable and understandable code. A downside of using Python is that, because data-types are not explicitly defined, it is easy for me to write a lot of sloppy and poorly-written code. Using C# will help me to remain accurate by defining the data type of variables. This will reduce unnecessary code which will improve the performance of the program.

I also find the syntax that Python uses to be extremely unhelpful. One of the reasons is the lack of explicit data-types as mentioned previously, but also the statements within Python are more ambiguous. An example of this is the For loop. I find C# to be much clearer and because there are curly bracers, the code is easier to follow. These factors cause code written in C# to be debugged much more easily as it is easier to read. Debugging is also made easier with C# due to it being in Visual Studio. This will make development easier and faster.

Finally, I find C# to have a better implementation of OOP. This is because the syntax is clearer and there is also clearly defined access levels for fields. This allows for clearer code and more versatility when programming. OOP concepts such as inheritance and overriding, are also much easier to do within C# in comparison to Python.

Graphics framework

There are two main frameworks that are available, these are WinForms (Windows Forms) or WPF (Windows Presentation Foundation). I have ultimately decided to use WPF for this program. One of the reasons for this is the introduction of XAML. One of the advantages of this is that XAML is much more customizable which allows me to create intricate custom userControls such as buttons, text blocks etc. This will allow for a better user interface which will create a nicer, more visually appealing program. Another benefit of using WPF is that there are lots of automatics features within it. The most obvious of these is automatic resizing, which is a nice feature (even though it is not significant). One of the biggest reasons I have chosen WPF is the ‘code-behind’ files. These files link the XAML of a userControl to its C# class. This is extremely useful as it allows me to create good visuals but pack lots of function into a ‘container’. This allows for a very reusable code-base which can be interacted with in a variety of different ways. This is superior to WinForms because custom userControls must be made through code exclusively. This creates controls that look worse, and the resulting code is long and difficult to read.

Finally, the last reason as to why I chose WPF is that userControls can be modified and interacted with independently. This allows a user to move objects around the window easily, creating a very smooth user experience. This differs from WinForms which only allows for static pictures. This means that to move an object around the window, the canvas must be updated frame-by-frame, which creates a very clunky experience for the user.

Design

The goal of this is section is to establish the how I am going to implement my proposed system, which has been discussed in the previous section.

Overall system design

Inputs

Input name	Description
Input pin	A visual block that always has a state. Its default state will be zero or 'off' to avoid the pin interacting with other elements within the diagram. A label will be assigned automatically and displayed to the user when the diagram is created. These can then be clicked to dry run the drawn diagram.
Logic diagram	A binary tree representation of a logic diagram. This will have been produced in the diagram creation window from an expression or through a loaded expression in a saved file. It will be used for generating truth tables and minimising diagrams.
Boolean Expression	A string representation of a Boolean expression. It will be used to generate truth tables, minimisation, and the creation of diagrams by converting it into postfix. The user will type these into text boxes within the program.
Generate Button	A small button within the truth table window that when clicked will generate a truth table from either a diagram or entered Boolean expression. If both are present within the program, the drawn diagram will be prioritised. A similar button will also be used to generate diagrams from expressions.

Outputs

Output name	Description
Diagram file	An XML file. This will allow the diagram to be recreated and then edited within the program.
Image file	A file that is exported by the program, (".png"). The image will contain the visual contents of the diagram creation window and will not allow diagrams to be reproduced or edited.
Input pin states	This is a visual representation within the diagram creation window. When a user interacts with an input pin, its state will be marked by a colour change and its value will be shown. This will show that the pin is currently supplying a one into the diagram.
Output pin states	This is a visual representation within the diagram creation window. The output pin will automatically update its state to match the connected element. This will be shown by its colour and label. This update will occur after any change in the diagram window to create a responsive system.
Wire states	This is a visual representation within the diagram creation window. When a wire is placed onto the diagram creation window it will initially read the states of the connected elements and match them. If an element is not connected to the wire, the wire will have a state of zero.
Status box	A small textbox in the corner of the program that will provide information on the current action, such as "Click on the window to connect a wire". It will also provide information on the most recent action, such as "AND gate placed".
Truth table	A visual representation of the input/output pairs of a given Boolean expression or logic diagram. The table will be structured with the intermediary steps of the diagram/expression. The table will be shown to the user within a sub-window.
Input/output Pairs	A 2D array that maps all of a diagrams/expressions' inputs to their outputs. This will not be shown to the user but will form the data within the truth table. Each pair will also include intermediary steps with extra indexes assigned to them.
Logic diagram	A visual representation of the Boolean expression that has been entered. This will be stored in the logic diagram structure and displayed in the diagram creation window.
Boolean expression	A string representation of a Boolean expression. This will be of the diagram drawn within the diagram creation window.

Postfix expression	The postfix representation of the Boolean expression entered by the user. This conversion will take place as the postfix can be used for generating the headers for a truth table and evaluation.
Minimised diagram	A data structure that stores the logic diagram representation of the minimised Boolean expression produced from minimisation. This is shown to the user within the diagram creation window.
Minimised expression	A Boolean expression in its simplest form. This expression is formed from the groups identified within the Karnaugh Map. It is also found using the essential prime implicants from the Quine-McCluskey algorithm. It will be displayed within a textbox.
Karnaugh map	An alternate form of the truth table. It can be used to identify groups of bits which then can be used to create a minimised Boolean expression. The groups will be drawn and shown on the map. This map will be displayed to the user.
Groupings	Within a Karnaugh map groups of ones are displayed to the user. These are also used to identify the minimised form of the inputted Boolean expression.

Processes

Process name	Description
Deleting elements	When the user holds right-click for 2 seconds the element will be removed from the diagram creation window. The diagram data structure will be updated to reflect the change that has been made.
Changing Input pins' state	When the user right-clicks on an input pin, its state will change. This will be visually shown by the pin changing colour.
Element states	When a user changes the states of an input pin within the logic diagram, the states of all other components will be updated. The updated states will be shown visually by a change of colour.
File save	The contents of the diagram data structure (elements within the diagram creation window) will be serialized into XML. This can be written to a file which allows diagrams to be saved.
File load	The contents of an XML file will be deserialized. The deserialized elements will be loaded into the diagram creation window.
Image save	The contents of the diagram creation window will be encoded and saved. An exported file cannot be used to recreate a logic diagram within the program.
Generating truth table	The user will click a 'generate truth table' button within a sub-window. The program will calculate the input/output pairs. It will then visually show these pairs to the user within the sub-window.
Calculating input/output pairs	The program will substitute all possible combinations of inputs into a Boolean expression. If a diagram is entered, then a Boolean expression will be generated. The Input/Output pairs will be stored in a 2D array for easy access.
Inputting Boolean expressions	When the user has entered a Boolean expression, the program will convert this into postfix, which will allow the intermediary stages to be defined and the input/output pairs to be calculated.
Defining headers/rows Generate K-map	The program will define all possible input combinations given the number of inputs within the diagram. These will be displayed as the headers and rows of the map, with the labels also displayed to the user.
Populating map Identify groups	The Karnaugh map will then be populated by substituting the inputs into the Boolean expression and then evaluating the expression. This will also be displayed to the user. The program will then identify the groups of ones within the Karnaugh map. These groups will then be used to form the minimised Boolean expression.
Identifying groups	The program will then identify the groups of ones within the map based off of a set of pre-defined rules within Karnaugh maps. This will not be displayed to the user.

Inspecting groups	The program will then use the groups within the maps and find corresponding subexpressions which can then be concatenated to form the completed minimised Boolean expression.
Finding the prime implicants	The program will merge the minterms of the entered boolean expression. These will then be merged to form the prime implicants. These can be converted to find the minimised expression.
Expression to diagram conversion	The program will, using the expression, define the inputs and add input pins. A postfix traversal will be used to draw the diagram in the correct order. An output pin will also add to give the diagram an output. Wires will be connected automatically when the conversion is being done.
Diagram to expression conversion	The program will carry out an in-order traversal on the diagram which will produce the Boolean expression of the respective diagram.

Storage

Storage item	Description
Logic diagram	A data structure that represents a logic gate diagram. This will be based off of a binary tree. The data structure will contain information of the gates/input and output pins (nodes) and wires (edges). The states of each component will also be stored. It will also be displayed within the diagram creation window.
Diagram File	An XML file. This will allow the diagram to be recreated and then edited within the program. The file will be saved in the users' file directory.
Image file	A file that is exported by the program, (".png"). The image will contain the visual contents of the diagram creation window and will not allow diagrams to be reproduced or edited. The file will be saved in the users' 'pictures' directory.
Boolean expression	The string representation of a Boolean expression. It will be stored within the textboxes in the interface and also the logic diagram class. This it can be manipulated and used for generation and minimisation.
Truth table	A 2d array storing the headings of the truth table and the input/output pairs, which occupy the body of the table.
Karnaugh map	The map that represents the logic diagram drawn by the users. This will show all of the inputs and their respective outputs, whilst being clearly labelled. The groups will also be drawn onto the map and displayed, so the user can clearly see the groups that have been identified by the program.
Groups	The program needs to keep track of all of the groups of ones that have currently been made within the map. This is so they can be used to form the minimised Boolean expression.
Users' age group	This will be stored so the program can correctly identify which features should be marked if a students is accessing areas that are above their age group.

Modular structure of system

System Structure

The following hierarchy charts some of the core parts of my program being, truth tables and file handling.

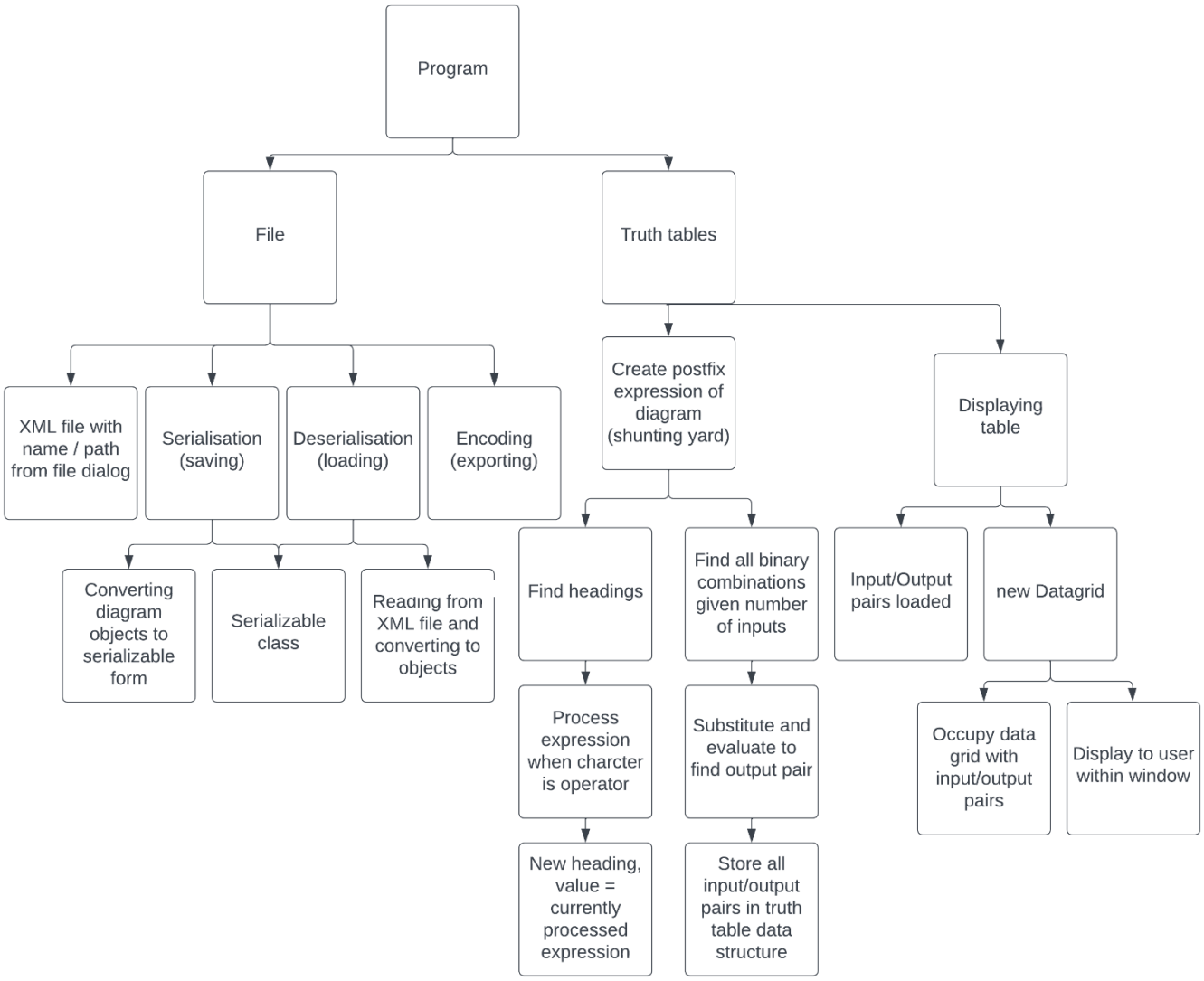


Figure 4: A hierarchy chart diagramming truth table creation and file handling.

The chart below shows the other core functions within the program, these being drawing diagrams from truth tables and diagram minimisation.

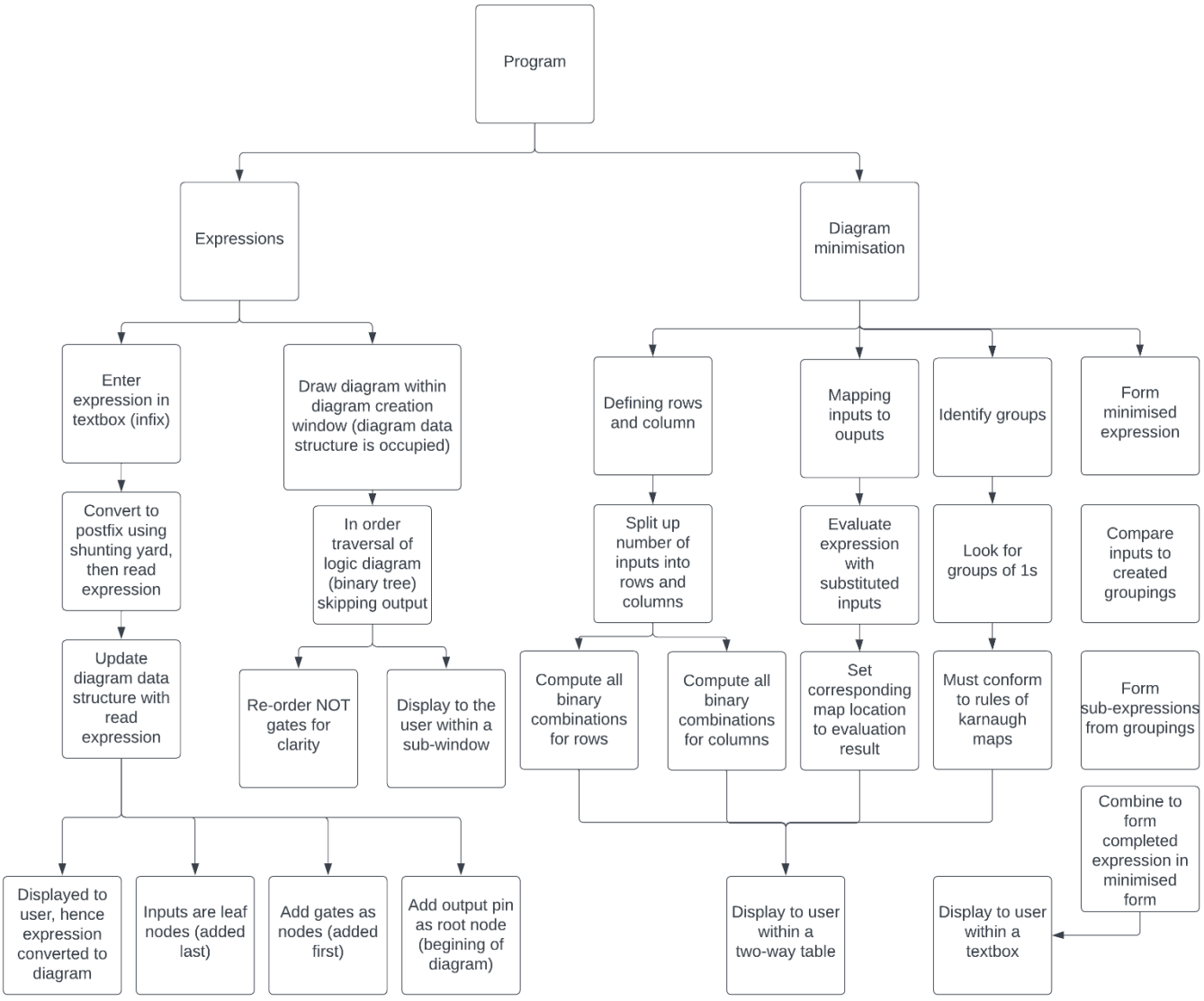


Figure 5: Hierarchy chart diagramming Boolean expressions and diagram minimisation.

Menu structure

Due to most of the functionality of the program being with buttons and textboxes within sub-windows, there is no need for a complex menu structure. This simplifies the program for the user as they do not need to search through a large menu hierarchy which helps them make full use of the functionality within the program.

- File
  - Save diagram.
  - Load diagram.
  - Export diagram.
- Truth table
- Expressions
- Diagram Minimisation
- Help
- About

File structure

When using the program, it can be run using an executable (.exe) file. This file is not significant in size and so it can be stored and run by a modestly powered computer easily.

There are two different file types that the program will interact with/produce. These being:

- Diagram files (.2b)
- Image files (.png)

There is no need for a specific location for these files when they are being saved/loaded or exported. This is because the location will be found using a dialog box.

Diagram files

As logic diagrams, truth tables and minimised diagram/expressions are all produced from the original Boolean expression entered by the user. There is no need for a complex file to store this data and so the original user-entered Boolean expression can simply be written to a text file when the user wants to save a diagram.

Exported images

These files will store the contents of the diagram creation window. This will be stored as a bitmap with a transparent background to help the user to integrate logic diagrams within produced work. The image files will be produced by encoding the window. An example image is shown below:

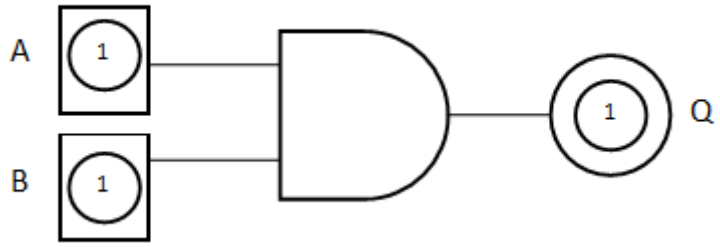


Figure 6: Example image file produced by the program.

Class diagrams

Overview

In my mind, a logic diagram is represented by a binary tree. A binary tree consists of nodes (elements) and edges (wires). The nodes within the logic diagram can be input pin, output pins and the logic gates themselves. Logically, this can be represented by one class and wires can be represented with another.

It should be noted that truth tables and Boolean expressions are all derived from a logic diagram and its output, which means that there is no need for separate classes to store them.

Below is an overall UML to indicate the different objects and relationships within the program.

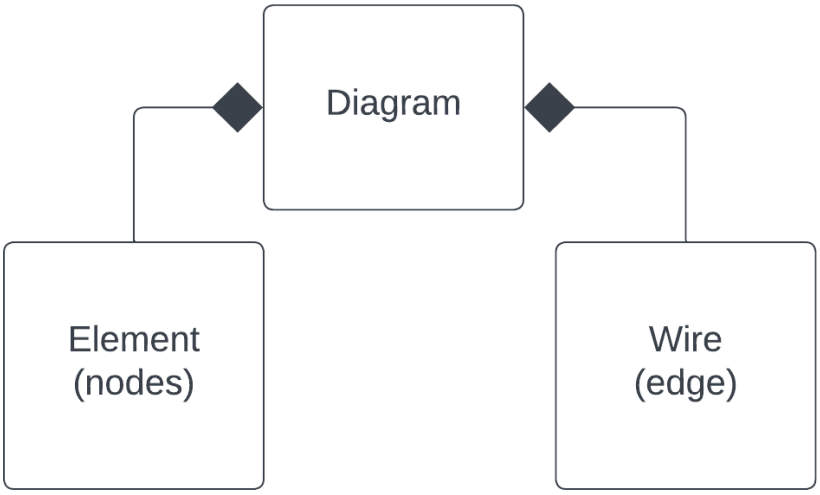


Figure 7: A UML diagram showing the overall structure of the program.

Diagram class

The purpose of this class is to store and manipulate the logic diagram being drawn by the user. I defined previously that this is represented by a binary tree. This means that this class has to store the nodes within the tree and the relationship between each node. As mentioned previously, truth tables, Boolean expressions and Karnaugh maps are derived from logic diagrams. This means that the diagram class should also include methods and attributes that enable these to be represented or processed.

Diagram
- nodes:List<Element> - edges:List<Wire> - inputOutputPairs:int[] - headings:List<String> - rows:List<int> - expression:String - postfixExpression:String - KmapRows:List<String> - KmapHeadings:List<String> - Kmap:int[] - KmapGroups:int[,] + miniExpression:String
+ addElement(obj: Element):void + removeElement(obj: Element):void + removeWire(obj: Wire):void + reIndex(a: int, b: int):void + updateTree():void + DrawTruthtable():void - generateInputBin(numInputs:int): List<int> - generateRowLines(): void - generateHeadLines(): void - generateInputOutputPairs():void + evaulate(expression:String) + getExpression():String + setExpression(exp: String):void + convertToPostfix():void + getPostfixExpression():String + DrawKmap(): void + DrawGroups():void + getMiniExpression():String - generateKrows - generateKheader - fillKmap():void - FindGroups():int[,] - ValidateGroups():int[,]

Element class (nodes)

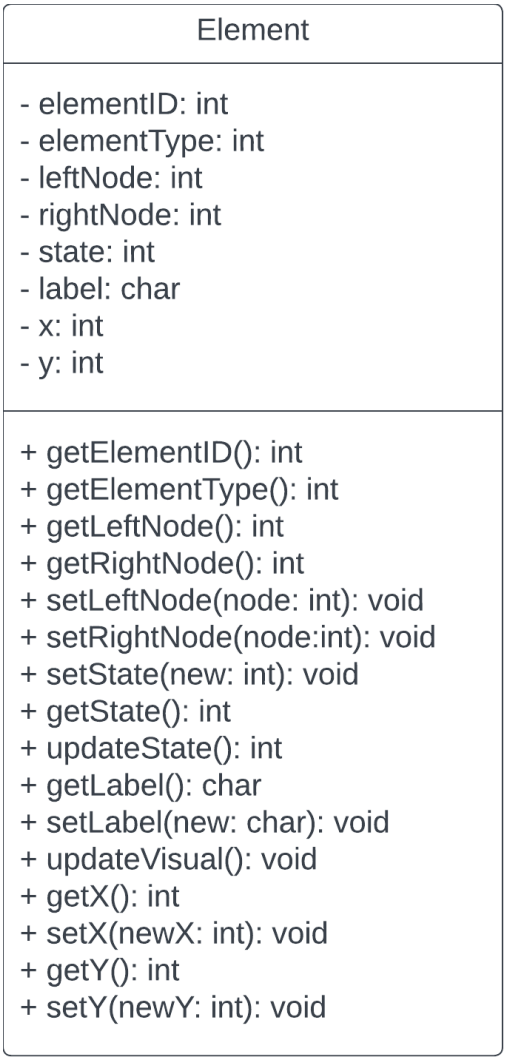
Within the program, an element is any item that goes into making a logic diagram, being input/output pins and logic gates. As stated previously a logic diagram can be represented by a binary tree where the nodes are the elements. Therefore, the purpose of this class is to store the data required for the nodes within the binary tree.

The element class can also store all of the information required for input pins, output pins and logic gates. Separate child classes for each element are not needed as each element behaves in a very similar manner.

In the class each element is differentiated by the element type which is set when the node is added to the tree. This ID will be used within the class's methods to define the objects behaviour. The table below shows the ID and its corresponding element.

ID	Element
0	AND
1	OR
2	NOT
3	XOR
4	NAND
5	NOR
6	Input pin
7	Output pin

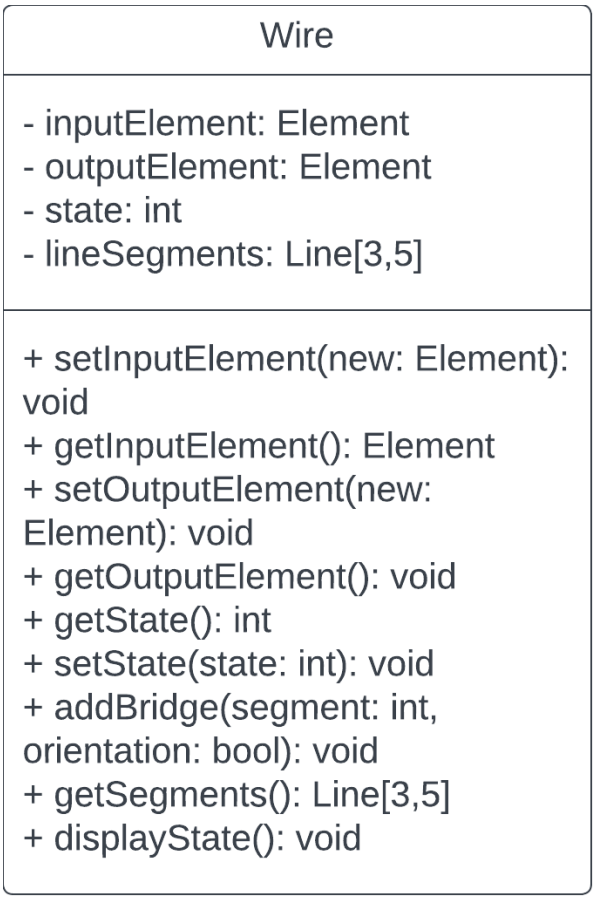
A label will not be assigned logic gates as this would become unfeasible very quickly because the labels would obstruct the view of other elements and it would also be unnecessary to generate lots of meaningful labels for each logic gate within a diagram.



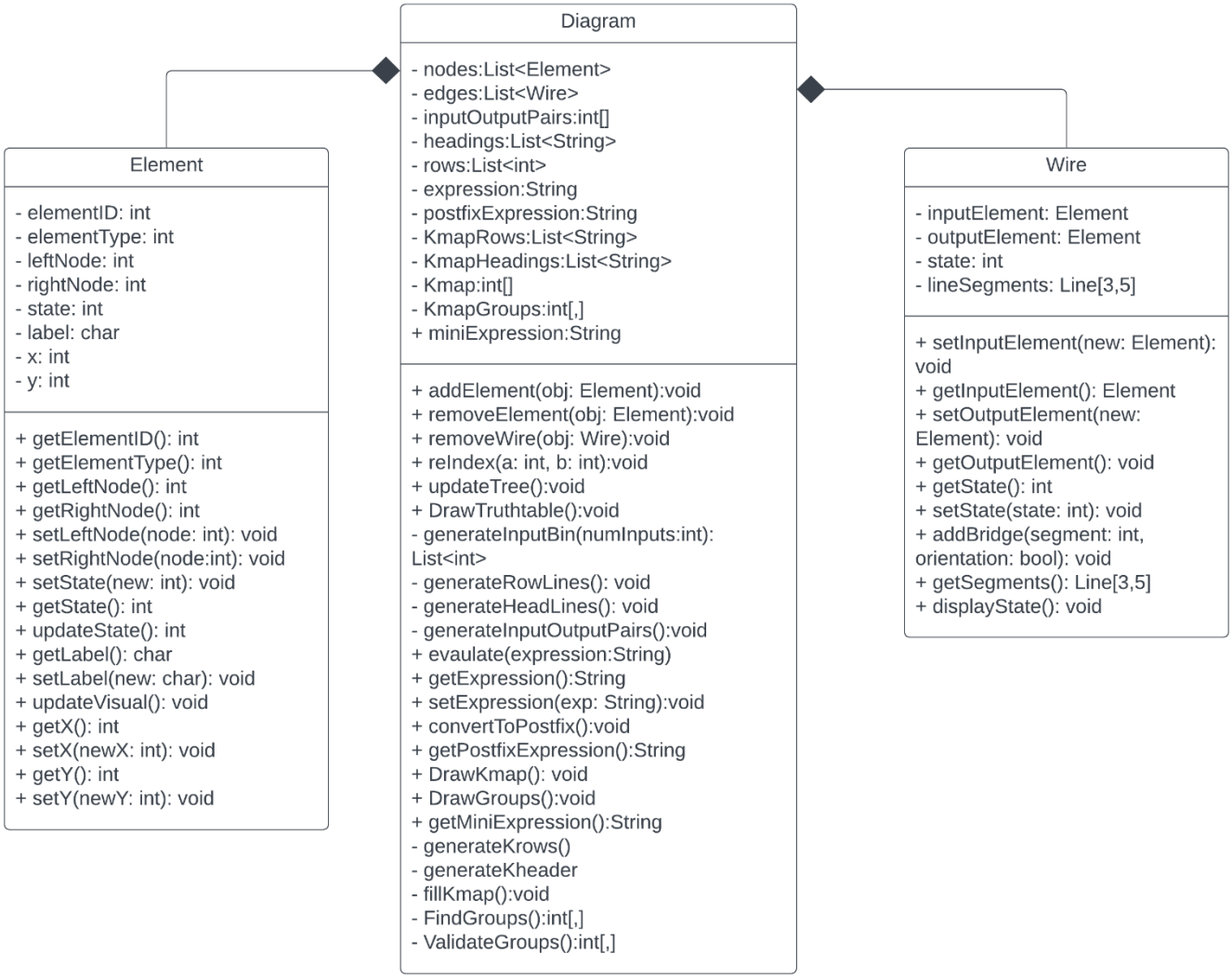
Wire class (edges)

The wire class is used to represent the edges within the binary tree. They are also used to add nodes to the binary tree because if a node (element) is not connected then it has no bearing on the final tree. Wires also mark the point for deleting nodes for the same reason. A wire does not have a label for the same reason as a logic gate and its state is indicated by its colour.

Visually, a wire is made up of three main line segments, 1 vertical and 2 horizontal. It also possible for two wires to intersect each other meaning that the wire object must have capacity for all of the lines. This is given by the 2D array.



Overall UML diagram



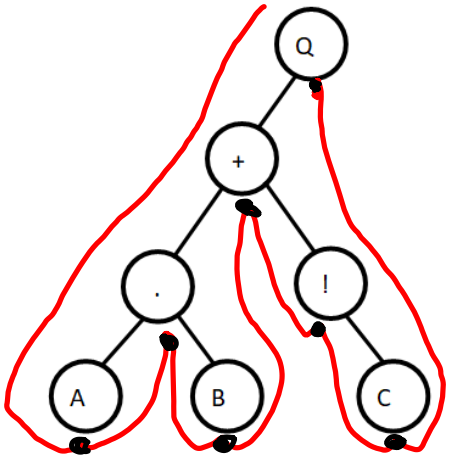
Algorithms

The following sections will document some of the key algorithms that are within the program. These have been written with OCR reference language. Data types that are used within the algorithms but are not within OCR reference language (such as stacks) are following my own notation.

Diagram drawing

In order traversal of a binary tree

As a logic gate diagram is a binary tree, we can make use of an in-order traversal to produce the infix Boolean expression of a drawn diagram. A graphical representation of this process is shown below:



Which produces the expression A.B+!C The written steps of this algorithm are as follows:

- Traverse the left child first until you reach a leaf node. This then outputted as the start of the expression.
- Backtrack to its parent and output the node.
- Traverse to the right child if it is present and output.
- Back track twice to two levels higher in the tree and repeat the process.

This written in pseudocode becomes:

```
Procedure inOrderTraversal(startNode)
      infixBooleanExpression = “”
      toBeAdded = a
      If nodes[a].LeftNode != 0 THEN
            If nodes[a].getElementType() == 6 THEN
                  toBeAdded = nodes[a].getLabel()
            Else if nodes[a].getElementType() < 6 THEN
                  toBeAdded = nodes[a].getSymbol()
            END IF
            inOrderTraversal(startNode)
      END IF

      infixBooleanExpression = infixBooleanExpression + toBeAdded

      if nodes[a].RightNode != 0 THEN
            If nodes[a].getElementType() == 6 THEN
                  toBeAdded = nodes[a].getLabel()
            Else if nodes[a].getElementType() < 6 THEN
                  toBeAdded = nodes[a].getSymbol()
            END IF
            inOrderTraversal(startNode)
      END IF
      logicDiagram.SetBooleanExpression(infixBooleanExpression)
END PROCEDURE
```

Finding the height of a binary tree

Finding the height of a binary tree can be used to make sure that when drawing the logic diagram from a Boolean expression, adequate spacing is given to fit all of the elements within the diagram creation window. Using the pseudocode below (from Baeldung), the following algorithm calculates the height of the tree.

Algorithm 1: Algorithm for calculating the height of a binary tree

```
Data: root: root node of the binary tree
Result: Height of the binary tree
Procedure BTHeight(root)
    if root == NULL then
        return 0;
    end
    leftTHeight = BTHeight(root → left);
    rightTHeight = BTHeight(root → right);
    returnMax(leftTHeight, rightTHeight) + 1;
```

```
Function calculateHeightOfTree(rootNode)
      If rootNode == null THEN
            Return 0;
      END IF
      LeftChildHeight = calculateHeightOfTree(rootNode.Left)
      RightChildHeight = calculateHeightOfTree(rootNode.Right)

      if LeftChildHeight > RightChildHeight THEN
            return LeftChildHeight + 1
      Else
            Return RightChildHeight + 1
      END IF
END FUNCTION
```

Creating a binary tree from a postfix expression.

When drawing diagrams, the underlying data structure is a binary tree. Within the program, the binary tree will be created from the Boolean expression. The algorithm below (from Baeldung) details the process of converting a postfix arithmetic expression into a binary tree.

Algorithm 1: Converting a Postfix Expression With Binary Operators Into a Tree

Data: $x = [x_1, x_2, \ldots, x_n]$: an array of symbols (constant values, variables, operators) representing an postfix expression.
Result: The expression tree of x.
$nodes \leftarrow$ initialize an empty stack.
for $i \leftarrow 1, 2, \ldots, n$ **do**
  if x_i *is not an operator* **then**
    $nodes.push(x_i)$
  else
    $right \leftarrow nodes.pop()$
    $left \leftarrow nodes.pop()$
    $new\_node \leftarrow Node(left, right, operator = x_i)$
    $nodes.push(new\_node)$
  end
end
$root \leftarrow nodes.pop()$
return $root$

Below is an adaptation of the algorithm above, which instead creates a binary tree for Boolean expressions.

```
Function convertExpressionIntoBinaryTree(expression)
      postfixExpression = convertInfixToPostfix(Expression, False)
      Array expressionArray = [postfixExpression.Length]
      Array operators = [“.”, “+”, “!”, “^”, “@”, “#”]
      Nodes = Stack()
      //converting the postfix into an array
      For I = 0 to postfixExpression.Length - 1
            expressionArray[i] = postfixExpression[i]
      NEXT I
      //converting the array into a binary tree using the algorithm above
      For I = 0 to expressionArray – 1
            If NOT(operators.Contains(expressionArray[i]) THEN
                  Nodes.push(expressionArray[i])
            Else
                  Left = nodes.pop()
                  //checking for NOT gate
                  If expressionArray[i] == “!” THEN
                        //not gate
                        newNode = new Element(left, null, expressionArray[i])
                  Else if
                        Right = nodes.Pop()
                        //creating a new node that is being added to the tree
                        newNode = new Element(left, right, expressionArray[i])
                  END IF
                  nodes.push(newNode)
            END IF
      NEXT I
```

```
      RootNode = nodes.Pop()
      Nodes = RootNode
END FUNCTION
```

Drawing the logic diagram from a binary tree

Drawing a logic diagram is done using the binary tree. This tree can be created using the algorithm described above. To draw a logic diagram a breadth-first traversal of the tree can be carried out and the nodes can be drawn when they are traversed by the algorithm. The pseudocode below (taken from the Wikipedia page on the algorithm) shows how to conduct the traversal on a graph.

```
 1  procedure BFS(G, root) is
 2      let Q be a queue
 3      label root as explored
 4      Q.enqueue(root)
 5      while Q is not empty do
 6          v := Q.dequeue()
 7          if v is the goal then
 8              return v
 9          for all edges from v to w in G.adjacentEdges(v) do
10              if w is not labeled as explored then
11                  label w as explored
12                  w.parent := v
13                  Q.enqueue(w)
```

The modification that must be made to this algorithm is that the nodes must be drawn when they are visited and there is no need to label the nodes when they are 'explored' as each node is only visited once.

Finding the positions of nodes

In my view, there is a mathematical relationship for calculating the position of a node within the binary tree. This gives rise to a set of formulae to be able to calculate the X and Y position of the node.

To calculate the X position of the node on the canvas, this is simply the depth within the tree (the column in the logic gate diagram) multiplied by the value taken in the grid squares.

To calculate the Y position of the node on the canvas, this is very similar to the X position, as it is simply a multiple of the spacing between each node within each level of depth within the tree. The spacing of the nodes within the binary tree is given by a decreasing geometric sequence. This is given below:

$\frac{2^h}{2^d}$. Where $\mathrm{h} = \text{height of the tree}$ and $\mathrm{d} = \text{depth within the tree}$.

To convert the grid square location into a position on the canvas, you can simply multiply by the number of pixels for each grid square and the position within this layer of the tree. This can all be put together to get the following functions.

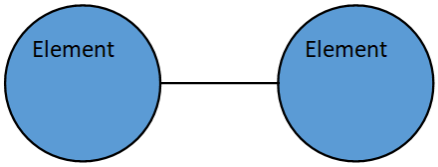
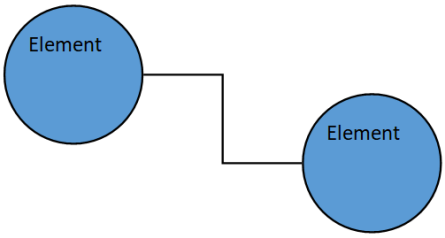
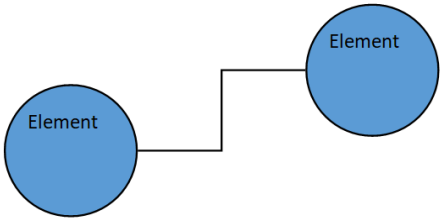
```
Function calculateXposition()
      Return canvasWidth - (pixelsPerSquare * elementWidth + pixelsPerSquare *
xOffset) * depthWithinTree

Function calculateYposition(heightOfTree, depthWithinTree, positionWithinLayer)
      //finding grid reference and converting to position on canvas.
      nodeSpacing = (2**heightOfTree / 2**depthWithinTree) * pixelsPerSquare
      //Position within layer initially 0, so node spacing needs to be added.
      return nodeSpacing + (nodeSpacing * positionWithinLayer)
```

These functions can be used within a breath first traversal, mentioned previously, of the tree to place all of the nodes onto the canvas. This creates the algorithm shown below:

```
Function drawDiagram(expression)
	convertExpressionIntoBinaryTree(expression)
	//Create a queue for the BFS
	q = Queue()
	q.Enqueue(rootNode)
	depthWithinTree = 0
	positionWithinLayer = 0
	sizeOfQ = 0
	WHILE q.Count != 0
		sizeOfQ = q.Count
		WHILE sizeOfQ != 0
			currentNode = q.peek();
			x = calculateXposition(
			y = calculateYposition(heightOfTree, depthWithinTree,
positionWithinLayer)
			drawNode(currentNode, x, y)
			q.Dequeue()
			positionWithinLayer = positionWithinLayer + 1
			if currentNode.leftChild != null
				q.Enqueue(currentNode.leftChild)
			END IF

			If currentNode.rightChild != null
				q.Enqueue(currentNode.rightChild)
			END IF
			sizeOfQ = sizeOfQ – 1
		END WHILE
		depthWithinTree = depthWithinTree + 1
		positionWithinLayer = 0
	END WHILE
END FUNCTION
```

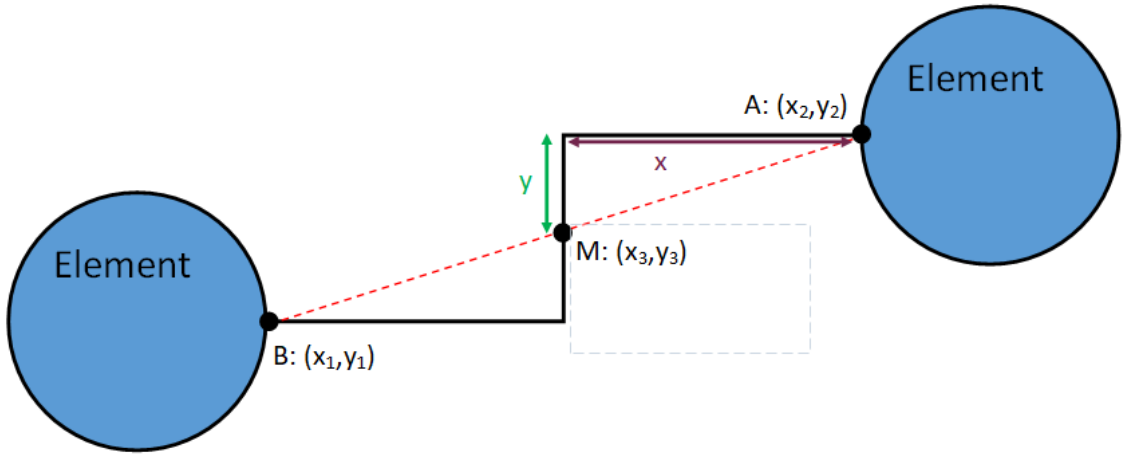
Drawing wires

There are three different ways a wire can look within a logic gate diagram. These are shown below with two example elements.

Case 1 (Elements are level)	Case 2 (elements are on negative gradient)	Case 3 (elements are on a postive gradient)
		

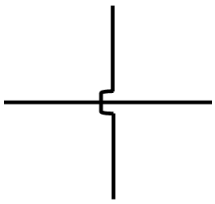
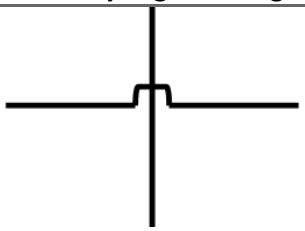
Within the program, wires have two important roles. These are to show the connection of elements in the diagram creation window and to also add elements to the binary tree when they are connected within the diagram creation

window. Using geometry, we can calculate the shape of the wire that has to be drawn, a diagram of this is shown below:



In the diagram above, the two horizontal lines come out of the elements until they meet at the vertical line which is at the midpoint of x_1 and x_2. The points A and B are locations which are offsets from the respective logic gates they are 'attached ' to.

One consideration that is not considered in the algorithm above is if a wire that is being drawn intersects with a wire already drawn in the diagram creation window. If a wire segment intersects another then depending on the orientation of the intersection, a bridge should be drawn. These are shown below:

Vertically aligned bridge	Horizontally aligned bridge
	

When drawing a wire, all other wires must be checked to ensure that no intersections occur. If they do a bridge should be drawn as shown above. To decide whether and where to draw the intersection, geometry can be used. An intersection should not be drawn if the gradients of the two lines are the same, as they are parallel lines and hence, never intersect. A bridge will also not need to be drawn if the intersection lies between the two pairs of X coordinates.

The functions below return whether an intersection is present on a wire segment and the coordinates of that intersection respectively.

```
Function findWireIntersection(wire1, wire2)
      Line1grad = (wire1.getY2() – wire1.getY1()) / (wire1.getX2() – wire1.getX1())
      Line2grad = (wire2.getY2() – wire2.getY1()) / (wire2.getX2() – wire2.getX1())
      If line1grad == line2grad Then
            Return False
      Else
            Line1C = Wire1.getY1() – (line1grad * Wire1.getX1())
            Line2C = Wire2.getY1() – (line2grad * Wire2.getX1())
            Cdiff = Line2C – Line1C
            If wire1.getX1() < cdiff OR cdiff < wire1.getX2() THEN
```

```
                  Return True
            Else
                  Return False
      END IF
END FUNCTION

Function findPointOfIntersection(wire1, wire2)
      Line1m = (wire1.getY2() – wire1.getY1()) / (wire1.getX2() – wire1.getX1())
      Line2m = (wire2.getY2() – wire2.getY1()) / (wire2.getX2() – wire2.getX1())
      Line1C = Wire1.getY1() – (line1m * Wire1.getX1())
      Line2C = Wire2.getY1() – (line2m * Wire2.getX1())
      Xintersection = (line1C – line2C) / (line2m – line1m)
      Yintersection = ((line1C * line1m) – (line2C * line2M)) / (line2m – line1)
      return Xintersection, Yintersection
END FUNCTION
```

The following function draws the two different possible orientations of a bridge, shown above. It does this by using the line section being drawn and redraws it using an arbitrary offset from the point of intersection. This algorithm is below:

```
PROCEDURE drawWireSectionBridge(wireSection, pointOfIntersection, horizontal)
      startPoint = wireSection.getX1()
      endPoint = wireSection.getX2()
      If horizontal == false THEN
            //drawing bottom vertical line
            DrawLine(startPoint, wireSection.getY1(), startPoint,
pointOfIntersection.Y – 10)
            //Drawing top vertical line
            DrawLine(startPoint, wireSection.getY2(), startPoint,
pointOfIntersection.Y + 10)
            //Drawing the bridge itself.
            //Drawing the bottom horizontal line
            DrawLine(startpoint, pointOfIntersection.Y – 10, startpoint – 10,
pointOfIntersection.Y - 10)
            //Drawing the top horizontal line
            DrawLine(startpoint, pointOfIntersection + 10, startpoint – 10,
pointOfIntersection + 10)
            //Drawing the vertical line connecting bridge
            DrawLine(startpoint – 10, pointofIntersection.Y – 10, startPoint – 10,
PointOfIntersection.Y + 10)
      Else
            DrawLine(startPoint, wireSection.getY1(), pointOfIntersection.Y – 10,
wireSection.getY1())
            DrawLine(endPoint, wireSection.getY2(), pointOfIntersection.Y + 10,
wireSection.getY2())
            //Drawing the bridge
            //two vertical lines
            DrawLine(pointOfIntersection.X - 10, wireSection.getY2(),
pointOfIntersection – 10, wireSection.getY2() + 10)
            DrawLine(pointOfIntersection.X + 10, wireSection.getY2(),
pointOfIntersection.X + 10, wireSection.getY2() + 10
            //drawing the bridge line
```

```
            DrawLine(pointOfIntersection.X – 10, wireSection.getY2() + 10,
pointOfintersection.X + 10, wireSection.getY2() + 10)
      END IF
END PROCEDURE
```

The orientation of a bridge is dependent on the line section that is being intersected. The wire segments are listed with an array. It has the structure of the first two items are the horizontal lines and the last item is the vertical line. The orientation is found with the function below:

```
Function determineBridgeOrientation(positionInArray)
      If positionInArray <= 1 THEN
            Return False
      Else
            Return True
      END IF
END FUNCTION
```

The two functions above can be integrated into the drawing function described before to create a new function that can also draw the intersections of the lines if necessary. The complete function for drawing wires is shown below:

```
Function drawWire(inputPoint1, inputPoint2)
      //an array of lines that make up the wire
      Array lines = [null, null, null]
      midpointX = (inputPoint1.X + inputPoint2.X) / 2
      //two horizontal lines are the same in all cases. For case one the two
horizontal
      //lines form the single straight one. Therefore, no special treatment.
      Tmp = drawLine(inputPoint1.X, inputPoint1.Y, midpointX, inputPoint1.Y)
      Lines[0] = tmp
      Tmp = drawLine(midpointX, inputPoint2.Y, inputPoint2.X, inputPoint2.Y)
      lines[1] = tmp
      if point1.Y != point2.Y THEN
            //drawing the vertical line
            Tmp = drawLine(midpointX, inputPoint1.Y, midpointX, inputPoint2.Y)
            Lines[2] = tmp
      END IF
      //getting the list of wires
      Wires = logicDiagram.Getwires()
      //iterating through each section of wire (lines) checking for intersections
      For I = 0 to Lines.length – 1
            Line = lines[i]
            //iterating through all the wires currently drawn
            For j = 0 to wires.Length – 1
                  Wire = wires[j]
                  //iterating through each section of drawn wire for intersections
                  For c = 0 to 2
                        drawnSection = wire.getLines()
                        If findWireIntersection(line, drawnSection[c]) == True
                              //finding the point of intersection
                              Intersection = findPointOfintersection(line,
drawnSection[c])
                              horiOrVert = determineBridgeOrientation(c)
                              //drawing the bridge on the line
                              drawWireSectionBridge(drawnSection[c], Intersection,
horiOrVert
```

```
                                //deleting the line that is being drawn
                                Lines[c] = null
                          END IF
                    NEXT C
              NEXT j
        NEXT I
END FUNCTION
```

Truth tables

Creating the headers for a truth table

To create the headers for a truth table without the intermediary steps we can use the diagrams' Boolean expression. This is because the expression contains all of the different inputs for a particular diagram. This means we can simply search through the expression looking for unique operators. The pseudocode for this is below:

```
Function generateTruthTableHeadersBasic()
      //function for generating truth table headers without steps
      Headers = [“Q”]
      booleanExpression = logicDiagram.getExpression()
      for I = 0 to booleanExpression.Length – 1
            if NOT(headers.contains(booleanExpression[i])) THEN
                  headers.prepend(booleanExpression[i])
            END IF
      NEXT I
      //sort the string into alphabetical order
      Return Headers
END FUNCTION
```

Creating the intermediary steps for a truth table

The following function creates the headers for a truth table. The algorithms generate headers for tables with intermediary steps and without by using the function previously described. This can be done with the following manner:

- Generate the list of headers without the intermediary steps and output present.
- Find the infix Boolean expression of the logic diagram.
- Convert the infix expression into postfix.
- Tokenise the postfix expression and iterate through it.
- If the character is an input (operand) add it to a stack. If the character is an operator pop the top two items and form a subexpression. Add this subexpression to the stack and a list of the created headers.
    - If the operator is a NOT gate, pop the top item on the stack as it only accepts one input.
- Add the last item on the stack to the headers.
- Return the completed list of headers with intermediary steps.

This written in pseudocode becomes:

```
Function generateTruthTableHeaders(steps)
      Headers = generateTruthTableHeadersBasic()
      Array operators = [“.”, “+”, “!”, “^”, “@”, “#”]
      If steps == True THEN
            Headers[headers.Length-1].Remove()
            //getting a non-primitive Boolean expression (no compound gates)
            //do not flatten because users will want to see the gates they have used
            //and not just AND, OR, NOT
            postfixExpression = convertInfixExpression(logicDiagram.getExpression,
False)
```

```
            Stack headingStack = new Stack()
            For I = 0 to postfixExpression.Length – 1
                  If NOT(operators.contains(postfixExpression[i])) THEN
                        Headingstack.Push(postfixExpression[i]
                  Else
                        //operator is NOT -> only pop 1 operand
                        If postfixExpression[i] == "!" THEN
                              //forming subexpression
                              Operand1 = headingStack.Pop()
                              Subexpression = postfixExpression[i] + operand1
                              headingStack.Push(subexpression)
                              Headers.Add(subexpression)
                        Else
                              //forming subexpression
                              Operand1 = headingStack.Pop()
                              Operand2 = headingStack.Pop()
                              //subexpression with operand2 first to maintain 'visual'
order
                              Subexpression = operand2 + postifixExpression[i] +
operand1
                              headingStack.Push(subexpression)
                              headers.Add(subexpression)
                        END IF
                  END IF
            NEXT I
            //the final item on the stack will be the full expression so add it.
            Headers.Add(headingStack.Pop())
      END if
      Return headers
END FUNCTION
```

Creating the Input/Output map

A truth table is a collection of inputs and outputs which forms a mapping between the two, these can be stored within a 2D array where the first dimension corresponds to the rows in the table and the second dimension are the columns. Creating the map can be done in the following manner:

- Define an array of the 2^n rows and n columns where $n = \text{number of inputs}$.
- Calculate all possible binary combinations for the given number of inputs.
- Substitute the binary combinations into the Boolean expression and evaluate it.
- Add the result of the evaluation to the respective row.
- The result is the completed Input/Output map.

Below is the process of creating all input combinations for a truth table:

- Determine the number of rows (number of different binary numbers). This can be done using 2^n where $n = \text{number of inputs}$.
- Increment by 1, starting from 0, for the number of rows in the table.
- Convert the integer to its binary representation and add it to the list of binary numbers.
- Return the collection of binary.

This creates an array of all possible binary combinations given a number of inputs.

```
Procedure generateInputOutputMap(steps, inputExpression)
```

```
	Headers = generateTruthTableHeaders(True)
	numberOfInputs = generateTruthTableHeader(False).Length - 1
	Array inputOutputMap = [2^numberOfInputs-1, headers.Length-1]
	BinaryValue = “”
	//adding the inputs to the map
	For j = 0 to 2^headers.Length-1
		Binaryvalue = convertIntToBinary(j)
		For i = 0 to binaryValue.Length-1
			inputOutputMap[I,j] = binaryValue[i]
		NEXT i
	NEXT I

	if steps == False THEN
		//evaluate the complete expression for the inputs
		For I = 0 to 2^numberOfInputs – 1
			Result = evaluateExpression(inputExpression, inputOutputMap[i])
			inputOutputMap[i][numberOfInputs + 1] = result
		NEXT I
	Else
		//adding outputs for the intermediary steps
		For I = 0 to headers.Length – 1
			subexpression = headers[i]
			for c = 0 to 2^numberOfInputs – 1
				result = evaluateExpression(subexpression, inputOutputMap[i])
				inputOutputMap[c][i] = result
			NEXT c
		NEXT i
	END IF
END PROCEDURE
```

Converting an integer into binary

Within the program, the states are stored as integers. This becomes problematic when dealing with repeated inputs within logic diagrams. Conversion between an integer and a string representation can be done with the simple function below:

```
Function convertIntToBinaryString(n)
	Remainder = 0
	binaryString = “”
	while remainder > 0
		remainder = n % 2
		binary = str(remainder) + binaryString
		remainder = remainder // 2
	END WHILE
	Return binaryString
END FUNCTION
```

Expressions

Removing compound gates

It should be noted that both the NAND and NOR gates are compound gates. This means that they can be defined in terms of other, more primitive gates, which is shown below:

- A NAND B = NOT (A AND B)
- A NOR B = NOT (A OR B)

To do this, the following steps should occur:

- Tokenise the expression.
- Iterate through the Boolean expression.
- If a compound operator (NAND or NOR) is found.
    - Find its operands.
    - Replace the operator with its simplified form using the same operands.
- Return the new expression.

This implemented in pseudocode becomes:

```
Function removeCompoundGates(inputExpression)
      //removes NAND and NOR and replaces them with primitive gates
      Operand1 = ‘’
      Operand2 = ‘’
      outputExpression = “”
      simplifiedSubExpression = “”
      lengthOfExpression = inputExpression.length-1
      for I = 0 to lengthOfExpression
            outputExpression = outputExpression + inputExpression[i]
            operand1 = inputExpression[i-1]
            operand2 = inputExpression[i+1]
            //Replacing a NAND gate
            If inputExpression[i] == ‘@’ THEN
                  outputExpression = outputExpression.Substring(0,
outputExpression.length-1)
                  simplifiedSubExpression = “!(“ + operand1 + “.” + operand2 + “)”
                  outputExpression = outputExpression + simplifiedSubExpression
            //Replacing a NOR gate
            Else if inputExpression[i] == ‘#’ THEN
                  outputExpression = outputExpression.Substring(0,
outputExpression.length-1)
                  simplifiedSubExpression = “!(“ + operand1 + “+” + operand2 + “)”
                  outputExpression = outputExpression + simplifiedSubExpression
            END IF
      NEXT I
END FUNCTION
```

Converting to postfix

Within the program, the user will enter a Boolean expression in infix notation. However, for a computer to evaluate an expression, the expression should be represented with postfix (RPN). To convert from infix to postfix, the ‘Shunting yard’ algorithm should be used. The pseudocode below is an adaptation (as the original algorithm is for arithmetic expression) of the respective Wikipedia page for the algorithm. This can be seen below:

```
/* The functions referred to in this algorithm are simple single argument functions
such as sine, inverse, or factorial. */
/* This implementation does not implement composite functions, functions with a
variable number of arguments, or unary operators. */

while there are tokens to be read:
    read a token
    if the token is:
    - a number:
        put it into the output queue
    - a function:
```

```
        push it onto the operator stack
    - an operator o₁:
        while (
            there is an operator o₂ at the top of the operator stack which is not a
left parenthesis,
            and (o₂ has greater precedence than o₁ or (o₁ and o₂ have the same
precedence and o₁ is left-associative))
        ):
            pop o₂ from the operator stack into the output queue
        push o₁ onto the operator stack
    - a ",":
        while the operator at the top of the operator stack is not a left
parenthesis:
             pop the operator from the operator stack into the output queue
    - a left parenthesis (i.e. "("):
        push it onto the operator stack
    - a right parenthesis (i.e. ")"):
        while the operator at the top of the operator stack is not a left
parenthesis:
            {assert the operator stack is not empty}
            /* If the stack runs out without finding a left parenthesis, then there
are mismatched parentheses. */
            pop the operator from the operator stack into the output queue
        {assert there is a left parenthesis at the top of the operator stack}
        pop the left parenthesis from the operator stack and discard it
        if there is a function token at the top of the operator stack, then:
            pop the function from the operator stack into the output queue
/* After the while loop, pop the remaining items from the operator stack into the
output queue. */
while there are tokens on the operator stack:
    /* If the operator token on the top of the stack is a parenthesis, then there
are mismatched parentheses. */
    {assert the operator on top of the stack is not a (left) parenthesis}
    pop the operator from the operator stack onto the output queue
```

```
Function convertInfixToPostfix(inputExpression, flattenCompounds)
      outputExpression = “”
      tmp = ‘’
      token = ‘’
      int operatorValue = 0
      operatorStack = new Stack<char>()
      //removing compound gates from the entered expression
      If flattenCompounds == True THEN
            inputExpression = removeCompoundGates(inputExpression)
      END IF

      for I = 0 to inputExpression.length-1
            if NOT(booleanOperators.Contains(inputExpression[i]) THEN
                  outputExpression = outputExpression + inputExpression[i]
            else if
                  operatorValue = getOperatorValue(inputExpression[i])
                  while (operatorStack.Count > 0) AND (operatorStack.Peek() != ‘(‘ AND
(getOperatorValue(operatorStack.Peek()) > operatorValue))
                        tmp = operatorStack.Pop()
                        outputExpression = outputExpression + tmp
                  END WHILE
```

```
            else if inputExpression == '(' THEN
                  operatorStack.Push(inputExpression[i])
            else if inputExpression == ')' THEN
                  while operatorStack.Peek() != '('
                        tmp = operatorStack.Pop()
                        outputExpression = outputExpression + tmp
                  END WHILE
                  operatorStack.pop()
            END IF
      NEXT i

      While operatorStack.Count > 0
            tmp = operatorStack.Pop()
            outputExpression = outputExpression + tmp
      END WHILE
      Return outputExpression
END FUNCTION
```

As shown above, NAND and NOR are compound operators meaning that they can be written in a simpler form. These compound gates have been removed using the `removeCompoundGates()` function.

Getting the operator value

In the 'Shunting yard' written above, the operator value is used within the function to determine the operator precedence. It should be noted that compound logic gates (NAND and NOR) do not have precedence as they can be rewritten as more primitive operators. Finding the precedence of an operator and hence its value can be done as follows:

```
Function getOperatorValue(operator)
      Array booleanOperators = ['.', '^', '+', '!']
      for I = 0 to booleanOperators.length
            if booleanOperators[i] == operator THEN
                  return I;
            END IF
      NEXT I
      Return -1
END FUNCTION
```

Evaluating a Boolean expression

To generate a truth table, the program must be able to produce the output of a Boolean expression, when substituting values into the expression. To evaluate an expression, the following steps should occur:

- Remove the compound logic gates from the Boolean expression. This will mean it is in a simpler form and this allows correct operator precedence to be observed.
- Convert the primitive expression into postfix so that it can be evaluated.
- Substitute the inputs into the expression.
- Tokenise the expression and iterate through it.
- If the token is an operand push it onto the stack.
- If the token is an operator, then pop one operand for a NOT gate or two for any other operator and form a subexpression.
- Evaluate the subexpression and push the result back onto the stack.
- Repeat for the whole expression.
- Return the last item on the stack which the result of the evaluation.

As mentioned in the steps above, we must be able to evaluate a subexpression formed between one or two operands and the operator. This can be done with the following algorithm:

```
Function evaluateSingleOperator(operator, operand1, operand2)
      // Uses custom operators for NAND and NOR due to lack of discrete characters
      // Do not need to consider not within function as it only takes one operand.
      if operator == “.” Then
            Return operand1 AND operand2
      Else if operator == “+” then
            Return operand1 OR operand2
      Else if operator == “^” then
            Return operand1 XOR operand2
      Else:
            Throw exception (“invalid operator)
      End if
End function
```

Using this algorithm, we can then (using the steps described above) evaluate a complete Boolean expression.

```
Function evalutateExpression(inputExpression, inputList)
      //creating a postfix expression in primitive form, easier evaluation
      postFixExpression = convertInfixToPostfix(inputExpression, True)
      evaluatedStack = stack()
      postfixExpression = substituteInputs(postfixExpression, inputList)
      for I = 0 to postfixExpression.Length – 1
            if inputList.Contains(postfixExpression[i]) THEN
                  evaluatedStack.Push(postfixExpression[i])
            else if postfixExpression[i] == “!” THEN
                  operand1 = evaluatedStack.Pop()
                  evaluatedStack.Push(!operand1)
            else
                  operand1 = evaluatedStack.Pop()
                  operand2 = evaluatedStack.Pop()
                  evaluatedStack.Push(evaluateSingleOperator(postfix[i], operand1,
operand2)
      NEXT I
      //final item in stack is the result of evaluation
      Return evaluatedStack.Pop()
```

Substituting inputs into an expression

To substitute the input values into a Boolean expression we need to know the expression itself and the inputs we are substituting into the expression. By then iterating through the expression where there is an operand, we can replace its token with the respective input in the list.

In the algorithm below, the function Index() return the position of the item supplied in the parameter.

```
Function substituteInputsIntoExpression(inputExpression, inputList)
      Inputs = generateTruthTableHeadersBasic()
      Inputs = inputs.Remove(inputs.Length-1)
      for I = 0 to inputExpression.Length – 1
            if inputs.contains(inputExpression[i]) THEN
                  infixExpression[i].Replace(CHR(ASC(inputExpression[i] – 65)))
            END IF
      NEXT I
```

```
    Return inputExpression
END FUNCTION
```

Diagram minimisation

To minimise a Boolean expression the Quine-McCluskey algorithm will be used, which is functionally identical to using a Karnaugh map but scales much more nicely with much larger numbers of inputs and is more compatible for computers. This method of minimisation involves three main steps, as mentioned on Wikipedia:

1. Finding all prime implicants of the expression.
2. Use those prime implicants in a prime implicant chart to find the essential prime implicants of the expression, as well as the other implicants that are necessary to cover the function.
3. Convert the essential prime implicants and prime implicants to produce the final minimised expression.

The sections below cover the process of these steps.

Finding prime implicants

The process of finding the prime implicants is as follows:

1. Produce the truth table for the Boolean expression being minimised.
2. Search through the table and produce a list of all of the input combinations where the expression evaluates to a one. This is a minterm of the Boolean expression.
3. Sort all minterms into groups where each groups contains minterms with the same number of ones within its binary representation. The groups should be sorted into ascending order.
4. Form pairs of minterms by merging them together.
    - A pair of minterms can be formed if the following criteria are met:
        i. If the number of ones within the binary input combination differs by only one binary digit.
        ii. If the dashes within the binary representation align between the minterms. This criterion only applies after the first set of merges takes place.
        iii. A minterm pair has not been formed of the same minterms previously.
    - If a pair is formed, then replace the differing bit between the minterms with a dash.
5. Repeat step 4 until a merge can no longer be made. The resulting groups of minterms are the prime implicants.

The algorithm above can be written with a series of functions which written in pseudocode becomes:

```
FUNCTION getPrimeImplicants(mintermList)
    List primeImplicants = new List()
    //stores which minterms have been merged
    Array merges = array[mintermList.Count]
    NumberOfMerges = 0
    MergedMinterm = “”
    String m1
    String m2
    For I = 0 to mintermList.Count – 1
        For c = I + 1 to mintermList.Count – 1
            M1 = mintermList[i]
            M2 = mintermList[c]
            //A merge can occur if dashes align, and one bit differs in minterms
            If checkDashesAlign(m1, m2) == True AND checkMintermDifference(m1,
        m2) == True THEN
                mergedMinterm = mergeMinterms(m1, m2)
                primeImplicants.Add(mergeMinterms)
                numberOfMerges = numberOfMerges + 1
                merges[i] = true
```

```
                        merges[c] = true
                  END IF
            NEXT c
      NEXT I
      //This loop ensures that prime implicants from previous calls are retained.
      For j = 0 to mintermList.Count – 1
            If merges[j] == False AND primeImplicants.Contains(mintermList[j]) ==
False THEN
                  primeImplicants.Add(mintermList[j])
            END IF
      NEXT j
      //Function should keep recursing, trying to find more prime implicants until no
more can be found.
      If numberOfMerges == 0 THEN
            Return primeImplicants
      Else
            Return getPrimeImplicants(primeImplicants)
      END IF
END FUNCTION

//A merge can only be formed if the dashes within the minterms align.
FUNCTION checkDashesAlign(minterm1, minterm2)
      If minterm1.Length != minterm2.Length THEN
            Throw length Exception
      ELSE
            For I = 0 to minterm1.Length
                  If minterm1[i] != minterm2[i] AND minterm1[i] == '-' THEN
                        Return False
                  END IF
            NEXT I
            Return True
      END IF
END FUNCTION

//Checks that two minterms differ by only one bit.
FUNCTION checkMintermDifference(minterm1, minterm2)
      Minterm1 = Minterm1.Replace('-', '0')
      Minterm2 = Minterm2.Replace('-', '0')
      If minterm2 – minterm1 == 0 THEN
            Return FALSE
      //Only one bit differs if the log of the difference is a whole number
      Else If log(2, ABS(minterm2 – minterm1)) MOD 2 == 0
            Return TRUE
      ELSE
            Return FALSE
      END IF
END FUNCTION
```

Finding the essential prime implicants

The process of finding the essential prime implicants can be done with the following steps:

- Create a dictionary where the keys are the prime implicants found in the previous step and the values are empty string.
- Iterate through each prime implicant (key in the dictionary) and carry out the following process:
    - Replace any dashes within the prime implicant with the string **"\d"**. The resulting string now forms a regular expression that can be evaluated.
    - Iterate through the list of minterms being minimised and compare the regular expression with the minterm.
        - If the result of the comparison is successful, then add a one to the key of the respective prime implicant.
        - Otherwise add a zero.
    - Repeat for all prime implicants.
- Create an array of the same length of the minterms.
- Read and sum the number of one's within each place value column and set the respective position in the array to the total. This forms a frequency table where each item is the number of one's in a place value column.
- Iterate through the frequency table:
    - If the frequency is a one, then:
        - Search through the values in the dictionary until a one is found in the same position as the frequency table.
        - The respective key of the value is therefore an essential prime implicant.
        - Add the essential prime implicant to a list.
    - Otherwise continue.

The algorithm described above will produce a list of essential prime implicants which can be converted to produce the final expression. The algorithm described above can be written in pseudocode code:

```
FUNCTION setRegexPatterns(regex, minterms)
      Array keys = regex.Keys.ToArray()
      Key = “”
      Res = False
      For I = 0 to keys.Count – 1
            Key = keys[i]
            For c = 0 to minterms.Count – 1
                  Res = CompareMintermAndImplicant(minterm, key)
                  If res == True THEN
                        Regex[key] = regex[key] + “1”
                  Else
                        regex[key] = regex[key] + “0”
                  END IF
            NEXT c
      NEXT I
END FUNCTION

FUNCTION getEssentialPrimeImplicant(regex, pos)
      Array essentialPrimes = regex.Values.ToArray()
      Array keys = regex.Keys.ToArray()
      Prime = “”
      For I = 0 to essentialPrimes.Count – 1
            Prime = essentialPrimes[i]
            If prime[pos] == ‘1’ THEN
                  Return keys[i]
            END IF
```

```
	NEXT i
END FUNCTION

FUNCTION getEssentialPrimeImplicants(regex, minterms)
	Array frequencyTable = getFrequencyTable(regex, minterms)
	List essentialPrimeImplicants = new List()
	Epi = “”
	For I = 0 to frequencyTable.Length – 1
		If frequencyTable[i] == 1 THEN
			Epi = getEssentialPrimeImplicant(regex, i)
			If essentialPrimeImplicants.Contains(epi) == False THEN
				essentialPrimeImplicants.Add(epi)
			END IF
		END IF
	NEXT i
	Return essentialPrimeImplicants;
END FUNCTION


//Produces the frequency table for the values within the dictionary
FUNCTION getFrequencyTable(regex, minterms)
	Array sums = array[minterms.Count]
	S = “”
	For I = 0 to regex.Values.ToList().Count
		S = regex.Values.ToList()[i]
		For c = 0 to S.Length – 1
			If s[c] == ‘1’ THEN
				Sums[c] = sums[c] + 1
			END IF
		NEXT c
	NEXT I
	Return sums
END FUNCTION
```

Finding the minimised expression.

The essential prime implicants of the boolean expression being minimised must be converted to form the final expression. This can be done with the following algorithm:

- Iterate through the list of the essential prime implicants.
- Iterate through each character of the essential prime implicants.
- If the character of the essential prime implicant is a one, then convert the position of the 1 into ASCII by adding 65 and add it to the return string.
- If the character is a zero, then convert the position into ASCII but add the complement to the return string.
- Repeat this process for each of the essential prime implicants. Separating each essential prime implicant with a logical OR.

This algorithm can be described with pseudocode as described below:

```
FUNCTION convertImplicantToExpression(epi)
	Expression = “”
	//Removing the regex chars.
	Epi.Replace(“\d”, “-“)
	For I = 0 to epi.Length - 1
```

```
            If epi[i] == '1' THEN
                  Expression = expression + ASC(I + 65)
            ELSE IF epi[i] == '0' THEN
                  Expression = expression + "!" + ASC(I + 65)
            END IF
      NEXT I
      Return expression
END FUNCTION

//Implicants are separated by OR.
//Produces the final expression to be outputted.
FUNCTION getMinimisedExpression(essentialPrimeImplicants)
      minimisedExpression = ""
      For I = 0 to essentialPrimeImplicants.Length – 1
            Epi = essentialPrimeImplicants[i]
            Expression = Expression + "+" + convertImplicantToExpression(epi)
      NEXT I
      Return minimisedExpression
END FUNCTION
```

Petrick's Method

It should be noted that for some expressions the essential prime implicants do not cover all of the expressions' minterms. This could be due to a cyclic prime implicant chart or a strange distribution of minterms. This means that with the current set of algorithms, the program will not always find the minimised expression. To ensure this is not the case then Petrick's method should be used. The algorithm is described below (From Wikipedia).

- Reduce the prime implicant chart by eliminating the essential prime implicant rows and corresponding columns.
- Label the rows of the reduced prime implicant chart $P_1,\ P_2,\ P_3,\ P_4,$ etc.
- Form a logical function P which is true when all of the columns of the reduced chart are covered. P consists of a product of sums where each sum term has the form $(P_{i0} + P_{i1} + \cdots + P_{iN})$, where each P_{ij} represents a row covering column i.
- Apply the Distributive Law, $(A + B)(A + C) \equiv A + (B\ .\ C)$ to expand P into a sum of products and minimise by applying the absorption law $X + (X\ .\ Y) \equiv X$.
- Each term in the result represents a solution, that is, a set of rows which covers all of the minterms in the table to determine the minimum solutions, first find those terms which contain a minimum number of prime implicants.
- Next, for each of the terms found in the previous step, count the number of literal in each prime implicant and find the total number of literals.
- Choose the term or terms composed of the minimum total number of literals and write out the corresponding sums of prime implicants.

To find the product of sums of the reduced prime implicant chart the following algorithm can be used which simply iterates through the chart.

```
FUNCTION getProductOfSums(PrimeImplicantChart, termToImplicantMap)
      productOfSums = new list()
      listOfBrackets sumsToAdd
      primeImplicant = ""
      key = ""
      for I = 0 to primeImplicantChart.Keys.Length
            key = primeImplicantChart.Keys[i]
            primeImplicant = primeImplicantChart[key]
```

```
            for c = 0 to primeImplicant.Length
                  if primeImplicant[i] == '1' THEN
                        //search through the column to make any pairs possible
                        sumsToAdd = GetSumsToAdd(primeImplicantChart,
termToImplicantMap, key, i)
                        //Adding the found pairs to the product.
                        AddSumsToList(productOfSums, sumsToAdd)
                  END IF
            NEXT c
      NEXT I
      Return productOfSums
END FUNCTION
```

The following set of algorithms cover the processing finding product of sums of the Boolean expression. The general method is described by the steps below:

- Merge each product within the product of sums using the distributive law, this creates the initial conditions to then continue with algebraic manipulation.
- Recursively apply the distributive law to expand each of the product of sums to produce the sum of products.

```
FUNCTION GetSumOfProducts(ProductOfSums)
      Bool merged = True
      WHILE merged == True
            Merged = False
            For I = 0 to productOfSums.Count – 1
                  For c = I + 1 to productOfSums.Count – 1
                        B1 = productOfSums[i]
                        B2 = productOfSums[c]
                        // Making sure brackets are not the same
                        if B1 != B2
                              mergedTerm = MergeBrackets(b1, b2)
                              productOfSums.Add(mergedTerm)
                              productOfSums.Remove(b1)
                              productOfSums.Remove(b2)
                              merged = true
                              I = c + 1
                              C = I + 1
                        END IF
                  NEXT c
            NEXT I
      END WHILE
      // Produces string representation of the brackets, eg "(A+B)", "(C+D)".
      stringProducts = ConvertBracketsToString(productOfSums)
      sumOfProducts = RecursiveDistributiveLaw(stringProducts)
      return sumOfProducts[0]
END FUNCTION

FUNCTION RecursiveDistributiveLaw(brackets)
      Lls = new List<List()>
      If brackets.Count > 1 THEN
            // Apply distributive law between two brackets.
            Lls.Add(SingleDistributiveLaw(brackets[0], brackets[1]))
```

```
            // Remove the brackets that are being distributed.
            Brackets.Remove(0)
            Brackets.Remove(0)
            Lls.Add(brackets)
            // Merges still could take place, so start again.
            Return RecursiveDistributiveLaw(lls)
      ELSE
            // Distributive law can no longer be applied.
            Return brackets
      END IF
END FUNCTION
```

Human-Computer interface

The following section shows the graphical user interface (GUI) of the program. It includes all possible windows, sub-windows, and dialog boxes, with exception to open/save dialogs as these are built into WPF. I have followed the same general design principles.

- The user interface should be minimalistic (in line with the responses to my student questionnaire, see Appendix B).
- There should be few menus and sub-menus as to avoid a user spending a large amount of time searching for features (in line with objective 1).
- Messages should be detailed but not overly as to confuse younger users of the program (as mentioned in objective 4).

Main window

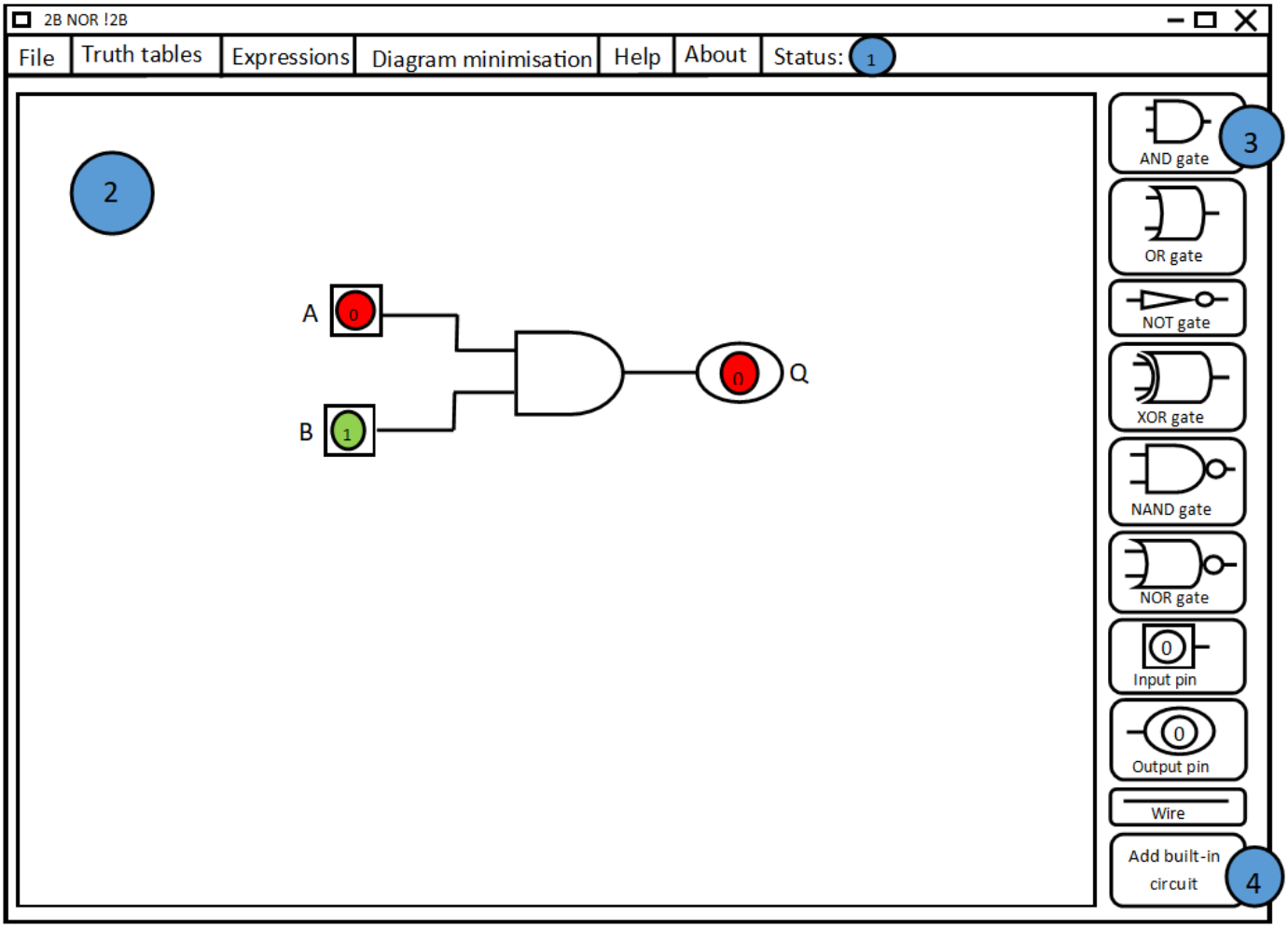


Figure 8: The main window of the program

This window is the first window that is shown to the user when the program is ran. My goal when creating the diagram, shown above, is to put as much functionality into one window whilst keeping the window uncluttered. This would reduce the number of windows and hidden buttons within the program and hence, simply its usage (in accordance with objective 1). When the program is opened this window will be shown in full screen meaning there is no need to consider the resizing of the window. This is to allow for the largest diagram creation window possible. If the window size is decreased, such as when the window is within split screen, only the size of the diagram creation window will decrease which ensures that the side panel is as readable as possible.

In the diagram above, there are some notable constituents of the main window which include:

1. This is the menu strip which will allow the user to navigate through the functionality of the program. Relevant names have been added to make navigation as easy as possible. Most of the items within the strip will simply open sub-windows dedicated to the relevant content (these sub-windows can be seen in following sections). The 'File' item is the exception, in which it shows a simple list of file handling processes. These being saving, loading, and exporting.
2. This is the previously mentioned diagram creation window. This window is where the diagrams will be drawn when the user enters a Boolean expression. A minimised diagram will also be drawn in this window. Input pins within the window be interactive to allow for the user to dry-run input states within the main window, wires will change colour to reflect the state of the diagram.
3. This is the side-bar that will act as a place where students can find out information for each of the logic gates by opening a pop-up window. This side-bar removes the need for a drop-down menu, further simplifying the

GUI. The elements have been shown both in text and pictorial form to help users to identify which components are which in case they are unsure.

4. This is a simple button which opens a dialog box, shown below. This allows users to add pre-made circuits such as adders to the diagram creation window. This item is not within the menu strip above because it relates to diagram creation and does not warrant its own separate window. The button will also make its function very clear and easy to access. This can be seen below.

Truth table window

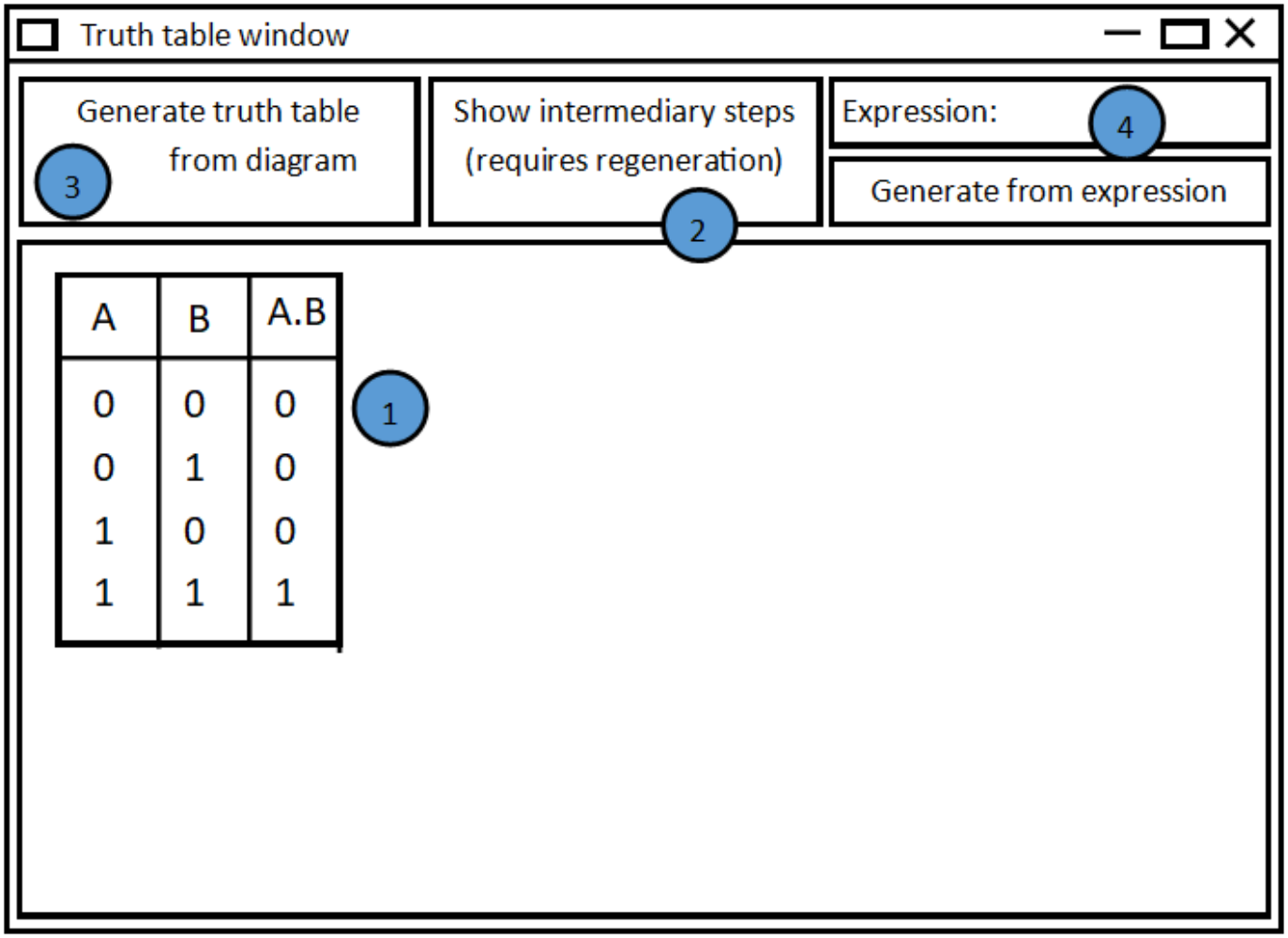


Figure 9: The truth table window

This sub-window can be accessed through the menu strip at the top of the main window, shown above. This window will be resizable, this to give enough room to draw for much larger truth tables. The elements within this sub-window include:

1. The Truth table canvas. This is the area in which the generated truth table is drawn and displayed to the user. An example truth table for an AND gate is shown above. The majority of the window space is allocated to this component as generated truth tables can become very large with complex diagrams with many inputs. This means a large space is needed to represent all of the data.
2. A button for toggling the intermediary steps. When drawing truth tables, there are two styles that they can take. One of these includes the intermediary steps when dry-running a diagram and the other only displays the inputs and outputs. Both are very useful for a student as both can be given in an exam question. A teacher could also make use of this to help students learn the process of dry-running logic gate diagrams. The example above does not show these steps however, they would simply be extra columns of the sub-expressions, with their respective output below. As the button is toggle-able, its state will be indicated

through its colour, this will change when clicked by the user. It should be noted (as indicated on the button) that the truth table will need to be regenerated to include the new state of the 'intermediary steps' button.

3. This is the button that the user will press to generate the truth table. The program will automatically read the contents of the diagram data structure and generate the table, making the process as convenient for the user as possible.
4. Also, for convenience, there is also a text box and button to allow the user to generate truth tables straight from expressions. This will cater for users who quickly want to generate a truth table without going through the process of drawing a diagram. This text box will follow the same schema as the expression window, this will be made clear in the help window. It will also be validated to ensure that the program will not break trying to generate an invalid expression.

Boolean expression window

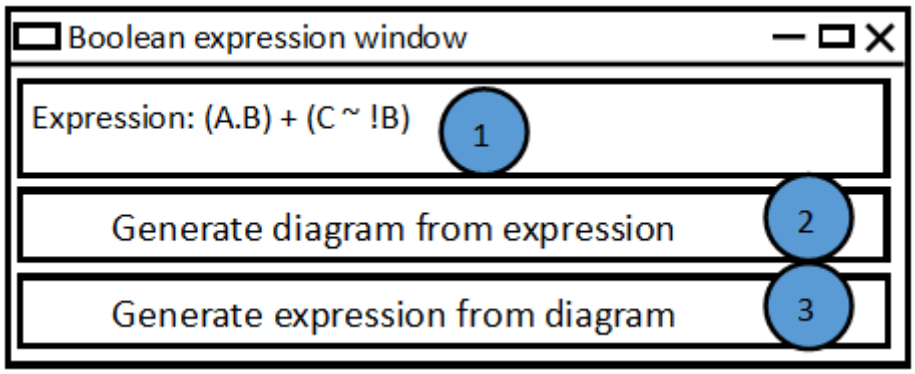


Figure 10: The Boolean expression window

This sub-window can be accessed through the menu strip at the top of the main window, shown above. This sub-window is very minimal as there are few visual elements of the expression aside from the expression itself and hence, there is no need for more details that would clutter the sub-window and confuse new users. This also means that this window will not need to be resized due to the lack of components. The labelled constituents include:

1. Expression box. This is simple text box that both shows expressions and is the point where users can input their own expressions depending on the button pressed below. It should be noted that the expression window will use a semi-custom schema to represent all of the different logic gates. This schema can be found within the 'help' sub-window.
2. This button, as described, generates logic diagram from the Boolean expression entered into the expression box above. When the button is pressed, this triggers validation of the expression and if the expression if valid then the diagram will be generated. This diagram will be shown in the diagram creation window with all inputs states as 'off' or zero. When the process is complete, the user will be notified through both the status box and a message box saying the process is complete and the next steps to take. It must be said that, if any elements are within the diagram class, then the diagram will not be generated to protect users work.
3. The button performs the opposite process to the one above. This is done by reading the diagram class using a postfix traversal and then reordering the expression and adding brackets where necessary. The completed expression will be displayed within the expression box for convenience and to also maintain a minimal profile. It should be noted that it is possible for elements to be within the diagram creation window and not be added to the expression. This is because these components must be added to the diagram class to be read.

Diagram minimisation window

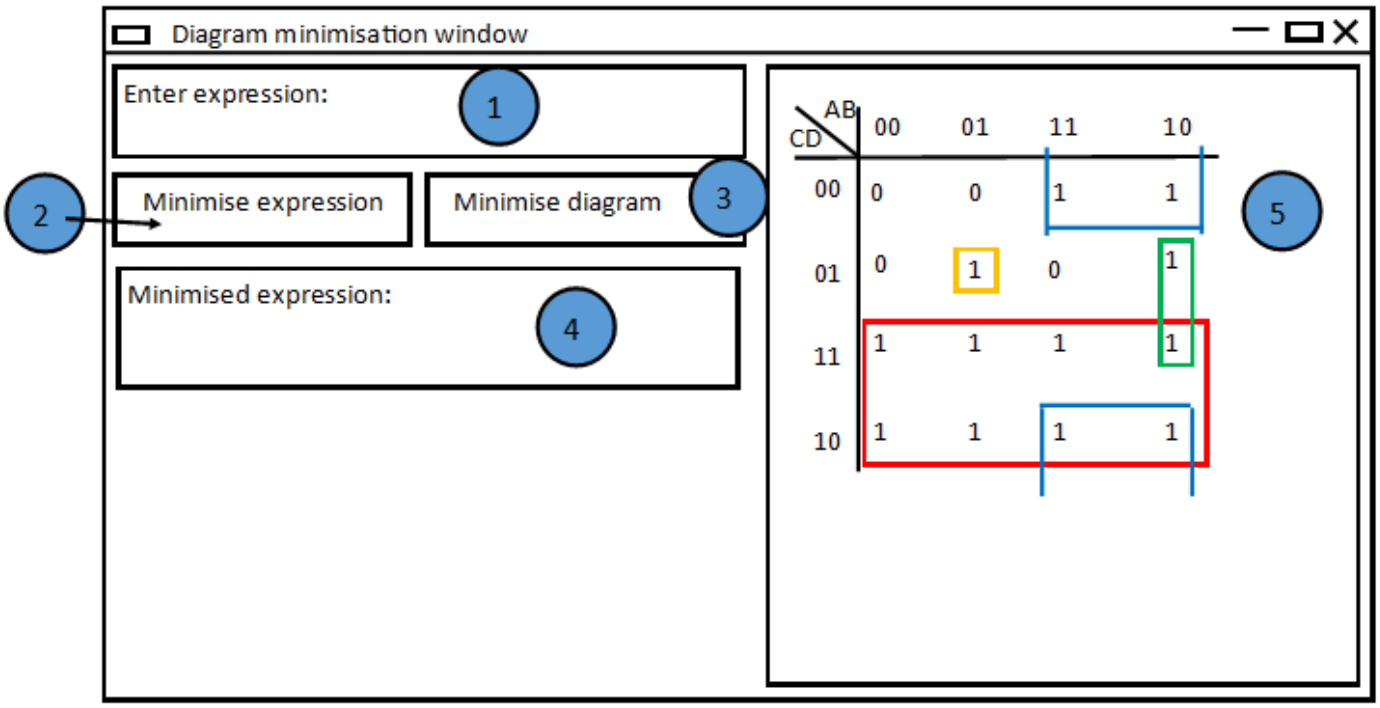


Figure 11: The diagram minimisation window

This sub-window can be opened when the user selects the respective item from the menu-strip in the main window. This window will not be resizable within the program and will simply 'lie on top of' the main window. This is to maintain a constant canvas size which makes it easier to draw the Karnaugh maps, it also allows the user to see the minimised diagram within the diagram creation window as soon as possible. Some notable components of this sub-window include:

1. Expression box. This is where the user can enter Boolean expressions into the program to be minimised without needing to draw a diagram.
2. When this button is clicked the expression that has been entered into the textbox above will be minimised. Before minimisation takes place, the program will validate the entered expression to ensure that it can be minimised. The result of this validation will be shown to the user using a message box. After validation, the Karnaugh map will be generated, and the expression will be minimised.
3. When this button is clicked the diagram class that represents the logic diagram currently drawn within the diagram creation window will be read and the expression will be generated. The program will set the expression box to the new expression to show the user the starting expression. The expression will be validated before the expression box is set as to not present invalid data to the program. The result of the validation will be presented to the user within a message. After the expression box is set the expression will be minimised.
4. Minimised Expression box. This is a textbox that displays the minimised expression produced from the Karnaugh map. This text box will be filled both when the expression is minimised from the expression box at the top of the sub-window and also when minimising from the logic diagram drawn by the user within the diagram creation window.
5. This is the canvas that will show the user the Karnaugh map that has been drawn by the program. The canvas will also display the groups of ones that have been identified by the program. These will be coloured

to make the groups clearer. The canvas will not show the process of using the identified groups to form the minimised expression.

Help window

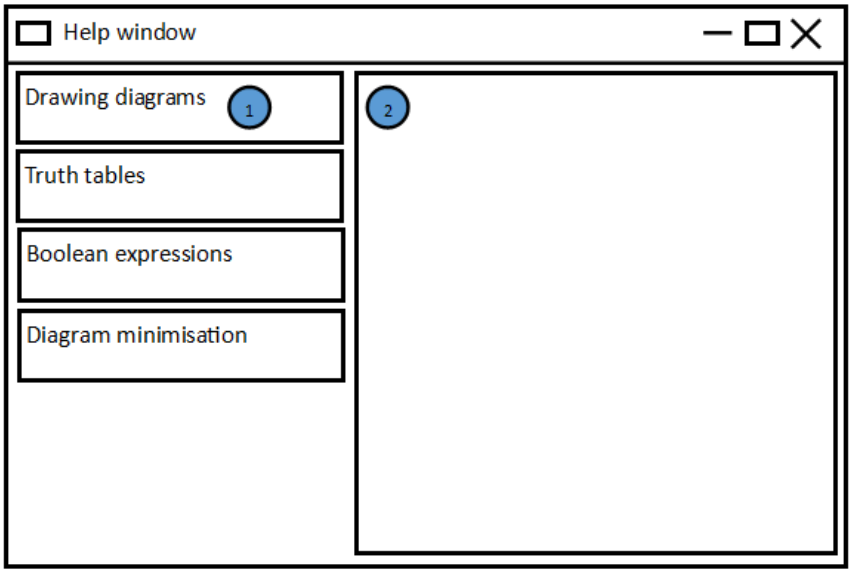


Figure 12: The help window

This sub-window is for users to get help if they get stuck when using the program. In the program:

1. This is a list of buttons that sets the text box [2] on the right side of the sub-window. These will not resize with the sub-window so to not occupy a large amount of window space.
2. Description text box. This the text box that displays the information requested by the user. When the sub-window is resized, this component is the one to take the majority of the window space. This to present as much information to the user as possible. This will reduce the amount of time a user will spend searching through the

When resizing this window, only the description text box will change size. This is to keep the side buttons as clear as possible when the window is being reduced in size or to reduce the amount of scrolling in the textbox when the window is maximised.

About window

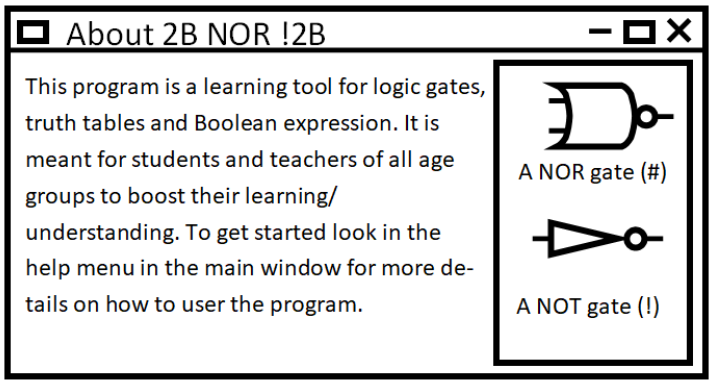


Figure 13: The about window

The sub-window does not serve any functional purpose however, it has been added for completeness. It could also make users feel less intimidated when using the program as there is a logical 'next step' or place to go within the program. This sub-window includes information on the help sub-window as that is where technical detail and how to use the program can be found.

Error messages

The following section details on the range of error message that can be shown to the user, if they make a mistake when using the program, with example text describing the nature of the error and its possible fixes. The goal of the error messages is to be as helpful as possible (in line with objective 4) whilst also being minimal in size and with simple descriptions to help younger users of the program to understand the error they have made. An example error message is shown below. This is displayed when an invalid Boolean expression is entered within the program:

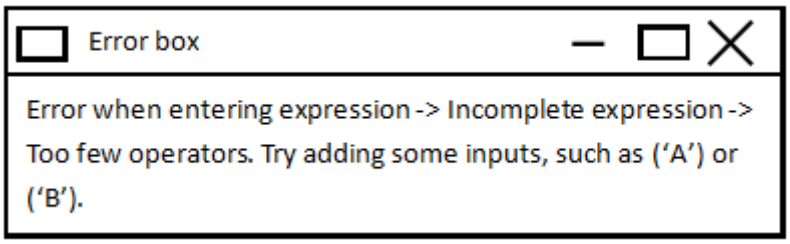


Figure 14: An example error message within the program.

The error box will not change, aside from the message within it depending on the error that has been generated. The general information contained within the message will be:

- The nature of the error, such as, invalid Boolean expressions or incomplete diagrams.
- The cause of the error, such as, too few operands or incorrect number of brackets.
- Possible fixes to the error.

System security

Security of data

The program does not store or process data that is sensitive in nature. Therefore, no special consideration must be made to conform to data protection laws. The program is educational in nature and so no malicious code will be contained within it. As the program can be compiled into an executable file, this reduces the chances of the program being tampered or edited. Due to the program being offline and not requiring a database there is no need to consider the risk of SQL injection or network attacks.

Integrity of data

The program does save/load and export files. This does present a small risk of file corruption; however, the program will include relevant error messages to notify the user if corruption has taken place and the data cannot be loaded/saved if this does happen.

Test strategy

In my opinion, testing can be split into two main groups. These being internal and external testing where internal testing is conducted by myself and external is done by someone else.

Testing during development

When programming, testing will be carried out throughout the development process to check that individual functions and methods in the desired manner. This will also allow me to build a strong foundation within the program to ensure that when creating more advanced features the program does not present errors that would require large reworking of the program.

When testing, a variety of inputs will be given to the program. Inputs should come in the following categories: valid, invalid, boundary and erroneous. Each category should be tested throughout the development process to ensure robustness for all cases of input data. To test these categories, the following input types should be checked:

- Erroneous inputs
    - Inputs that are of the incorrect type. For example, an integer instead of a string.
    - Inputs that are out of range. An example of this using an array of too few or too many items.

- Invalid inputs
    - Inputs that are not in the correct order. Entering an expression with all operands then operators. Such as: ABC.+. Which includes valid characters but are not in a useful order.
- Boundary inputs
    - Inputs that are on the edge of a range. This is to ensure that a range covers the correct area. Examples include the groups identified within the Karnaugh map are 2^n in size.
- Valid inputs
    - Any input value that do not infringe on the constraints listed above. Some examples include a valid expression such A . B or a completed diagram within the diagram creation window.

User testing

It must be said that even if internal testing is exhaustive there are still possible errors that are unaccounted for. This is where external testing can be used because it can find errors in many cases that internal testing does not consider. For the program, two types of external test will be conducted. One by the user (students) and another done by the client. This will ensure that the program fits the needs of the client and covers the necessary bases and also users will put the program through many cases that I have not considered which will create a more rigorous test.

Client testing

Client testing will ensure that the program is made to the correct standard and covers all of the necessary content to ensure that the program is useful and fits the client's needs. To conduct client testing, the client will be given the completed program and then be asked to conduct a review of the software and to write down their thoughts on the program. The client will also be given three checklists, each coming from the specifications from each type of students (these being GCSE and A-level students). They should then tick an item if it has been covered in the program and they will also be prompted to give some thoughts on how the item has been implemented within the program. This is to make sure that the program does not simply do everything in the specifications, but it excels at them. An example of the checklist for an A-level student is shown below:

Specification item	Completed	Further comments
Construct truth tables for the following logic gates: • NOT • AND • OR • XOR • NAND • NOR.	□	*Teachers comments about constructing truth tables within the program.*

User testing

Aside from client testing, user testing will also be conducted to make sure that the program is effective for its users. This is because a program could be made to the correct specification, however, it is very difficult to use and hence, is not a good program.

Users will be asked to complete a variety of tasks. This is to make sure that the objectives outlined in the analysis are complete but to also test a wider variety of cases to ensure the program does not crash. They will also be encouraged to enter weird expressions and type/draw a variety of things. This means that user testing is much more effective way to do this as they can test many more unique cases and can test situations that I have not thought of myself. User testing also indicates that the program works correctly 'on the ground' and is effective for students. After using the program, students will be asked to complete a questionnaire about their experience of the program. It should be noted that user testing must be conducted with all age groups of students, this ensures that a wide range of data/opinions is/are collected about the program. This will help to evaluate the robustness of the program. User testing will also help the development process by showing me inputs that crash my program and so I can fix them to create a more robust program.

Technical solution

The following section is a full code listing of the project written using C# and WPF.

It should be noted that the LATEX renderer being used within the program is taken from the xaml math repository on GitHub. However, the conversion into LATEX is done by a Boolean converter written specifically for the program, it is exclusively the rendering that is handled by the repository. Usages of the library are within the expression input dialog, the output of the minimisation and the expression in the pop-up menus.

Coding Conventions

As mentioned within the objectives of the program. The code written follows the Microsoft Coding Conventions for C#. The code follows the convention as it means the code is readable, consistent with industry standards and it makes it easier for other developers to use the code written for this program. An extract from the conventions is shown below:

"We chose our conventions based on the following goals:

1. *Correctness: Our samples are copied and pasted into your applications. We expect that, so we need to make code that's resilient and correct, even after multiple edits.*
2. *Teaching: The purpose of our samples is to teach all of .NET and C#. For that reason, we don't place restrictions on any language feature or API. Instead, those samples teach when a feature is a good choice.*
3. *Consistency: Readers expect a consistent experience across our content. All samples should conform to the same style.*
4. *Adoption: We aggressively update our samples to use new language features. That practice raises awareness of new features and makes them more familiar to all C# developers." – Microsoft*

The program follows the following conventions, these have been taken directly from the Microsoft website. All guidelines that have been given have been followed where possible. Some guidelines could not be followed as there were no instances of their use, such as the use of delegates. The program also

The table below shows all of the guidelines that have been followed within the program because they apply specifically. This is because some of the guidelines do not apply to the program as some technique are not being used at all, such as using delegates. The program also follows the guidelines given by Microsoft for C# identifier naming rules and conventions. Additionally, the program also follows the guidelines for commenting. Employing XML comments for all classes and methods and "single-line comments (//) for brief explanations"(Microsoft, 2024).

Convention	Followed
Use string interpolation to concatenate short strings.	Y
Use the concise syntax when you initialise arrays on the declaration line. This is shown below: • **String[] vowels1 = { "a", "e", "i", "o", "u" };** **If you use explicit instantiation, you can use var.**	Y
Use a try-catch statement for most exception handling.	Y
Simplify you code by using the C# using statement.	Y
Use && instead of & and || instead of | when you perform comparisons.	Y
Use one of the concise forms of object instantiation, as shown in the following declarations, as shown below: • **var firstExample = new ExampleClass();** • **ExampleClass instance2 = new();**	Y
Use object initialisers to simplify object creation, as shown in the following example: • **var thirdExample = new Example { Name = "Desktop", ID = 37414, Location = "Redmond" };**	Y
Use implicit typing for the loop variable in for loops.	Y

Do not use implicit typing to determine the type of the loop variable in foreach loops.	Y
When a "using" directive is outside a namespace declaration, that imported namespace is its fully qualified name.	
Use four spaces for indentation and do not use tabs.	Y
Align code consistently to improve readability.	Y
Limit lines 65 characters to enhance code readability on docs, especially on mobile screens. Note: This project follows a limit of 100 characters, to ensure that the formatting of the code snippets is not unreasonable.	Y
Break long statements into multiple lines to improve clarity.	Y
Use the "Allman" style for braces: open and closing brace its own new line. Braces line up with the current indentation level.	Y
Line breaks should occur before binary operators, if necessary.	Y
Use single-line comments for brief explanations.	Y
Avoid multi-line comments for longer explanations. Comments are not localised.	Y
For describing methods, classes, fields, and all public member use XML comments.	Y
Place the comment on a separate line, not at the end of a line of code.	Y
Begin comment text with an uppercase letter.	Y
End comment text with a full-stop.	Y
Insert one space between the comment delimiter and the comment text.	Y
Write only one statement per line.	Y
Write only one declaration per line.	Y
If continuation lines are not indented automatically, indent them one tab stop (four spaces).	Y
Add at least one blank line between method definitions and property definitions.	Y
Use brackets to make clauses in an expression apparent.	Y

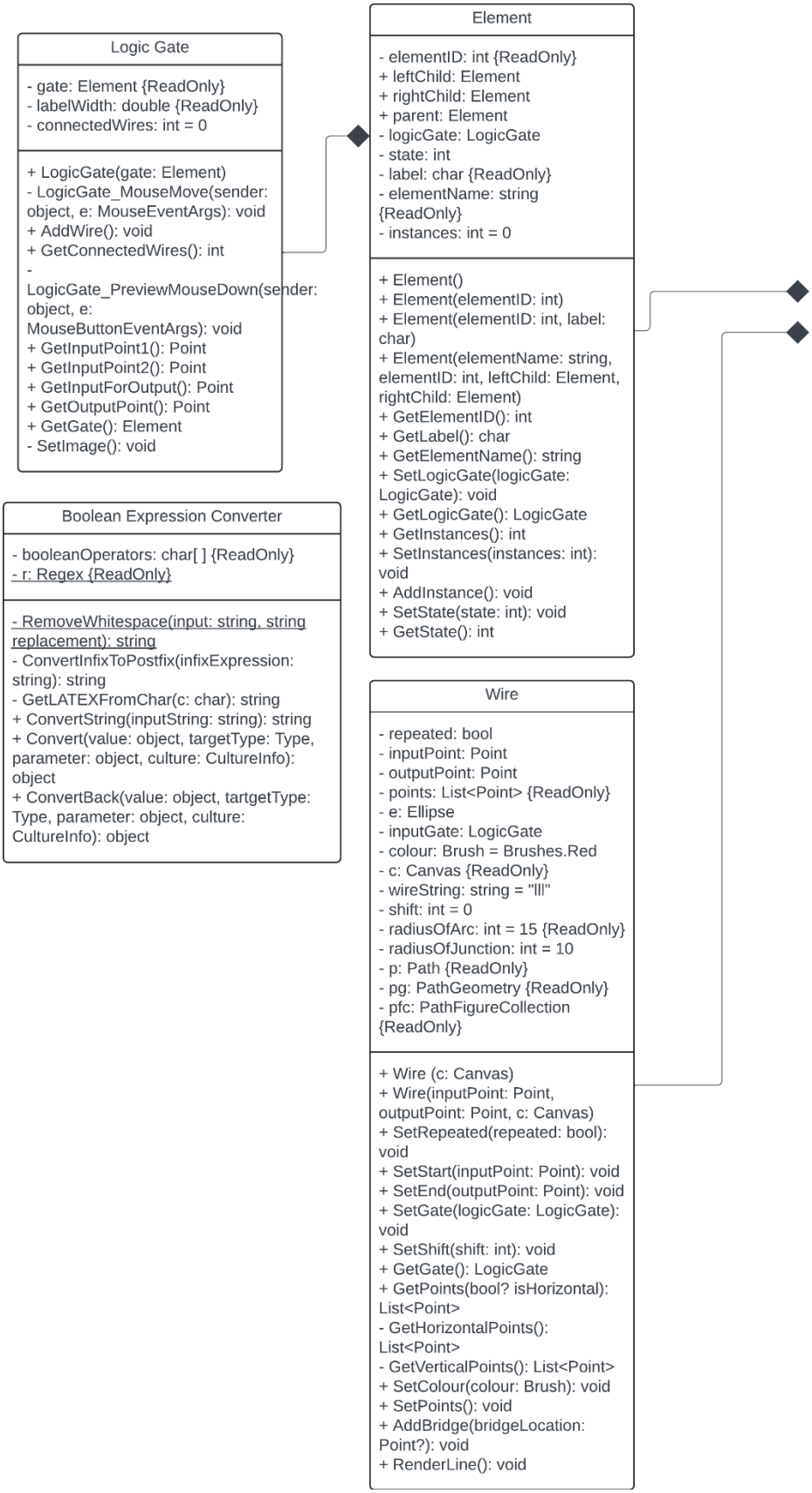
Updated UML class diagram

Below is an updated UML class diagram of the program. This is to reflect the changes that have been made to the program. It must be said that the program significantly extends the scope of my design for the program.

One of the main differences between the new class diagram and the one in design is the addition of the LogicGate class and Bracket struct. These, although not as extensive in comparison to the wire class, serve very important functions in handling and storing data. An example in this is in the minimisation methods such as GetSumOfProducts where the merging process is made easier due to the Bracket struct packaging the data in a easily interactable form. The logicGate class acts as the bridge between the user interface and the diagram class. This class was very useful as it helped solve the problem of expressions with repeated inputs. This is because it allows the program to have elements within the binary tree (such as the repeated input) but not display it on the canvas which creates the illusion that the logic gate diagram is not a binary tree.

The addition of the BooleanConverter serves as a utility class for the UI, allowing a rendered LATEX expression to be shown to the user in multiple different windows. It makes it possible to use data binding to have a rendered expression that updates to the expression that the user is currently inputting.

It must be noted that the composition arrows from the Element and Wire classes go into the diagram class. This class has been put onto separate pages to ensure that fields and methods are readable due to the number of methods.


Logic Gate
- gate: Element {ReadOnly}
- labelWidth: double {ReadOnly}
- connectedWires: int = 0
+ LogicGate(gate: Element)
- LogicGate_MouseMove(sender: object, e: MouseEventArgs): void
+ AddWire(): void
+ GetConnectedWires(): int
- LogicGate_PreviewMouseDown(sender: object, e: MouseButtonEventArgs): void
+ GetInputPoint1(): Point
+ GetInputPoint2(): Point
+ GetInputForOutput(): Point
+ GetOutputPoint(): Point
+ GetGate(): Element
- SetImage(): void
Element
- elementID: int {ReadOnly}
+ leftChild: Element
+ rightChild: Element
+ parent: Element
- logicGate: LogicGate
- state: int
- label: char {ReadOnly}
- elementName: string {ReadOnly}
- instances: int = 0
+ Element()
+ Element(elementID: int)
+ Element(elementID: int, label: char)
+ Element(elementName: string, elementID: int, leftChild: Element, rightChild: Element)
+ GetElementID(): int
+ GetLabel(): char
+ GetElementName(): string
+ SetLogicGate(logicGate: LogicGate): void
+ GetLogicGate(): LogicGate
+ GetInstances(): int
+ SetInstances(instances: int): void
+ AddInstance(): void
+ SetState(state: int): void
+ GetState(): int
Boolean Expression Converter
- booleanOperators: char[ ] {ReadOnly}
- r: Regex {ReadOnly}
- RemoveWhitespace(input: string, string replacement): string
- ConvertInfixToPostfix(infixExpression: string): string
- GetLATEXFromChar(c: char): string
+ ConvertString(inputString: string): string
+ Convert(value: object, targetType: Type, parameter: object, culture: CultureInfo): object
+ ConvertBack(value: object, tartgetType: Type, parameter: object, culture: CultureInfo): object
Wire
- repeated: bool
- inputPoint: Point
- outputPoint: Point
- points: List<Point> {ReadOnly}
- e: Ellipse
- inputGate: LogicGate
- colour: Brush = Brushes.Red
- c: Canvas {ReadOnly}
- wireString: string = "lll"
- shift: int = 0
- radiusOfArc: int = 15 {ReadOnly}
- radiusOfJunction: int = 10
- p: Path {ReadOnly}
- pg: PathGeometry {ReadOnly}
- pfc: PathFigureCollection {ReadOnly}
+ Wire (c: Canvas)
+ Wire(inputPoint: Point, outputPoint: Point, c: Canvas)
+ SetRepeated(repeated: bool): void
+ SetStart(inputPoint: Point): void
+ SetEnd(outputPoint: Point): void
+ SetGate(logicGate: LogicGate): void
+ SetShift(shift: int): void
+ GetGate(): LogicGate
+ GetPoints(bool? isHorizontal): List<Point>
- GetHorizontalPoints(): List<Point>
- GetVerticalPoints(): List<Point>
+ SetColour(colour: Brush): void
+ SetPoints(): void
+ AddBridge(bridgeLocation: Point?): void
+ RenderLine(): void

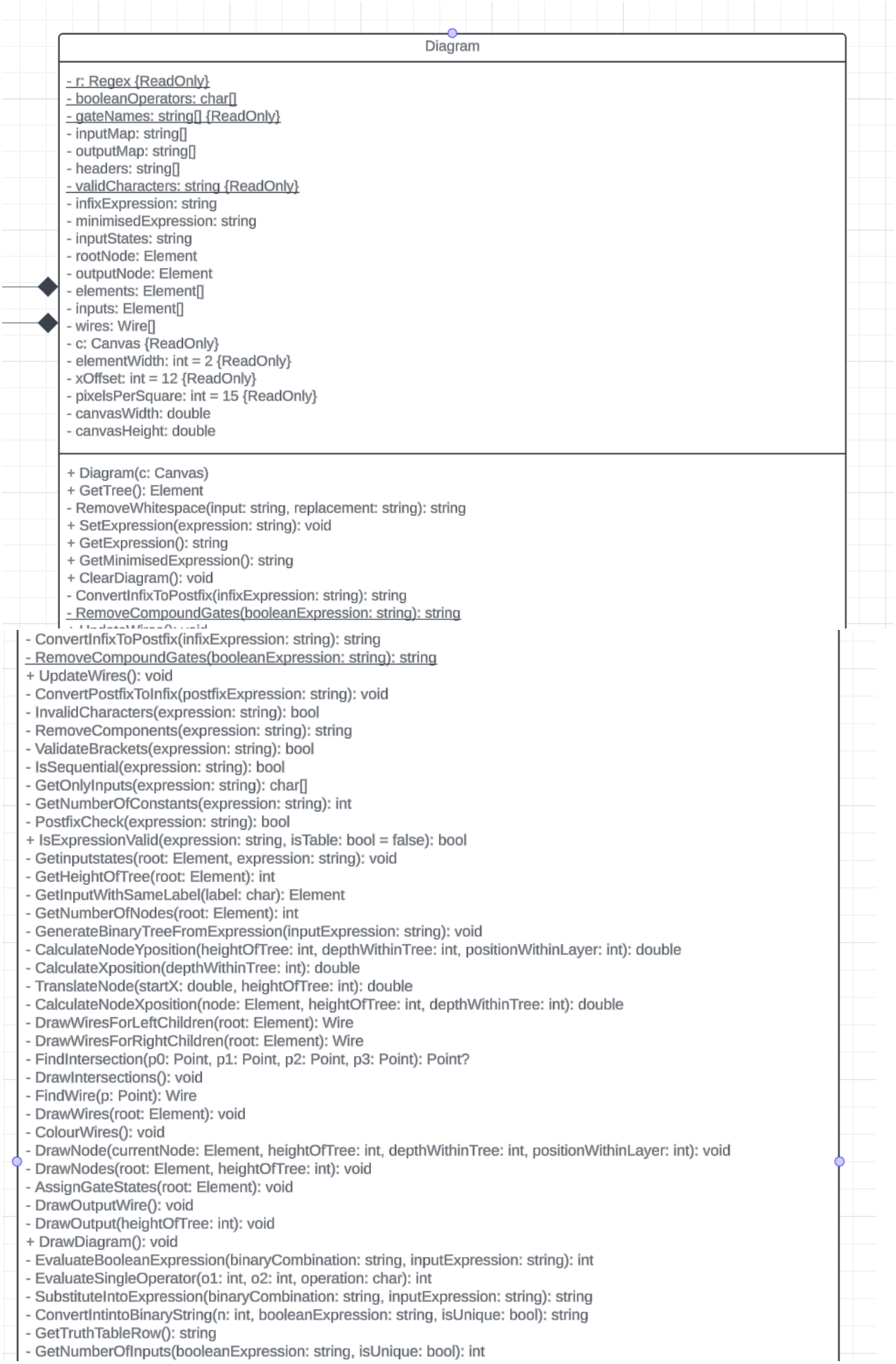


Diagram
- r: Regex {ReadOnly}
- booleanOperators: char[]
- gateNames: string[] {ReadOnly}
- inputMap: string[]
- outputMap: string[]
- headers: string[]
- validCharacters: string {ReadOnly}
- infixExpression: string
- minimisedExpression: string
- inputStates: string
- rootNode: Element
- outputNode: Element
- elements: Element[]
- inputs: Element[]
- wires: Wire[]
- c: Canvas {ReadOnly}
- elementWidth: int = 2 {ReadOnly}
- xOffset: int = 12 {ReadOnly}
- pixelsPerSquare: int = 15 {ReadOnly}
- canvasWidth: double
- canvasHeight: double
+ Diagram(c: Canvas)
+ GetTree(): Element
- RemoveWhitespace(input: string, replacement: string): string
+ SetExpression(expression: string): void
+ GetExpression(): string
+ GetMinimisedExpression(): string
+ ClearDiagram(): void
- ConvertInfixToPostfix(infixExpression: string): string
- RemoveCompoundGates(booleanExpression: string): string
+ UpdateWires(): void
- ConvertPostfixToInfix(postfixExpression: string): void
- InvalidCharacters(expression: string): bool
- RemoveComponents(expression: string): string
- ValidateBrackets(expression: string): bool
- IsSequential(expression: string): bool
- GetOnlyInputs(expression: string): char[]
- GetNumberOfConstants(expression: string): int
- PostfixCheck(expression: string): bool
+ IsExpressionValid(expression: string, isTable: bool = false): bool
- Getinputstates(root: Element, expression: string): void
- GetHeightOfTree(root: Element): int
- GetInputWithSameLabel(label: char): Element
- GetNumberOfNodes(root: Element): int
- GenerateBinaryTreeFromExpression(inputExpression: string): void
- CalculateNodeYposition(heightOfTree: int, depthWithinTree: int, positionWithinLayer: int): double
- CalculateXposition(depthWithinTree: int): double
- TranslateNode(startX: double, heightOfTree: int): double
- CalculateNodeXposition(node: Element, heightOfTree: int, depthWithinTree: int): double
- DrawWiresForLeftChildren(root: Element): Wire
- DrawWiresForRightChildren(root: Element): Wire
- FindIntersection(p0: Point, p1: Point, p2: Point, p3: Point): Point?
- DrawIntersections(): void
- FindWire(p: Point): Wire
- DrawWires(root: Element): void
- ColourWires(): void
- DrawNode(currentNode: Element, heightOfTree: int, depthWithinTree: int, positionWithinLayer: int): void
- DrawNodes(root: Element, heightOfTree: int): void
- AssignGateStates(root: Element): void
- DrawOutputWire(): void
- DrawOutput(heightOfTree: int): void
+ DrawDiagram(): void
- EvaluateBooleanExpression(binaryCombination: string, inputExpression: string): int
- EvaluateSingleOperator(o1: int, o2: int, operation: char): int
- SubstituteIntoExpression(binaryCombination: string, inputExpression: string): string
- ConvertIntintoBinaryString(n: int, booleanExpression: string, isUnique: bool): string
- GetTruthTableRow(): string
- GetNumberOfInputs(booleanExpression: string, isUnique: bool): int

- GetNumberOfInputs(booleanExpression: string, isUnique: bool): int
- GetNumberOfOperators(booleanExpression: string): int
- GenerateInputMap(inputExpression: string, isUnique: bool): string[]
- GenerateOutputMap(inputExpression: string, headers: string[], isUnique: bool): string[]
- GetOutputRow(headers: string[], inputCombination: string): string
- GetHeaders(inputExpression: string, isDisplay: bool): string[]
- GenerateDisplayOperatorHeaders(inputExpression: string, numberOfInputs: int, numberOfOperators: int): string[]
- GeneratePostOrderHeaders(postfix: string, numberOfInputs: int, numberOfOperators: int): string[]
- CalculateCellWidth(header: string): double
- DrawTruthTableHeaders(c: Canvas, headers: string[]): void
- DrawTruthTableBody(c: Canvas, headers: string[], outputMap: string[]): void
- TrimBrackets(headers: string[]): string[]
+ DrawTruthTable(c: Canvas, inputExpression: string): void
- ConvertEPIsToExpression(essentialPrimeImplicants: List<string>): string
- ConvertImplicantToExpression(epi: string): string
- SetRegexPatterns(regex: Dictionary<string, string>, minterms: List<string>): void
- ConvertImplicantsIntoRegex(regex: Dictionary<string, string>, primeImplicants: List<string>): void
- MergeMinterms(m1: string, m2: string): string
- CheckDashesAlign(m1: string, m2: string): bool
- CheckMintermDifference(m1: string, m2: string): bool
- RemoveDashes(minterm: string): int
- GetMinterms(expression: string): List<string>
- GetFrequencyTable(regex: Dictionary<string, string>, minterms: List<string>): int[]
- GetEssentialPrimeImplicant(regex: Dictionary<string, string>, pos: int): string
- GetEssentialPrimeImplicants(regex: Dictionary<string, string>, minterms: List<string>): List<string>
- GetPrimeImplicants(mintermList: List<string>): List<string>
+ MinimiseExpression(expression: string): void
- ReplaceDashesFromRegex(PIchart: Dictionary<string, string>): Dictionary<string, string>
- GetCoveredString(epis: List<string>, PIchart: Dictionary<string, string>): string
- RemoveEPIs(PIchart: Dictionary<string, string>, epis: List<string>, minterms: List<string>): Dictionary<string, string>
- TrimMinterm(PIchart: Dictionary<string, string>, pos: int): Dictionary<string, string>
- GetSignifcantBit(epi: string, freq: int[]): int
- DoPetriksMethod(PIchart: Dictionary<string, string>, epis: List<string>, primeImplicants: List<string>, minterms: List<string>): string
- MapTermsToImplicants(primeImplicants: List<string>): Dictionary<string, string>
- GetProductOfSums(termToImplicantMap: Dictionary<char, string>, primeImplicantChart: Dictionary<string, string>): List<Bracket>

- MapTermsToImplicants(primeImplicants: List<string>): Dictionary<string, string>
- GetProductOfSums(termToImplicantMap: Dictionary<char, string>, primeImplicantChart: Dictionary<string, string>): List<Bracket>
- AddSumsToList(product: List<Bracket>, sumsToAdd: List<Bracket>): void
- GetSumsToAdd(PIchart: Dictionary<string, string>, termToImplicantMap: Dictionary<char, string>, key: string, positionWithinKey: int): List<Bracket>
- GetTermFromImplicant(termToImplicantMap: Dictionary<char, string>, implicant: string): char
- GetSumOfProducts(productOfSums: List<Bracket>): string[]
- MergeBrackets(b1: Bracket, b2: Bracket): Bracket
- ConvertBracketsToString(brackets: List<Bracket>): List<List<string>>
- RecursiveDistributiveLaw(brackets: List<List<string>>): List<List<string>>
- SingleDistributiveLaw(b1: List<string>, b2: List<string>): List<string>
- ApplyDistributiveLaw(a: string, b: string): string
- GetMinProduct(sumOfProducts: string[]): string
- GetFinalExpression(termToImplicantMap: Dictionary<char, string>, minProduct: string): string

Notable Code

Diagram creation

Converting infix Boolean expressions into postfix

To convert an infix Boolean expression into postfix the 'Shunting yard' algorithm can be used. The code for this is shown below:

```
        /// <summary>
        /// Converts a user entered infix boolean expression to the postfix representation
        /// of the boolean expression. This is a modified version of the 'Shunting yard' as
        /// given by the pseudo-code on Wikipedia.
        /// </summary>
        /// <param name="infixExpression">An infix boolean expression. </param>
        /// <returns>The postfix boolean expression of the supplied infix expression.</returns>
        private string ConvertInfixtoPostfix(string infixExpression)
        {
            infixExpression = RemoveWhitespace(infixExpression, "");
            var operatorStack = new Stack<char>();
            string postfixExpression = "";
            int operatorPrecedence;
            foreach (char token in infixExpression)
            {
                if (char.IsLetter(token) || char.IsNumber(token))
                {
                    postfixExpression += token;
                }
                else if (booleanOperators.Contains(token))
                {
                    operatorPrecedence = Array.IndexOf(booleanOperators, token);
                    while (operatorStack.Count > 0 && operatorStack.Peek() != '(' &&
Array.IndexOf(booleanOperators, operatorStack.Peek()) > operatorPrecedence)
                    {
                        postfixExpression += operatorStack.Pop();
                    }
                    operatorStack.Push(token);
                }
                else if (token == '(')
                {
                    operatorStack.Push(token);
                }
                else if (token == ')')
                {
                    while (operatorStack.Peek() != '(')
                    {
                        Debug.Assert(operatorStack.Count > 0, "The stack is empty.");
                        postfixExpression += operatorStack.Pop();
                    }
                    Debug.Assert(operatorStack.Peek() == '(', "The top item is a (");
                    operatorStack.Pop();
                }
            }
            while (operatorStack.Count > 0)
            {
                Debug.Assert(operatorStack.Peek() != '(', "The top item is a (");
                postfixExpression += operatorStack.Pop();
            }
            return postfixExpression;
        }
```

Getting the input states of the logic gate diagram.

To colour the wires, the current states of the input pins on the canvas need to be used. To get these states, an iterative post-order traversal of the binary tree can be used. This is described below:

```
        /// <summary>
        /// Iterative postorder traverasl of the binary tree representing the logic diagram.
        /// The traversal does not consider whether or not an input is visible or not.
        /// It simply produces a string of the input states.
        /// </summary>
        /// <param name="root"> The root of the tree. The last gate/input and also
        /// the point where the traversal starts. </param>
        private void Getinputstates(Element root)
        {
            var s = new Stack<Element>();
            string visited = "";
            while (true)
            {
                while (root != null)
                {
                    // Push the root onto the stack and traverse to the left child.
                    // Repeat until this cannot be done anymore. Same as traversing to the
                    // left-most gate in the tree.
                    s.Push(root);
                    s.Push(root);
                    root = root.leftChild;
                }
                if (s.Count == 0)
                {
                    return;
                }
                root = s.Pop();
                if (s.Count != 0 && s.Peek() == root)
                {
                    root = root.rightChild;
                }
                else
                {
                    // Get the state if the node the current traversal is on, is an input.
                    if (root.GetElementName() == "input_pin")
                    {
                        inputStates += root.GetState();
                        visited += root.GetLabel();
                        root = null;
                    }
                    else
                    {
                        //Used to traverse to the right child.
                        root = null;
                    }
                }
            }
        }
```

Creating the binary tree from the postfix Boolean expression

The process of creating the binary tree from the postfix expression is the inverse of the 'Shunting yard' algorithm. The tree is constructed by tokenising the expression and then building the tree by evaluating the postfix. This algorithm is very important not only due to the job that it performs but also because the same logic can be used in many different applications, such as evaluating Boolean expressions. The algorithm to construct the tree is shown below:

```
        /// <summary>
        /// Converts the user-entered and validated infix boolean expression and represents
        /// it as a binary tree. This is the same logic as the evaluated where in postfix
        /// the first operands are the deepest within the tree.
        /// </summary>
        /// <param name="inputExpression">A user-entered infix boolean expression that will
        /// be represented as the binary tree. This is for drawing the logic diagram.</param>
        private void GenerateBinaryTreeFromExpression(string inputExpression)
        {
            string postfixExpression = ConvertInfixtoPostfix(inputExpression);
            inputs = new Element[GetNumberOfInputs(inputExpression, false)];
            var nodeStack = new Stack<Element>();
            elements = new Element[inputExpression.Length + 1];
            Element nodeToAdd;
            Element leftChild;
            Element rightChild;
            Element tmp;
            int elementID = 0;
            int i = 0;
            string elementName;
            string inputsAdded = "";
            foreach (char c in postfixExpression)
            {
                if (char.IsLetter(c) && char.IsUpper(c) || char.IsNumber(c))
                {
                    nodeToAdd = new Element(elementID, c);
                    inputs[i] = nodeToAdd;
                    i++;
                    if (char.IsNumber(c))
                    {
                        nodeToAdd.SetState(c - 48);
                    }
                    if (inputsAdded.Contains(c) == false)
                    {
                        nodeToAdd.SetInstances(1);
                        inputsAdded += c;
                    }
                    else
                    {
                        tmp = inputs[c - 65];
                        tmp.AddInstance();
                    }
                }
                else if (c == '!')
                {
                    rightChild = nodeStack.Pop();
                    nodeToAdd = new Element("not_gate", elementID, null, rightChild);
                    nodeToAdd.SetInstances(1);
                    rightChild.parent = nodeToAdd;
                }
                else
                {
                    // Any logic gate is a binary operator, so pop two items and these are
                    // the operands for the current boolean operation.
                    rightChild = nodeStack.Pop();
                    leftChild = nodeStack.Pop();
                    elementName = gateNames[Array.IndexOf(booleanOperators, c)];
                    nodeToAdd = new Element(elementName, elementID, leftChild, rightChild);
                    nodeToAdd.SetInstances(1);
                    leftChild.parent = nodeToAdd;
                    rightChild.parent = nodeToAdd;
                }
```

```
                nodeStack.Push(nodeToAdd);
                elements[elementID] = nodeToAdd;
                elementID++;
            }
            rootNode = nodeStack.Pop();
        }
```

Formulae for calculating the position of the nodes in the logic gate diagram

As mentioned previously, there are mathematical formulae to calculate the position of the nodes in the logic gate diagram. These are shown below:

```
        /// <summary>
        /// Calculates the y-position of the logic gate on the canvas given the logic
        /// gate diagram being drawn.
        /// </summary>
        /// <param name="heightOfTree">The height of the tree. This is the deepest node in
        /// the tree. </param>
        /// <param name="depthWithinTree">Integer which stores the current layer that the
        /// node is on. This is also the layer that the BST is on. </param>
        /// <param name="positionWithinLayer">The x-position within the tree at the
        /// depth given by depthWithinTree</param>
        /// <returns>The Y-position of the node on the canvas given its position
        /// within the tree. </returns>
        private double CalculateNodeYposition(int heightOfTree, int depthWithinTree, int
positionWithinLayer)
        {
            // Formula to calculate the spacing for a node in this position within tree
            // in relation to the size of the tree.
            double initialY = Math.Pow(2, heightOfTree) / Math.Pow(2, depthWithinTree) *
pixelsPerSquare;
            // Multiply by the position within the layer to get the Y position of the node.
            // This is the same idea as the X position calculation.
            initialY += initialY * positionWithinLayer * 2;
            // Apply a squish to the Y position so the size of the diagram is reduced.
            // This makes odd-shaped diagrams more easily viewable.
            if (heightOfTree > 5 && depthWithinTree < 4)
            {
                return initialY / 2;
            }
            else
            {
                return initialY;
            }
        }

        /// <summary>
        /// Calculates the X position of a logic gate on the canvas. This is simply
        /// applying a shift factor across the canvas based off of the depth within the tree.
        /// </summary>
        /// <param name="depthWithinTree">The depth within the tree, the current node is.
</param>
        /// <returns>The X position on the canvas which is the location of logic
gate.</returns>
        private double CalculateXposition(int depthWithinTree)
        {
            return canvasWidth - ((pixelsPerSquare - 7) * elementWidth + (pixelsPerSquare - 7)
* xOffset) * depthWithinTree;
        }
```

Finding the location of wire intersections

When wires go to different inputs, a small arc must be drawn at the intersection point to show that they are not connected. This location can be calculated with the algorithm below, where the parameters correspond to two different line segments of two different wires:

```
        /// <summary>
        /// Calculates the point of intersection, if any, given 4 points which
        /// corresponds to 2 lines that are on the canvas. These are always a
        /// vertical and horizontal line as this is the only case where an
        /// intersection would have to be drawn.
        /// </summary>
        /// <param name="p0">The start point of the vertical line.</param>
        /// <param name="p1">The end point of the vertical line.</param>
        /// <param name="p2">The start point of the horizontal line.</param>
        /// <param name="p3">The end point of the horizontal line.</param>
        /// <returns>The point on the canvas where the intersection is located.</returns>
        private static Point? FindIntersection(Point p0, Point p1, Point p2, Point p3)
        {
            double s_x = p1.X - p0.X;
            double s_y = p1.Y - p0.Y;
            double s2_x = p3.X - p2.X;
            double s2_y = p3.Y - p2.Y;
            double denom = s_x * s2_y - s2_x * s_y;

            if (denom == 0)
            {
                // The line supplied are collinear.
                return null;
            }
            bool isDenomPositive = denom > 0;

            double s3_x = p0.X - p2.X;
            double s3_y = p0.Y - p2.Y;
            double s_numer = s_x * s3_y - s_y * s3_x;

            if (s_numer < 0 == isDenomPositive)
            {
                // There is no intersection between the two lines.
                return null;
            }

            double t_numer = s2_x * s3_y - s_x * s3_y;

            if (t_numer < 0 == isDenomPositive)
            {
                // There is no intersection between the two lines.
                return null;
            }

            if (s_numer > denom == isDenomPositive || t_numer > denom == isDenomPositive)
            {
                // There is no intersection between the lines.
                return null;
            }

            // There must be an intersection between the two lines.
            double t = t_numer / denom;
            // Finding the position and returning the point of intersection that has been
            // found between the two lines.
            Point? result = new Point(p0.X + t * s_x, p0.Y + t * s_y);
            return result;
```

```
        }
```

Drawing the nodes and wires onto the canvas.

The process of drawing the nodes onto the canvas is very similar to the process of drawing the wires. They can be both be done using a breadth first traversal. The traversal shown below is used to draw the nodes onto the canvas. This is slightly modified to keep track of the horizontal position within the binary tree.

```
        /// <summary>
        /// A modified breadth first traversal that is responsible for placing all of the
        /// logic gates onto the diagram canvas. The modification is that the traversal
        /// also tracks the horizontal position within a given depth of the binary tree.
        /// This allows the y-position to be calculated.
        /// </summary>
        /// <param name="root">The root node of the binary tree. This is the last logic
        /// gate.</param>
        /// <param name="heightOfTree">The height of the tree given. This is the maximum
        /// depth of the tree. </param>
        private void DrawNodes(Element root, int heightOfTree)
        {
            var q = new Queue<Element>();
            q.Enqueue(root);
            int depthWithinTree = 0;
            // The horizontal position within a given depth of the tree.
            int positionWithinLayer = 0;
            int sizeOfQ;
            Element currentNode;
            while (q.Count != 0)
            {
                sizeOfQ = q.Count;
                // Travelling along a particular layer of the binary tree.
                while (sizeOfQ != 0)
                {
                    currentNode = q.Peek();
                    // Draw node at the current position of the breadth first traversal.
                    DrawNode(currentNode, heightOfTree, depthWithinTree, positionWithinLayer);
                    q.Dequeue();
                    // Travelling horizontally along the tree.
                    positionWithinLayer++;
                    // If the left child exists then add it to the queue to be reached.
                    if (currentNode.leftChild != null)
                    {
                        q.Enqueue(currentNode.leftChild);
                    }
                    // If the right child exists then add it to the queue to be reached.
                    if (currentNode.rightChild != null)
                    {
                        q.Enqueue(currentNode.rightChild);
                    }
                    sizeOfQ--;
                }
                // Next level of the tree so increase the depth and reset the position
                // within the layer to the left-most node => 0.
                depthWithinTree++;
                positionWithinLayer = 0;
            }
        }
```

The traversal below is the one used in the program to draw the wires.

```
        /// <summary>
        /// Draws the wires to create the logic gate diagram. The states of the gates has
```

```
        /// not been set so the colour of the wires are at their default state.
        /// </summary>
        /// <param name="root">The root node of the binary tree. </param>
        private void DrawWires(Element root)
        {
            var q = new Queue<Element>();
            // The number of wires is always the number of nodes as every node has
            // a wire that is drawn from the nodes output.
            wires = new Wire[GetNumberOfNodes(root)];
            Element tmp;
            int i = 0;
            // Traversing the nodes using a breadth first traveral to add the wires
            // in that order.
            q.Enqueue(root);
            while (q.Count != 0)
            {
                tmp = q.Dequeue();
                // Traversing the left child if it exists.
                if (tmp.leftChild != null)
                {
                    // Adding the wire created into the array.
                    wires[i] = DrawWiresForLeftChildren(tmp);
                    i++;
                    // Traversing the left child because it exists.
                    q.Enqueue(tmp.leftChild);
                }
                // Traversing the right child if it exists.
                if (tmp.rightChild != null)
                {
                    // Adding the created wire into the array.
                    wires[i] = DrawWiresForRightChildren(tmp);
                    i++;
                    // Traversing the right child because it exists.
                    q.Enqueue(tmp.rightChild);
                }
            }
            // Intersections have not been considered up until this point.
            // Compute all of the intersections of the currently drawn wires.
            DrawIntersections();
            // Adding the wires to the canvas as all of the points have been calculated.
            for (var j = 0; j < wires.Length - 1; j++)
            {
                wires[j].RenderLine();
            }
        }
```

Truth table generation

Evaluating Boolean expressions

The process of evaluating Boolean expressions is the same as constructing the binary tree, as shown previously. The evaluation is done by simply substituting the inputs in and then reading through the expression. This is shown by the code snippet below:

```
        /// <summary>
        /// Evaluates a user-inputted infix boolean expression. This is done by first
        /// converting the expression into postfix and then using a stack.
        /// </summary>
        /// <param name="binaryCombination">A string storing the input combination to be
        /// substituted into the postfix boolean expression. </param>
        /// <param name="inputExpression">The infix expression being evaluated. </param>
        /// <returns>The result of evaluating an infix boolean expression with a particular
```

```
        /// string of inputs. </returns>
        private int EvaluateBooleanExpression(string binaryCombination, string inputExpression)
        {
            string postfix = ConvertInfixtoPostfix(inputExpression);
            int operand1;
            int operand2;
            int tmp;
            string substitutedExpression = SubsituteIntoExpression(binaryCombination, postfix);
            var evaluatedStack = new Stack<int>();
            foreach (char c in substitutedExpression)
            {
                // An operand has been found.
                if (char.IsNumber(c))
                {
                    evaluatedStack.Push(c);
                }
                else if (c == '!')
                {
                    // NOT is an unary operator so only pop one item off of the stack
                    operand1 = evaluatedStack.Pop();
                    // Applying logical is the same as applying an XOR of 1 as the
                    // operand is only every one bit in length.
                    evaluatedStack.Push(operand1 ^ 1);
                }
                else if (c == '^')
                {
                    // Pop two items for the XOR as it is a binary operator.
                    operand1 = evaluatedStack.Pop();
                    operand2 = evaluatedStack.Pop();
                    // Carry out the operation and push the result to the stack.
                    tmp = EvaluateSingleOperator(operand1, operand2, c);
                    evaluatedStack.Push(tmp + 48);
                }
                else
                {
                    // Pop two items for the gate as it is a binary operator.
                    operand1 = evaluatedStack.Pop();
                    operand2 = evaluatedStack.Pop();
                    // Carry out the operation and push the result to the stack.
                    tmp = EvaluateSingleOperator(operand1, operand2, c);
                    evaluatedStack.Push(tmp);
                }
            }
            // The final item on the stack is the result of the evaluation.
            return evaluatedStack.Pop();
        }
```

Generating the data for a truth table

The truth table is simply a map of all of the inputs and outputs of a particular Boolean expression. The table also shows the steps when evaluating the expression. These can be created by evaluating the sub-expression headers of the Boolean expression. To create the table data, the following code can be used:

```
        /// <summary>
        /// Computes all of the input combinations for a truth table. These can be used to
        /// computer the rest of the binary values in the truth table.
        /// </summary>
        /// <param name="inputExpression">The boolean expression being tabulated. </param>
        /// <param name="isUnique">Whether or not the map is for a table with unique input
        /// headers. </param>
        /// <returns>All possible binary input combinations given a boolean
expression.</returns>
```

```
        private static string[] GenerateInputMap(string inputExpression, bool isUnique)
        {
            int numberOfInputs = GetNumberOfInputs(inputExpression, isUnique);
            // There is always 2^n different input combinations where n is the number of
            // unique inputs.
            int numberOfRows = (int)Math.Pow(2, numberOfInputs);
            string[] inputMap = new string[numberOfRows];
            for (var i = 0; i < numberOfRows; i++)
            {
                // The input combination is simply the row of the table converted into
                // its respective binary string.
                inputMap[i] = ConvertIntintoBinaryString(i, inputExpression, isUnique);
            }
            return inputMap;
        }

        /// <summary>
        /// Computes the complete set of values for a truth table. This is done by
        /// </summary>
        /// <param name="inputExpression">The boolean expression being tabulated. </param>
        /// <param name="headers">The headers of the truth table. </param>
        /// <param name="isUnique">Whether the table has unique inputs or not. This
        /// decides whether or not the table is being drawn or used for diagram interactivity.
        /// </param>
        /// <returns>The complete table of values for the truth table. </returns>
        private string[] GenerateOutputMap(string inputExpression, string[] headers, bool
isUnique)
        {
            // Generating all of the different input combinations for the truth table.
            // These will be substituted into the boolean expression.
            string[] inputMap = GenerateInputMap(inputExpression, isUnique);
            int numberOfRows = inputMap.Length;
            string inputCombination;
            string[] outputMap = new string[inputMap.Length];
            for (var i = 0; i < numberOfRows; i++)
            {
                // Getting the input combination for the truth table.
                inputCombination = inputMap[i];
                // Computing the row of the truth table.
                outputMap[i] += GetOutputRow(headers, inputCombination);
            }
            return outputMap;
        }
```

Generating headers for the truth table

The headers of a truth table can be created with the same method as evaluating Boolean expressions. This is shown by the algorithm below:

```
        /// <summary>
        /// Generates headers for colouring the wires when the user clicks on the inputs.
        /// These represent the subexpression at a certain point within the tree. This is
        /// also, the postorder traversal of the tree. This is done by evaluating the
expression.
        /// </summary>
        /// <param name="postfix">The postfix expression that the header will
represent.</param>
        /// <param name="numberOfInputs">The number of non-unique inputs within the given
boolean expression. </param>
        /// <param name="numberOfOperators">The number of operators within the given boolean
expression. </param>
```

```
        /// <returns>An array representing the postorder headers of a postfix boolean
expression. </returns>
        private static string[] GeneratePostOrderHeaders(string postfix, int numberOfInputs,
int numberOfOperators)
        {
            var subExpressionStack = new Stack<string>();
            // The number of headers is always the number of non-unique inputs + the number
            // of operators as this is the number of nodes in the tree.
            string[] headers = new string[numberOfOperators + numberOfInputs];
            string subexpression;
            string operand1;
            string operand2;
            int i = 0;
            // Tokenising the expression.
            foreach (char c in postfix)
            {
                // When an operand is found then pop it and add it to the header array as
                // An input is a header in itself.
                if (char.IsLetter(c) || char.IsNumber(c))
                {
                    subExpressionStack.Push(c.ToString());
                    headers[i] = c.ToString();
                    i++;
                }
                // An operator has been found.
                else
                {
                    if (c == '!')
                    {
                        // As a NOT gate is a unary operator, pop one item off of the stack.
                        operand1 = subExpressionStack.Pop();
                        subexpression = $"({c}{operand1})";
                    }
                    else
                    {
                        // All other operators are binary operators so pop two items off of
                        // the stack and push the result pack onto the stack.
                        operand1 = subExpressionStack.Pop();
                        operand2 = subExpressionStack.Pop();
                        // Add brackets for the sub-expression and for the next headers.
                        subexpression = $"({operand2}{c}{operand1})";
                    }
                    // Pushing the result pack onto the stack and add to the headers array
                    // as a subexpression is a header in itself.
                    subExpressionStack.Push(subexpression);
                    headers[i] = subexpression;
                    i++;
                }
            }
            return headers;
        }
```

Minimisation

Finding the prime implicants

Within the Quine-McCluskey Algorithm the first step is finding the prime implicants of the Boolean expression. These are terms that cannot be merged with any other implicant if the following conditions are not met:

- All dashes within the implicant must align.
- Only one bit can differ between the two minterms being merged.

The prime implicants are used to create the prime implicant which allows the simplification of the expression to be carried out. The algorithm is described by the code below:

```
        /// <summary>
        /// Finds the prime implicants of the boolean expression. This comes from the minterms
        /// of the boolean expression. This is done using a recursive merging process where
        /// merges take place. Once no merges have taken place on the prim implicants then they
        /// all have been found and the method ends.
        /// </summary>
        /// <param name="mintermList">The binary input combinations that result in the
        /// boolean expression evaluating to one. </param>
        /// <returns>The list of prime implicants of the expression. </returns>
        private static List<string> GetPrimeImplicants(List<string> mintermList)
        {
            var primeImplicants = new List<string>();
            bool[] merges = new bool[mintermList.Count];
            int numberOfMerges = 0;
            string mergedMinterm;
            string minterm1;
            string minterm2;
            for (var i = 0; i < mintermList.Count; i++)
            {
                for (var c = i + 1; c < mintermList.Count; c++)
                {
                    minterm1 = mintermList[i];
                    minterm2 = mintermList[c];
                    // A merge can be made only if the dashes of the minterms align and
                    // there is only one bit of difference between the two minterms.
                    if (CheckDashesAlign(minterm1, minterm2) &&
CheckMintermDifference(minterm1, minterm2))
                    {
                        mergedMinterm = MergeMinterms(minterm1, minterm2);
                        primeImplicants.Add(mergedMinterm);
                        numberOfMerges++;
                        // Mark the terms that have been merged so that they do not persist
                        // to the next stage of the merging process.
                        merges[i] = true;
                        merges[c] = true;
                    }
                }
            }
            // Filtering out the terms that have been merged so that the minterms do not
            // remain when the next stage occurs.
            for (var j = 0; j < mintermList.Count; j++)
            {
                if (!merges[j] && !primeImplicants.Contains(mintermList[j]))
                {
                    primeImplicants.Add(mintermList[j]);
                }
            }
            // If no more merges can be made on the list of implicants then all of the prime
            // implicants for the expression must have been found. Otherwise, recurse and try
            // merging again.
            if (numberOfMerges == 0)
            {
                return primeImplicants;
            }
            else
            {
                return GetPrimeImplicants(primeImplicants);
            }
```

```
        }
```

Checking only one bit differs between two binary numbers

As mentioned above, to merge two minterms, only one bit can differ between the two binary numbers. This can be checked with the following function.

```
        /// <summary>
        /// A merge can only take place if only one bit differs between the two minterms.
        /// </summary>
        private static bool CheckMintermDifference(string m1, string m2)
        {
            // Removing the dashes so that bitwise operators can be used.
            int minterm1 = RemoveDashes(m1);
            int minterm2 = RemoveDashes(m2);
            // XOR identifies any bits that differ between the two minterms.
            int res = minterm1 ^ minterm2;
            // If res != 0, then one bit could differ. This checked with the AND.
            return res != 0 && (res & res - 1) == 0;
        }
```

Creating the minterm coverage

To simplify the Boolean expression, the essential prime implicants must be found. This is a prime implicant that is the only one to cover a specific minterm. The coverage of a prime implicant can be found using regex where the implicant is the pattern and the binary representation of the minterm is the minterm being tested. This is because the prime implicants are a general form to describe a series of binary numbers. This is done using the algorithm below:

```
/// <summary>
/// Creates the minterm coverage strings for the prime implicants. These string show
/// which prime implicants cover which minterms. This is used in finding the essential
/// prime implicants and also, they are used in Petricks method for finding the product of
sums.
/// </summary>
/// <param name="regex">The prime implicant chart being filled.</param>
/// <param name="minterms">The minterms for the user entered boolean expression.</param>
private static void SetRegexPatterns(Dictionary<string, string> regex, List<string> minterms)
{
    Match res;
    foreach (string regexPattern in regex.Keys.ToList())
    {
        foreach (string minterm in minterms)
        {
            // If minterm matches the form of the prime implicants, shown by Regex,
            // then a one can be written to the coverage string showing that the
            // implicant covers this minterm.
            res = Regex.Match(minterm, regexPattern);
            if (res.Success)
            {
                regex[regexPattern] += "1";
            }
            //The implicant does not cover this minterm.
            else
            {
                regex[regexPattern] += "0";
            }
        }
    }
}
```

Getting the product of sums

A product of sums is always of the form (A+B)(C+D) within Boolean algebra. A sum can be formed if two implicants cover the same minterm. This is shown by the example below, from Wikipedia:

	0	1	2	5	6	7	⇒	A	B	C
K = m(0,1)	✓	✓					⇒	0	0	—
L = m(0,2)	✓		✓				⇒	0	—	0
M = m(1,5)		✓		✓			⇒	—	0	1
N = m(2,6)			✓		✓		⇒	—	1	0
P = m(5,7)				✓		✓	⇒	1	—	1
Q = m(6,7)					✓	✓	⇒	1	1	—

From the prime-implicant chart above, the product of sums is given as:

$$(K+L)(K+M)(L+N)(M+P)(N+Q)(P+Q)$$

This can be computed by simply iterating through the prime-implicant chart, looking for any non-duplicate sums that could be made. Each sum is unique as $X+X=X$ within Boolean algebra. The code below finds the product of sums:

```
        /// <summary>
        /// Produces the product of sums of the prime implicant chart. This stage prepares
        /// the data so that the boolean algebra can be done on the expression.
        /// </summary>
        /// <param name="termToImplicantMap">The relationship between terms and
        /// prime implicants. </param>
        /// <param name="primeImplicantChart">The relationship between minterm coverage
        /// and the prime implicants found in the intial process of QM. </param>
        /// <returns> The product of sums of the PI chart, in the form [('K', 'L'),
etc.]</returns>
        private static List<Bracket> GetProductOfSums(Dictionary<char, string>
termToImplicantMap, Dictionary<string, string> primeImplicantChart)
        {
            var productOfSums = new List<Bracket>();
            List<Bracket> sumsToAdd;
            string primeImplicant;
            // A sum can be made if the minterm is covered by two implicants.
            // So, loop through each key to find its coverage of the minterms
            foreach (string key in primeImplicantChart.Keys)
            {
                primeImplicant = primeImplicantChart[key];
                // Iterate through each minterm.
                for (var i = 0; i < primeImplicant.Length; i++)
                {
                    // If the prime implicant covers the minterm then a possible sum
                    // could be found so search through the chart vertically.
                    if (primeImplicant[i] == '1')
                    {
                        // Get all of the possible sums within the column of the covered
                        // minterm and add them to the found sums.
                        sumsToAdd = GetSumsToAdd(primeImplicantChart, termToImplicantMap, key,
i);
                        AddSumsToList(productOfSums, sumsToAdd);
                    }
                }
```

```
            }
            return productOfSums;
        }
        /// <summary>
        /// Gets all of the possible sums that are within a column of the prime implicant
        /// chart. This is done by searching through and if another implicant covers the
        /// same minterm then a sum can be made.
        /// </summary>
        /// <param name="PIchart">The relationship between the prime implicants and
        /// the minterms that they cover. </param>
        /// <param name="termToImplicantMap">The relationship between terms in the boolean
        /// algebra and the prime implicants that they represent.</param>
        /// <param name="key">The prime implicant that caused the column search</param>
        /// <param name="positionWithinKey">The minterm that the prime implicant must
        /// cover in order for a sum to be created. It must not be the same implicant. </param>
        /// <returns>All of the possible sums for a prime implicant that covers a minterm.
</returns>
        private static List<Bracket> GetSumsToAdd(Dictionary<string, string> PIchart,
Dictionary<char, string> termToImplicantMap, string key, int positionWithinKey)
        {
            var sumsToAdd = new List<Bracket>();
            Bracket sum;
            string implicant;
            char term1;
            char term2;
            // Iterate through each prime implicant to try and find as many sums as possible.
            for (var i = 0; i < PIchart.Keys.Count; i++)
            {
                // The prime implicant that could make a sum.
                implicant = PIchart.Keys.ToArray()[i];
                // If the implicant covers the same minterm then a sum has been found and a
                // bracket can be created.
                if (PIchart[implicant][positionWithinKey] == '1')
                {
                    // Getting the letters that represent a certain prime implicant.
                    term1 = GetTermFromImplicant(termToImplicantMap, key);
                    term2 = GetTermFromImplicant(termToImplicantMap, implicant);
                    // Ensuring brackets are not made with duplicate term.
                    if (term1 != term2)
                    {
                        // A new sum is created and added to the list.
                        sum = new Bracket(term1, term2);
                        sumsToAdd.Add(sum);
                    }
                }
            }
            return sumsToAdd;
        }
```

Getting the sum of products

The sum of products produces from the product of sums on the prime implicant chart, is what is used to find the minimised Boolean expression. This is because each product is a potential solution to the minimized Boolean expression. To find the sum of products, an initial merge is applied to the sums as these always follow the same form, after the distributive law is applied to further expand the product of sums.

```
        /// <summary>
        /// Converts the product of sums (A+B)(C+D)... into the sum of products of the Boolean
        /// expression. sum of products = AB + CD + etc. This allows the program to find the
        /// minimal term(solution) which can ultimately give the minimised expression.
        /// </summary>
```

```
        /// <param name="productOfSums">The product of sums found in the prime implicant
        /// chart. </param>
        /// <returns> The sum of products of the expression which is used to minimise the
input</returns>
        private static string[] GetSumOfProducts(List<Bracket> productOfSums)
        {
            Bracket b1;
            Bracket b2;
            Bracket mergedTerm;
            bool merged = true;
            // Keep trying to merge the brackets together until no more merges can be made.
            // This means that the distributive law can be applied to the brackets at that
point.
            while (merged)
            {
                merged = false;
                // Checking each sum to ensure the most merges take place.
                for (var i = 0; i < productOfSums.Count - 1; i++)
                {
                    // Start c at i + 1 because the order of brackets does not matter, so
                    // simply search through the remaining brackets.
                    for (var c = i + 1; c < productOfSums.Count; c++)
                    {
                        b1 = productOfSums[i];
                        b2 = productOfSums[c];
                        // Ensuring that the brackets are not the same. Must also consider if
                        // the terms are the same but in the opposite order.
                        if (b1.term1 == b2.term1 != (b1.term2 == b2.term2) != (b1.term1 ==
b2.term2) != (b1.term2 == b2.term1))
                        {
                            // A make the merge as it is a valid one and remove the brackets
                            // that result in the merge. This is to stop the same merges
                            // occurring again and again.
                            mergedTerm = MergeBrackets(b1, b2);
                            productOfSums.Add(mergedTerm);
                            productOfSums.Remove(b1);
                            productOfSums.Remove(b2);
                            merged = true;
                            i = c + 1;
                            c = i + 1;
                        }
                    }
                }
            }
            // Convert the merged product of sums into a string, this allows each bracket
            // to have more than two products within it.
            List<List<string>> stringProducts = ConvertBracketsToString(productOfSums);
            // Recursively apply the distributive law which does the rest of the work.
            List<List<string>> sumOfProducts = RecursiveDistributiveLaw(stringProducts);
            // The first term within the 2D list is the only bracket that remains and hence
            // can no longer be merged. It also contains all of the solutions to the
minimisation.
            return sumOfProducts[0].ToArray();
        }
```

Recursive distributive law application

When following Petricks' method, after finding the product of sums, Boolean algebra and namely the distributive law must be used to expand the brackets. The following code recursively applies the distributive law to the product of sums.

```
        /// <summary>
```

```
        /// Carries out the main distribution to convert the product of sums into the sum
        /// of products. This recursively applies the distributive law until only one
        /// bracket remains which means the law can no longer be applied.
        /// </summary>
        /// <param name="brackets">The string representation of the product of sums.</param>
        /// <returns>A singular string representing the sum of products. </returns>
        private static List<List<string>> RecursiveDistributiveLaw(List<List<string>> brackets)
        {
            var lls = new List<List<string>>();
            // The distributive law can be applied as long as there are at least 2 brackets
            // remaining in the expression.
            if (brackets.Count > 1)
            {
                // Applies the distributive law on two brackets that contain n and m terms
                // each and adds the result to the list of remaining brackets.
                lls.Add(SingleDistributiveLaw(brackets[0], brackets[1]));
                // The brackets used within the distributive law do not persist.
                // This means that they should be removed.
                brackets.RemoveAt(0);
                brackets.RemoveAt(0);
                lls.AddRange(brackets);
                // More than 2 brackets remain and so repeat the process.
                return RecursiveDistributiveLaw(lls);
            }
            else
            {
                // Distributive law can no longer be applied and so return the complete
                // sum of products.
                return brackets;
            }
        }

        /// <summary>
        /// Applies the distributive law on two brackets that store m and n terms
        /// respectively. This is because (KN+KLQ+LMN+LMQ)(P+MQ) can be further
        /// distributed.
        /// </summary>
        /// <param name="b1">A bracket storing m number of terms. </param>
        /// <param name="b2">A bracket storing n number of terms that merges into b1.</param>
        /// <returns></returns>
        private static List<string> SingleDistributiveLaw(List<string> b1, List<string> b2)
        {
            var lls = new List<string>();
            // Apply the law to every term within both of the brackets to find the complete
            // expansion of the two brackets.
            for (var i = 0; i < b1.Count; i++)
            {
                for (var j = 0; j < b2.Count; j++)
                {
                    // Applying the distributive law to two products. e.g. KLQ and P
                    lls.Add(ApplyDistributiveLaw(b1[i], b2[j]));
                }
            }
            return lls;
        }
```

File Structure

The following file structure of the program is given by the table of contents below.

Full code listing

It should be noted that the order of the code listing below is given by the file structure described above.

Code

BooleanConverter.cs

The renderer within the program uses LATEX to show the expression. The following code is a converter for converting the custom format of the user-entered Boolean expressions into their LATEX representation, this is done by converting it into postfix and then “evaluating” the expression to build the LATEX string.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Linq.Expressions;
using System.Runtime.ExceptionServices;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;
using System.Windows.Data;

namespace _2BNOR_2B.Code
{
    public class BooleanConverter : IValueConverter
    {
        // All of the Boolean operators that a user could enter into a Boolean expression.
        private readonly char[] booleanOperators = { '+', '^', '.', '!' };
        // Regular expression used to remove the whitespace in a string.
        private static readonly Regex r = new(@"\s+");

        /// <summary>
        /// Utility function for removing the whitespace from user-entered Boolean expressions.
        /// This makes them easier to deal with and removes an element to validate, overall
```

```
        /// simplifying the algorithm.
        /// </summary>
        /// <param name="input">The string being modified by the method.</param>
        /// <param name="replacement">The character that will replace the white space in the
        /// input, value = "", to remove the whitespace completely.</param>
        /// <returns></returns>
        private static string RemoveWhitespace(string input, string replacement)
        {
            return r.Replace(input, replacement);
        }

        /// <summary>
        /// Converts a user entered infix Boolean expression to the postfix representation
        /// of the Boolean expression. This is a modified version of the 'Shunting yard' as
        /// given by the pseudo-code on Wikipedia.
        /// </summary>
        /// <param name="infixExpression">An infix Boolean expression. </param>
        /// <returns>The postfix Boolean expression of the supplied infix expression.</returns>
        private string ConvertInfixtoPostfix(string infixExpression)
        {
            // Removing whitespace to simplify processing.
            infixExpression = RemoveWhitespace(infixExpression, "");
            var operatorStack = new Stack<char>();
            string postfixExpression = "";
            int operatorPrecedence;
            // Tokenising the expression.
            foreach (char token in infixExpression)
            {
                // If the token is an input/constant then it can be added straight into the
output.
                if (char.IsLetter(token) || char.IsNumber(token))
                {
                    postfixExpression += token;
                }
                else if (booleanOperators.Contains(token))
                {
                    operatorPrecedence = Array.IndexOf(booleanOperators, token);
                    while (operatorStack.Count > 0 && operatorStack.Peek() != '(' &&
Array.IndexOf(booleanOperators, operatorStack.Peek()) > operatorPrecedence)
                    {
                        postfixExpression += operatorStack.Pop();
                    }
                    operatorStack.Push(token);
                }
                else if (token == '(')
                {
                    operatorStack.Push(token);
                }
                else if (token == ')')
                {
                    while (operatorStack.Peek() != '(')
                    {
                        postfixExpression += operatorStack.Pop();
                    }
                    operatorStack.Pop();
                }
            }
            while (operatorStack.Count > 0)
            {
                postfixExpression += operatorStack.Pop();
            }
            return postfixExpression;
```

```
        }

        /// <summary>
        /// Utility function to convert a given character into its latex representation.
        /// </summary>
        /// <param name="c">The character being converted.</param>
        /// <returns>The latex representation of the suplied character.</returns>
        private static string GetLATEXFromChar(char c)
        {
            switch (c)
            {
                case '!':
                    return @"\overline{";
                case '.':
                    // Added spaces in the latex as a stylistic choice.
                    return @"\;.\;";
                case '^':
                    return @"\oplus ";
                default:
                    return c.ToString();
            }
        }

        /// <summary>
        /// Converts the postfix string by evaluating it and building the string, that way.
        /// However, this differs by instead of popping the program's notation onto the stack,
        /// the LATEX representation of the string is pushed back onto the stack.
        /// </summary>
        /// <param name="inputString"></param>
        /// <returns></returns>
        public string ConvertString(string inputString)
        {
            var subExpressionStack = new Stack<string>();
            string subexpression;
            // Creating the postfix expression.
            string postfix = ConvertInfixtoPostfix(inputString);
            string operand1;
            string operand2;
            foreach (char c in postfix)
            {
                if (char.IsLetter(c) || char.IsNumber(c))
                {
                    subExpressionStack.Push(c.ToString());
                }
                else
                {
                    if (c == '!')
                    {
                        // As NOT is an unary operator, then only pop one item from the stack.
                        operand1 = subExpressionStack.Pop();
                        // Push the result pack onto the stack in LATEX form.
                        subexpression = $"{GetLATEXFromChar(c)}{operand1}" + "}";
                    }
                    else
                    {
                        // Any other operator is a binary operator so pop two items from the
                        // stack.
                        operand1 = subExpressionStack.Pop();
                        operand2 = subExpressionStack.Pop();
                        // Adding brackets to show precedence and make the resulting expression
                        // clearer.
```

```
                        subexpression =
$@"\left({operand2}{GetLATEXFromChar(c)}{operand1}\right)";
                    }
                    // Push the result back onto the stack.
                    subExpressionStack.Push(subexpression);
                }
            }
            // The LATEX expression is the final result on the stack.
            return subExpressionStack.Pop();
        }

        /// <summary>
        /// Method to convert the expression the user is entering at runtime into its
        /// LATEX representation. This is done by converting the expression into postfix
        /// and then building the string by "evaluating" it.
        /// </summary>
        /// <param name="value">The string value from the textbox that is being
converted.</param>
        /// <returns>LATEX representation of the user-inputted Boolean expression.</returns>
        public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
        {
            // Try converting the string into its LATEX representation otherwise display
            // nothing as the user has not entered a decent expression.
            try
            {
                return ConvertString(value as string);
            }
            catch
            {
                return "";
            }
        }

        /// <summary>
        /// Function to convert back into the string. However, this function is N/A for the
        /// program, hence, the notImplmentedException(). It should be noted that it is
required
        /// for a IValueConverter to have a ConvertBack method.
        /// </summary>
        /// <returns>A Boolean expression given its LATEX representation.</returns>
        /// <exception cref="NotImplementedException">This method is not needed within the
        /// program so it does not need to be implemented.</exception>
        public object ConvertBack(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
        {
            throw new NotImplementedException();
        }
    }
}
```

Bracket.cs

This is a simple struct that is used within Petrick's method. This is because it abstracts the data involved and makes it easier to work with when performing the Boolean algebra on the product of sums. This is shown below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _2BNOR_2B.Code
```

```
{
    /// <summary>
    /// Represents two product brackets within the product of sums. This simplifies the
    /// process of finding the sum of products by keeping the terms clear and in order.
    /// </summary>
    struct Bracket
    {
        public Bracket(char term1, char term2)
        {
            // Sorting into alphabetical order. This can be done as OR is commutative.
            if (term1 < term2)
            {
                this.term1 = term1.ToString();
                this.term2 = term2.ToString();
            }
            else
            {
                this.term1 = term2.ToString();
                this.term2 = term1.ToString();
            }
        }
        // The two terms that make up the brackets. These are assigned to when the
        // product of sums is created in Petrick's method.
        public string term1;
        public string term2;
    }
}
```

Element.cs

This is the main class used to represent the nodes of the binary tree, which is in turn the logic gate diagram. This class stores the children and parent of itself which how traversals can be carried out on the tree. A unique ID is assigned to each object in order for unique identification of the nodes. This class in itself does not have any important functionality contained within it, but it is very helpful for data encapsulation, and it is vital for the creation of the binary tree. When a diagram is being drawn with repeated inputs, although only one logic gate is added to the canvas, there are two instances of the element class in order to preserve the tree structure, otherwise a graph would need to be used. The class is given below:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace _2BNOR_2B.Code
{
    /// <summary>
    /// Acts as the node for the binary tree in the logic gate diagram. This can represent
    /// input pins, outputs, and logic gates. It is important to note that an element is a
    /// non-visual node that pairs with the visual through the logicGate class. This allows
    /// for repeated inputs to either be displayed or not.
    /// </summary>
    public class Element
    {
        // The ID of the node within the tree. This is also the position within the string
        // produced by a postorder traversal.
        private readonly int elementID;
        public Element leftChild;
        public Element rightChild;
        public Element parent;
        // The visual gate that is paired with the node in the tree. This is null if the
```

```
        // node is repeated.
        private LogicGate logicGate;
        // The binary state of the element (1 or 0), used to colour the wires when the diagram
        // is interacted with.
        private int state;
        // If the node is an input, then it has a respective label which should be displayed.
        // Only ever null when a logic gate as they have no label.
        private readonly char label;
        // The string name of the node "and_gate", etc.
        private readonly string elementName;
        // The number of occurrences within the tree. Such as (A.B)^A => A would have 2
instances.
        private int instances = 0;

        public Element()
        {

        }
        /// <summary>
        /// Constructor for generating the output pin of the logic gate diagram. Simply
        /// sets the default values for it.
        /// </summary>
        /// <param name="elementID">The output pin has an ID of -1 as it is not in the
tree.</param>
        public Element(int elementID)
        {
            this.elementID = elementID;
            state = 0;
            label = 'Q';
            elementName = "output_pin";
        }

        /// <summary>
        /// Constructor for creating input pins within the diagram. As it is a leaf node,
        /// the children of the input are always null. The inputs also default to a state of
zero.
        /// </summary>
        /// <param name="elementID"> The postorder position within the tree.</param>
        /// <param name="label"> The label of the input. Used to decide whether to display
        /// the element or not. </param>
        public Element(int elementID, char label)
        {
            this.elementID = elementID;
            this.label = label;
            leftChild = null;
            rightChild = null;
            state = 0;
            elementName = "input_pin";

        }

        /// <summary>
        /// Constructor used to create logic gates.
        /// </summary>
        /// <param name="elementName">The name and hence, type, of logic gate</param>
        /// <param name="elementID">The postorder position of the gate within the tree.</param>
        /// <param name="leftChild"> The left child of the gate, this could be an input
        /// or gate. If the element is a NOT gate, then this is null as a NOT only has one
        /// child</param>
        /// <param name="rightChild"> The right child of the gate, this could be an input
        /// or a gate.</param>
```

```
        public Element(string elementName, int elementID, Element leftChild, Element
rightChild)
        {
            this.elementID = elementID;
            this.leftChild = leftChild;
            this.rightChild = rightChild;
            this.elementName = elementName;
            label = ' ';
        }

        public int GetElementID()
        {
            return elementID;
        }

        public char GetLabel()
        {
            return label;
        }

        public string GetElementName()
        {
            return elementName;
        }

        public void SetLogicGate(LogicGate logicGate)
        {
            this.logicGate = logicGate;
        }

        public LogicGate GetLogicGate()
        {
            return logicGate;
        }

        public int GetInstances()
        {
            return instances;
        }

        public void SetInstances(int instances)
        {
            this.instances = instances;
        }

        public void AddInstance()
        {
            instances++;
        }

        public void SetState(int state)
        {
            this.state = state;
        }

        public int GetState()
        {
            return state;
        }
    }
}
```

Wire.cs

This class is responsible for storing all of the points that represent the wire in the logic gate diagram. It also draws the wires onto the canvas. The colour of the wire is also changed when the state of the inputs are changed. This allows the diagram to show how states transmit through the diagram.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Numerics;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Shapes;

namespace _2BNOR_2B.Code
{
    public class Wire
    {
        // Defines whether or not the wire travels to a repeated input. This is to determine
        // whether or not a bridge should be drawn onto the canvas.
        private bool repeated;
        // The input point of the wire (node lower down the tree). Given by the respective
        // logicGate object.
        private Point inputPoint;
        // The output point of the wire (node higher up the tree). Given by the logicGate
        // object.
        private Point outputPoint;
        // The list of points that create the line segments of the wire.
        private readonly List<Point> points = new();
        // The junction on the wire to show that it connects the two wires together.
        private Ellipse junction;
        // The logic gate that gives the wire its state.
        private LogicGate inputGate;
        // The default colour of the wire. Red => a state of 0.
        private Brush colour = Brushes.Red;
        // The canvas in which the wire is drawn onto.
        private readonly Canvas c;
        // asd
        private string wireString = "lll";
        // The shift value applied to a repeated input wire to separate the two wires and
        // make them clearer. The default value is zero as a wire is assumed to not be
repeating.
        private int shift = 0;
        // Properties of the bridges and junctions to make them easily modifiable.
        private readonly int diameterOfArc = 15;
        private readonly int radiusOfJunction = 10;
        // Path which is added to the canvas. This is what stores the created line segments on
        // the canvas. It acts like a container which makes changing the wire (such as the
state)
        // simpler.
        private readonly Path path = new();
        private readonly PathGeometry pathGeometry = new();
        private readonly PathFigureCollection pathFigureCollection = new();


        public Wire(Canvas c)
        {
            this.c = c;
            path.Stroke = colour;
```

```
            path.StrokeThickness = 2;
        }

        public Wire(Point inputPoint, Point outputPoint, Canvas c)
        {
            this.inputPoint = inputPoint;
            this.outputPoint = outputPoint;
            this.c = c;
            path.Stroke = colour;
            path.StrokeThickness = 2;
        }

        public void SetRepeated(bool repeated)
        {
            this.repeated = repeated;
        }

        public void SetStart(Point inputPoint)
        {
            this.inputPoint = inputPoint;
        }

        public void SetEnd(Point outputPoint)
        {
            this.outputPoint = outputPoint;
        }

        public void SetGate(LogicGate logicGate)
        {
            inputGate = logicGate;

        }

        public void SetShift(int shift)
        {
            this.shift = shift;
        }


        public LogicGate GetGate()
        {
            // returns the gate that is the input of the wire.
            return inputGate;
        }

        /// <summary>
        /// Utility method to get all of the points that would form a defined line
        /// segment of the wire. ie Point 1 and Point 2 would form vertical line is
        /// isHorizontal is set to FALSE.
        /// </summary>
        /// <param name="isHorizontal">Decides which type of points to return, it is
        /// nullable to allow all of the points to be get with one method call.</param>
        /// <returns>A list of points specified by the isHorizontal parameter which could
        /// cause only a subset of those points to be returned. </returns>
        public List<Point> GetPoints(bool? isHorizontal)
        {
            if (isHorizontal == true)
            {
                return GetHorizontalPoints();
            }
            else if (isHorizontal == false)
            {
```

```
                return GetVerticalPoints();
            }
            else
            {
                // isHorizontal is null => return all of the wires' points.
                return points;
            }
        }

        /// <summary>
        /// Filters out points so the sequence of points only forms horizontal lines.
        /// </summary>
        /// <returns>A list of points which represent horizontal lines.</returns>
        private List<Point> GetHorizontalPoints()
        {
            var p = new List<Point>();
            for (var i = 0; i < points.Count - 1; i++)
            {
                // A horizontal line => the y-axis is the same for both points.
                if (points[i].Y == points[i + 1].Y)
                {
                    p.Add(points[i]);
                    p.Add(points[i + 1]);
                }
            }
            return p;
        }

        /// <summary>
        /// Filters out points so the sequence of points only forms verical lines.
        /// </summary>
        /// <returns>A list of points which represent vertical lines. </returns>
        private List<Point> GetVerticalPoints()
        {
            var p = new List<Point>();
            for (var i = 0; i < points.Count - 1; i++)
            {
                // A vertical line => the x-axis is the same for both points.
                if (points[i].X == points[i + 1].X)
                {
                    p.Add(points[i]);
                    p.Add(points[i + 1]);
                }
            }
            return p;
        }

        /// <summary>
        /// Sets the colour of the wire based off of the state of the input gate.
        /// This is to show how the states transmit around the diagram.
        /// </summary>
        /// <param name="colour"></param>
        public void SetColour(Brush colour)
        {
            this.colour = colour;
            path.Stroke = colour;
            // Changing the colour of the junction if it exists.
            // It only exists if the wire is a repeated one.
            if (repeated)
            {
                junction.Fill = colour;
                junction.Stroke = colour;
```

```
            }
        }

        /// <summary>
        /// Simple method that calculates the basic shape of the wire given its input and
        /// output points. This method does not consider any intersections or
        /// junctions that may need to be added. However, a shift value is taken into
        /// account for any repeated input wires.
        /// </summary>
        public void SetPoints()
        {
            // Adding the first horizontal line.
            points.Add(inputPoint);
            // A basic wire assumes that the position of the vertical line is simply the
            // midpoint between the input and output points.
            double midpointX = (inputPoint.X + outputPoint.X) / 2 - shift * 10;
            var midpoint = new Point(midpointX, inputPoint.Y);
            // Adding the vertical line to the list of points.
            points.Add(midpoint);
            midpoint.Y = outputPoint.Y;
            points.Add(midpoint);
            // Adding the second horizontal line.
            points.Add(outputPoint);
        }

        /// <summary>
        /// Adds two new points into the array to consider the split sections
        /// of the vertical line and to also mark the diameter of the bridge that will
        /// be drawn.
        /// </summary>
        /// <param name="bridgeLocation">The location on the canvas where a wire intersection
        /// has been found. </param>
        public void AddBridge(Point? bridgeLocation)
        {
            // Inserting the points of the bridge that connect to the straight
            // vertical sections of the wire.
            points.Insert(2, new Point(bridgeLocation.Value.X, bridgeLocation.Value.Y -
diameterOfArc / 2));
            points.Insert(2, new Point(bridgeLocation.Value.X, bridgeLocation.Value.Y +
diameterOfArc / 2));
            // Inserting the bridge into the string so that it will be rendered.
            wireString = wireString.Insert(wireString.Length - 1, "bl");
        }

        /// <summary>
        /// Draws the wire onto the canvas, considering any bridges or junctions
        /// where necessary. This is from the list of points which have been calculated
        /// previously.
        /// </summary>
        public void RenderLine()
        {
            PathFigure pf;
            ArcSegment arc;
            LineSegment line;
            Size s;
            c.Children.Remove(path);
            pathGeometry.Clear();
            pathFigureCollection.Clear();
            // Iterating through the wire string to know what to draw.
            for (var i = 0; i < wireString.Length; i++)
            {
                // Every pathfigure is a segment of the path and so a new object must
```

```
                // be instantiated every time.
                pf = new PathFigure
                {
                    StartPoint = points[i]
                };
                // Draw a line segment if the token indicates a line .
                if (wireString[i] == 'l')
                {
                    line = new LineSegment
                    {
                        Point = points[i + 1],
                        IsStroked = true
                    };
                    // Adding the line to the path figure.
                    pf.Segments.Add(line);
                }
                // Otherwise draw a bridge.
                else
                {
                    // Creating the arc of the wire bridge.
                    s = new Size(diameterOfArc / 4, diameterOfArc / 4);
                    arc = new ArcSegment(points[i + 1], s, 180, true, SweepDirection.Clockwise,
true);
                    // Adding it to the path figure.
                    pf.Segments.Add(arc);
                }
                // Adding to the path figure collection to make a complete
                // section of the wire.
                pathFigureCollection.Add(pf);
            }
            pathGeometry.Figures = pathFigureCollection;
            // setting to the path to the path figure collection
            // containing the completed wire.
            path.Data = pathGeometry;
            path.Stroke = Brushes.Black;
            // Setting a low ZIndex to ensure that the wires are not
            // drawn over the logic gates on the canvas.
            Panel.SetZIndex(path, 1);
            // Adding the wire to the canvas.
            c.Children.Add(path);

            // If a wire is a repeated one, then a junction must be added
            // to show this.
            if (repeated)
            {
                // Create the junction.
                junction = new Ellipse
                {
                    Width = radiusOfJunction,
                    Height = radiusOfJunction,
                    Fill = Brushes.Black,
                    Stroke = Brushes.Black
                };
                // Setting its position, which is always the joint between
                // the horizontal input wire and the vertical wire.
                Canvas.SetTop(junction, points[points.Count - 2].Y - radiusOfJunction / 2);
                Canvas.SetLeft(junction, points[points.Count - 2].X - radiusOfJunction / 2);
                // Setting the low ZIndex to stop overlapping and adding
                // the junction to the canvas.
                Panel.SetZIndex(junction, 1);
                c.Children.Add(junction);
            }
```

```
        }
    }
}
```

Diagram.cs

This is the main class of the program; it brings together all of the processing needed for the program. This includes minimisation, drawing logic diagrams and truth tables. The diagram class also constructs the main binary tree for the program which is used to process the expressions and make the diagram interactive. It is also the main class that is interacted with through the user interface. The class also contains the validation methods for the user interface.

```
using Microsoft.VisualBasic;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Globalization;
using System.Linq;
using System.Net.NetworkInformation;
using System.Numerics;
using System.Reflection.Metadata;
using System.Runtime.CompilerServices;
using System.Runtime.Intrinsics.X86;
using System.Security;
using System.Text.RegularExpressions;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Markup;
using System.Windows.Media;

namespace _2BNOR_2B.Code
{
    /// <summary>
    /// Serves as the main class of the application. It handles most of the expression
    /// processing and drawing of diagrams/tables to their respective canvases.
    /// </summary>
    public class Diagram
    {
        // Simply used to remove whitespace from entered Boolean expressions.
        private static readonly Regex r = new(@"\s+");
        private readonly char[] booleanOperators = { '+', '^', '.', '!' };
        private readonly string[] gateNames = { "or_gate", "xor_gate", "and_gate", "not_gate"
};
        // Stores the input portion of a truth table. For a 2 input table => ["00", "01", etc.]
        private string[] inputMap;
        // Stores the complete truth table. For an AND gate => ["000", "010", "100", "111"]
        private string[] outputMap;
        private string[] headers;
        // The only characters allowed to be in the user entered expression.
        private static readonly string validCharacters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ().!^+10";
        private string infixExpression = "";
        private string minimisedExpression = "";
        // A binary string of the inputs of the diagram. Position corresponds to input.
        // e.g. position 0 => A, position 1 => B, etc.
        private string inputStates = "";
        // Stores the binary tree that is being drawn, traversed, etc.
        private Element rootNode;
        // Stores the visual output of the logic gate diagram however, it does not need to
        // be considered when traversing the diagram.
        private Element outputNode;
        private Element[] elements;
        private Element[] inputs;
        private Wire[] wires;
```

```
        // The main window canvas. This is the one where logic diagrams are drawn.
        private readonly Canvas c;
        // Constants used for the diagram drawing formulae.
        // Square allocation for elements.
        readonly int elementWidth = 2;
        readonly int xOffset = 12;
        // Number of pixels on the canvas each square takes up.
        readonly int pixelsPerSquare = 15;
        readonly int maxNumberOfInputs = 13;
        readonly int minCellWidth = 30;
        readonly double initialXofTable = 20;
        readonly double initialYofTable = 20;
        double canvasWidth;

        public Diagram(Canvas c)
        {
            this.c = c;
        }

        /// <returns>The binary tree representing the logic gate diagram. </returns>
        public Element GetTree()
        {
            return rootNode;
        }

        /// <summary>
        /// Removes all whitespace from an entered string.
        /// </summary>
        /// <param name="input">The string where whitespace is being removed from. </param>
        /// <param name="replacement">The character that is replacing the whitespace.</param>
        /// <returns>The input with all whitespace replaced with a defined string. </returns>
        private static string RemoveWhitespace(string input, string replacement)
        {
            return r.Replace(input, replacement);
        }

        /// <summary>
        /// Sets the infix (user-entered) expression of the logic gate diagram.
        /// </summary>
        /// <param name="expression">The new expression that will be stored. </param>
        public void SetExpression(string expression)
        {
            infixExpression = expression;
        }

        public string GetExpression()
        {
            return infixExpression;
        }

        public string GetMinimisedExpression()
        {
            return minimisedExpression;
        }

        /// <summary>
        /// Clears all relevant values stored in the diagram. This ensures a clean slate
        /// for when new expressions are entered and manipulated by the user.
        /// </summary>
        public void ClearDiagram()
        {
            rootNode = null;
```

```
            outputNode = null;
            wires = null;
            outputMap = null;
            inputMap = null;
            headers = null;
            elements = null;
            inputStates = null;
            infixExpression = "";
        }

        /// <summary>
        /// Converts a user entered infix Boolean expression to the postfix representation
        /// of the Boolean expression. This is a modified version of the 'Shunting yard' as
        /// given by the pseudo-code on Wikipedia.
        /// </summary>
        /// <param name="infixExpression">An infix Boolean expression. </param>
        /// <returns>The postfix Boolean expression of the supplied infix expression.</returns>
        private string ConvertInfixtoPostfix(string infixExpression)
        {
            infixExpression = RemoveWhitespace(infixExpression, "");
            var operatorStack = new Stack<char>();
            string postfixExpression = "";
            int operatorPrecedence;
            foreach (char token in infixExpression)
            {
                if (char.IsLetter(token) || char.IsNumber(token))
                {
                    postfixExpression += token;
                }
                else if (booleanOperators.Contains(token))
                {
                    operatorPrecedence = Array.IndexOf(booleanOperators, token);
                    while (operatorStack.Count > 0 && operatorStack.Peek() != '(' &&
Array.IndexOf(booleanOperators, operatorStack.Peek()) > operatorPrecedence)
                    {
                        postfixExpression += operatorStack.Pop();
                    }
                    operatorStack.Push(token);
                }
                else if (token == '(')
                {
                    operatorStack.Push(token);
                }
                else if (token == ')')
                {
                    while (operatorStack.Peek() != '(')
                    {
                        Debug.Assert(operatorStack.Count > 0, "The stack is empty.");
                        postfixExpression += operatorStack.Pop();
                    }
                    Debug.Assert(operatorStack.Peek() == '(', "The top item is a (");
                    operatorStack.Pop();
                }
            }
            while (operatorStack.Count > 0)
            {
                Debug.Assert(operatorStack.Peek() != '(', "The top item is a (");
                postfixExpression += operatorStack.Pop();
            }
            return postfixExpression;
        }
```

```
        /// <summary>
        /// When the diagram is clicked, get the states of the inputs, and map the states
        /// of the truth table to the gates within the logic gate diagram. Colour the wires
        /// to be respective of the states of the gates.
        /// </summary>
        public void UpdateWires()
        {
            inputStates = "";
            Getinputstates(rootNode);
            AssignGateStates(rootNode);
            ColourWires();
        }

        #region Validation routines
        /// <summary>
        /// Used to validate the postfix produced by the Shunting yard. If this conversion
        /// causes an error then the original postfix must be invalid and so the expression
        /// should not be accepted by the program.
        /// </summary>
        /// <param name="postfixExpression">A postfix Boolean expression produced by the
        /// program, which is going to be validated by this method.</param>
        private static void ConvertPostfixtoInfix(string postfixExpression)
        {
            var s = new Stack<string>();
            string operand1;
            string operand2;
            // Tokenising the expression.
            foreach (char c in postfixExpression)
            {
                // Inputs can be pushed straight to the stack.
                if (char.IsLetter(c) || char.IsNumber(c))
                {
                    s.Push(c.ToString());
                }
                // If an operator has been found then pop the desired number of operators
                // and push the result to the stack.
                else if (c == '!')
                {
                    operand1 = s.Pop();
                    s.Push($"{c}{operand1}");
                }
                else
                {
                    operand1 = s.Pop();
                    operand2 = s.Pop();
                    s.Push($"{operand1}{c}{operand2}");
                }
            }
        }

        /// <summary>
        /// Checks whether the user entered Boolean expression contains any invalid
        /// characters.
        /// </summary>
        /// <param name="expression">The expression being checked.</param>
        /// <returns>Whether or not the expression contains invalid characters.</returns>
        private static bool InvalidCharacters(string expression)
        {
            // Tokenising the expression so that each character can be checked.
            foreach (char c in expression)
            {
                if (validCharacters.Contains(c) == false)
```

```
                {
                    return false;
                }
            }
            return true;
        }

        /// <summary>
        /// Produces a string of only brackets from a user entered Boolean expression.
        /// This will be used within the validate brackets method to ensure that the
        /// brackets of the expression have been entered properly.
        /// </summary>
        /// <param name="expression">A user entered Boolean expression. </param>
        /// <returns>A string containing only brackets "(A.B)" => "()"</returns>
        private static string RemoveComponents(string expression)
        {
            string result = "";
            foreach (char c in expression)
            {
                if (c == '(' || c == ')')
                {
                    result += c;
                }
            }
            return result;
        }

        /// <summary>
        /// Ensures that the brackets in the user-entered Boolean expression has a valid
        /// sequence of brackets. ie () => valid and (() => invalid, etc.
        /// </summary>
        /// <param name="expression">The user entered expression that is being checked.</param>
        /// <returns>Returns a Boolean showing whether or not the brackets are valid.</returns>
        private static bool ValidateBrackets(string expression)
        {
            // Getting a string of just the brackets to make the validation easier.
            // The order of operands and operators is checked during postfix/infix check.
            string brackets = RemoveComponents(expression);
            var s = new Stack<char>();
            // Tokenising the expression.
            foreach (char c in brackets)
            {
                // If the token is an open bracket, then the corresponding closed bracket must
                // exist and so push it onto the stack.
                if (c == '(')
                {
                    s.Push(')');
                }
                // If no items remain on the stack and the token exists => bracket imbalance =>
                // the expression is invalid. Or pop an item and see if the close bracket
                // matches with the current token.
                else if (s.Count == 0 || s.Pop() != c)
                {
                    return false;
                }
            }

            // If no tokens remain on the stack, then all brackets have been matched and so the
            // string is valid, in terms of its brackets.
            if (s.Count == 0)
            {
                return true;
```

```
            }
            else
            {
                return false;
            }
        }

        /// <summary>
        /// Produces a Boolean value showing whether or not the ASCII codes of the entered
        /// Boolean expression is sequential. ie, "A+B" => valid and "A+C" => invalid.
        /// </summary>
        /// <param name="expression">The user entered expression being validated. </param>
        /// <returns>Whether or not the expression contains sequential inputs.</returns>
        private static bool IsSequential(string expression)
        {
            // Getting all of the inputs of the expression in array form so that they can
            // be sorted into alphabetical order.
            char[] inputs = GetOnlyInputs(expression);
            Array.Sort(inputs);
            for (var i = 0; i < inputs.Length; i++)
            {
                // validCharacters string is written in sequential order therefore if the
                // sorted inputs do not match this then the string is invalid otherwise,
                // string is valid.
                if (inputs[i] != validCharacters[i])
                {
                    return false;
                }
            }
            return true;
        }

        /// <summary>
        /// Gets all of the unique inputs of a supplied Boolean expression.
        /// </summary>
        /// <param name="expression">The expression being filtered.</param>
        /// <returns>An array representing the sequence of unique inputs within
        /// the user entered Boolean expression. </returns>
        private static char[] GetOnlyInputs(string expression)
        {
            // Ensures that constants (0 or 1) are not considered so subtract from the
            // number of unique inputs.
            char[] inputs = new char[GetNumberOfInputs(expression, true) -
GetNumberOfConstants(expression)];
            int i = 0;
            foreach (char c in expression)
            {
                // Ensuring only unique inputs are added to the array.
                if (char.IsLetter(c) && inputs.Contains(c) == false)
                {
                    inputs[i] = c;
                    i++;
                }
            }
            return inputs;
        }


        /// <param name="expression">The user-entered Boolean expression being
validated.</param>
        /// <returns>The number of constants (0 or 1) in the entered Boolean
expression.</returns>
```

```
        private static int GetNumberOfConstants(string expression)
        {
            int numberOfConstants = 0;
            foreach (char c in expression)
            {
                if (c == '0' || c == '1')
                {
                    numberOfConstants++;
                }
            }
            return numberOfConstants;
        }


        /// <param name="expression">Postfix Boolean expression being validated.</param>
        /// <returns>Whether or not that postfix expression is valid postfix.</returns>
        private static bool PostfixCheck(string expression)
        {
            // If the postfix cannot be converted to infix, the supplied postfix must be
            // invalid so it fails validation.
            try
            {
                ConvertPostfixtoInfix(expression);
                return true;
            }
            catch
            {
                return false;
            }
        }

        /// <summary>
        /// Main Validation method, this ties together all of the checks to produce a
        /// comprehensive validation algorithm.
        /// </summary>
        /// <param name="expression">The user entered Boolean expression from the expression
        /// dilog.</param>
        /// <param name="isTable">Whether or not the expression being validated is
        /// for producing a truth table or not.</param>
        /// <returns>A Boolean value representing whether or not the
        /// Boolean expression is a valid one. (true => valid) </returns>
        public bool IsExpressionValid(string expression)
        {
            string postfix = ConvertInfixtoPostfix(expression);
            if (IsSequential(expression) && InvalidCharacters(expression) &&
ValidateBrackets(expression))
            {
                // Imposing an input limit for Petrick's method. This is a choice because
                // it means that the term to implicant map has a unique character as the
                // key.
                if (GetNumberOfInputs(expression, true) > maxNumberOfInputs)
                {
                    return false;
                }
                else
                {
                    // Ensuring that the postfix is valid postfix. Covers expressions
                    // such as "(A.)" which pass the other checks.
                    return PostfixCheck(postfix);
                }
            }
            // If the expression is of the incorrect form, then discard immediately as valid
```

```
            // postfix cannot be produced from the expression.
            else
            {
                return false;
            }
        }

        #endregion

        #region diagram drawing
        /// <summary>
        /// Iterative postorder traverasl of the binary tree representing the logic diagram.
        /// The traversal does not consider whether or not an input is visible or not.
        /// It simply produces a string of the input states.
        /// </summary>
        /// <param name="root"> The root of the tree. The last gate/input and also
        /// the point where the traversal starts. </param>
        private void Getinputstates(Element root)
        {
            var s = new Stack<Element>();
            string visited = "";
            while (true)
            {
                while (root != null)
                {
                    // Push the root onto the stack and traverse to the left child.
                    // Repeat until this cannot be done anymore. Same as traversing to the
                    // left-most gate in the tree.
                    s.Push(root);
                    s.Push(root);
                    root = root.leftChild;
                }
                if (s.Count == 0)
                {
                    return;
                }
                root = s.Pop();
                if (s.Count != 0 && s.Peek() == root)
                {
                    root = root.rightChild;
                }
                else
                {
                    // Get the state if the node the current traversal is on, is an input.
                    if (root.GetElementName() == "input_pin")
                    {
                        inputStates += root.GetState();
                        visited += root.GetLabel();
                        root = null;
                    }
                    else
                    {
                        //Used to traverse to the right child.
                        root = null;
                    }
                }
            }
        }

        /// <param name="root">The root node (startpoint) of the tree.</param>
        /// <returns>The maximum depth (height) of the supplied binary tree. </returns>
        private static int GetHeightOfTree(Element root)
```

```
        {
            // If the root does not exist then the tree does not exist and therefore the
            // height of the tree is zero.
            if (root == null)
            {
                return 0;
            }
            // Recurse to the next level of the tree.
            // Results in two sums of the deepest left/right child.
            int leftChildHeight = GetHeightOfTree(root.leftChild);
            int rightChildHeight = GetHeightOfTree(root.rightChild);
            // Need to add one so that the root is considered within the height. Otherwise
            // return the largest value.
            if (leftChildHeight > rightChildHeight)
            {
                return leftChildHeight + 1;
            }
            else
            {
                return rightChildHeight + 1;
            }
        }

        private Element GetInputWithSameLabel(char label)
        {
            foreach (Element e in elements)
            {
                if (e.GetLabel() == label)
                {
                    return e;
                }
            }
            return null;
        }

        /// <summary>
        /// Finds the number of non-visual nodes that are within the binary tree.
        /// </summary>
        /// <param name="root">The root of the tree storing the entire tree itself.</param>
        /// <returns>The number of non-visual nodes within the supplied tree. </returns>
        private static int GetNumberOfNodes(Element root)
        {
            // If the root does not exist then the tree must not exist and so the number of
            // nodes must be zero.
            if (root == null)
            {
                return 0;
            }
            // Adding one because the root node exists and then traversing the entire tree.
            return 1 + GetNumberOfNodes(root.leftChild) + GetNumberOfNodes(root.rightChild);
        }

        /// <summary>
        /// Converts the user-entered and validated infix Boolean expression and represents
        /// it as a binary tree. This is the same logic as the evaluated where in postfix
        /// the first operands are the deepest within the tree.
        /// </summary>
        /// <param name="inputExpression">A user-entered infix Boolean expression that will
        /// be represented as the binary tree. This is for drawing the logic diagram.</param>
        private void GenerateBinaryTreeFromExpression(string inputExpression)
        {
            string postfixExpression = ConvertInfixtoPostfix(inputExpression);
```

```
            inputs = new Element[GetNumberOfInputs(inputExpression, false)];
            var nodeStack = new Stack<Element>();
            elements = new Element[inputExpression.Length + 1];
            Element nodeToAdd;
            Element leftChild;
            Element rightChild;
            Element tmp;
            int elementID = 0;
            int i = 0;
            string elementName;
            string inputsAdded = "";
            foreach (char c in postfixExpression)
            {
                if (char.IsLetter(c) && char.IsUpper(c) || char.IsNumber(c))
                {
                    nodeToAdd = new Element(elementID, c);
                    inputs[i] = nodeToAdd;
                    i++;
                    if (char.IsNumber(c))
                    {
                        nodeToAdd.SetState(c - 48);
                    }
                    if (inputsAdded.Contains(c) == false)
                    {
                        nodeToAdd.SetInstances(1);
                        inputsAdded += c;
                    }
                    else
                    {
                        tmp = inputs[c - 65];
                        tmp.AddInstance();
                    }
                }
                else if (c == '!')
                {
                    rightChild = nodeStack.Pop();
                    nodeToAdd = new Element("not_gate", elementID, null, rightChild);
                    nodeToAdd.SetInstances(1);
                    rightChild.parent = nodeToAdd;
                }
                else
                {
                    // Any logic gate is a binary operator, so pop two items and these are
                    // the operands for the current Boolean operation.
                    rightChild = nodeStack.Pop();
                    leftChild = nodeStack.Pop();
                    elementName = gateNames[Array.IndexOf(booleanOperators, c)];
                    nodeToAdd = new Element(elementName, elementID, leftChild, rightChild);
                    nodeToAdd.SetInstances(1);
                    leftChild.parent = nodeToAdd;
                    rightChild.parent = nodeToAdd;
                }
                nodeStack.Push(nodeToAdd);
                elements[elementID] = nodeToAdd;
                elementID++;
            }
            rootNode = nodeStack.Pop();
        }

        /// <summary>
        /// Calculates the y-position of the logic gate on the canvas given the logic
        /// gate diagram being drawn.
```

```
        /// </summary>
        /// <param name="heightOfTree">The height of the tree. This is the deepest node in
        /// the tree. </param>
        /// <param name="depthWithinTree">Integer which stores the current layer that the
        /// node is on. This is also the layer that the BST is on. </param>
        /// <param name="positionWithinLayer">The x-position within the tree at the
        /// depth given by depthWithinTree</param>
        /// <returns>The Y-position of the node on the canvas given its position
        /// within the tree. </returns>
        private double CalculateNodeYposition(int heightOfTree, int depthWithinTree, int
positionWithinLayer)
        {
            // Formula to calculate the spacing for a node in this position within tree
            // in relation to the size of the tree.
            double initialY = Math.Pow(2, heightOfTree) / Math.Pow(2, depthWithinTree) *
pixelsPerSquare;
            // Multiply by the position within the layer to get the Y position of the node.
            // This is the same idea as the X position calculation.
            initialY += initialY * positionWithinLayer * 2;
            // Apply a squish to the Y position so the size of the diagram is reduced.
            // This makes odd-shaped diagrams more easily viewable.
            if (heightOfTree > 5 && depthWithinTree < 4)
            {
                return initialY / 2;
            }
            else
            {
                return initialY;
            }
        }

        /// <summary>
        /// Calculates the X position of a logic gate on the canvas. This is simply
        /// applying a shift factor across the canvas based off of the depth within the tree.
        /// </summary>
        /// <param name="depthWithinTree">The depth within the tree, the current node is.
</param>
        /// <returns>The X position on the canvas which is the location of logic
gate.</returns>
        private double CalculateXposition(int depthWithinTree)
        {
            return canvasWidth - ((pixelsPerSquare - 7) * elementWidth + (pixelsPerSquare - 7)
* xOffset) * depthWithinTree;
        }

        /// <summary>
        /// Translates the node across the canvas that ensures the logic gates do not
        /// spill over the left side of the canvas.
        /// </summary>
        /// <param name="startX">The start position of the node on the canvas.</param>
        /// <param name="heightOfTree">The height of the binary tree being drawn.</param>
        /// <returns>The translated position of the node on the canvas. This is the
        /// final position of the node on the canvas. </returns>
        private double TranslateNode(double startX, int heightOfTree)
        {
            double maxX = CalculateXposition(heightOfTree);
            return startX - maxX + 50;
        }

        /// <summary>
        /// Linking method which calculates the final position of the node on the canvas.
        /// </summary>
```

```
        /// <param name="node">The node being drawn to the canvas. </param>
        /// <param name="heightOfTree">The height of the binary tree being drawn.</param>
        /// <param name="depthWithinTree">The current depth of the node within the
tree.</param>
        /// <returns></returns>
        private double CalculateNodeXposition(Element node, int heightOfTree, int
depthWithinTree)
        {
            double xPosition;
            // If the node is an input(leaf node) then it should be drawn at the
            // left-most position of the tree. This is the height of the binary tree.
            if (node.leftChild == null && node.rightChild == null)
            {
                xPosition = CalculateXposition(heightOfTree);
            }
            else
            {
                // Calculating at that particular depth.
                // Subtracting fifty pixels to remain within the bounds of the canvas.
                xPosition = CalculateXposition(depthWithinTree) - 50;
            }
            // Translating the node to stay within the bounds of the canvas.
            return TranslateNode(xPosition, heightOfTree);
        }

        /// <summary>
        /// Utility method which is responsible for drawing the left child of the node.
        /// The method simply sets the points for the wires. It is important to note that
        /// intersections between the wires are not considered.
        /// </summary>
        /// <param name="root">The wire connects to the input of this node.</param>
        /// <returns>A wire with its points set. Ready to be drawn to the canvas. </returns>
        private Wire DrawWiresForLeftChildren(Element root)
        {
            var wire = new Wire(c);
            LogicGate rootLogicGate = root.GetLogicGate();
            LogicGate leftchildLogicGate = root.leftChild.GetLogicGate();
            Element input;
            wire.SetStart(rootLogicGate.GetInputPoint1());
            // If the left child does not have a logic gate, then it must be a repeated
            // node and so the input with the same label needs to be found.
            if (leftchildLogicGate != null)
            {
                wire.SetRepeated(false);
                wire.SetEnd(leftchildLogicGate.GetOutputPoint());
                wire.SetGate(leftchildLogicGate);
                wire.SetShift(leftchildLogicGate.GetConnectedWires());
                wire.SetPoints();
            }
            else
            {
                // The wire goes to a node that is a repeated input.
                wire.SetRepeated(true);
                // Searching for the input with the same label as this is the visual
                // node that the wire needs to be drawn to.
                input = GetInputWithSameLabel(root.leftChild.GetLabel());
                wire.SetEnd(input.GetLogicGate().GetOutputPoint());
                wire.SetGate(input.GetLogicGate());
                input.GetLogicGate().AddWire();
                // Applying a shift value to separate the wires going to the same
                // gate. This makes the diagram clearer.
                wire.SetShift(input.GetLogicGate().GetConnectedWires());
```

```
                // Setting the points but not rendering so that the intersections
                // can be calculated and included.
                wire.SetPoints();
            }
            // Returning the wire so that it can be added to the array of wires.
            return wire;
        }

        /// <summary>
        /// Utility method which is resonsible draw drawing the right child of the node.
        /// The method simply sets the points for the wires. It is important to note that
        /// intersections between the wires are not considered.
        /// </summary>
        /// <param name="root">The wire connects to the input of this node.</param>
        /// <returns>A wure with its points set. Ready to be drawn to the canvas. </returns>
        private Wire DrawWiresForRightChildren(Element root)
        {
            var wire = new Wire(c);
            LogicGate rootLogicGate = root.GetLogicGate();
            LogicGate rightchildLogicGate = root.rightChild.GetLogicGate();
            Element input;
            wire.SetStart(rootLogicGate.GetInputPoint2());
            // If the left child does not have a logic gate, then it must be a repeated
            // node and so the input with the same label needs to be found.
            if (rightchildLogicGate != null)
            {
                wire.SetRepeated(false);
                wire.SetEnd(rightchildLogicGate.GetOutputPoint());
                wire.SetGate(rightchildLogicGate);
                wire.SetShift(rightchildLogicGate.GetConnectedWires());
                wire.SetPoints();
            }
            else
            {
                // The wire goes to a node that is a repeated input.
                wire.SetRepeated(true);
                // Searching for the input with tha same label as this is the visual
                // node that the wire needs to be drawn to.
                input = GetInputWithSameLabel(root.rightChild.GetLabel());
                wire.SetEnd(input.GetLogicGate().GetOutputPoint());
                wire.SetGate(input.GetLogicGate());
                input.GetLogicGate().AddWire();
                // Applying a shift value to separate the wires going to the same
                // gate. This makes the diagram clearer.
                wire.SetShift(input.GetLogicGate().GetConnectedWires());
                // Setting the points but not rendering so that the intersections
                // can be calculated and included.
                wire.SetPoints();
            }
            // Returning the wire so that it can be added to the array of wires.
            return wire;
        }

        /// <summary>
        /// Calculates the point of intersection, if any, given four points which
        /// corresponds to two lines that are on the canvas. These are always a
        /// vertical and horizontal line as this is the only case where an
        /// intersection would have to be drawn.
        /// </summary>
        /// <param name="p0">The start point of the vertical line.</param>
        /// <param name="p1">The end point of the vertical line.</param>
        /// <param name="p2">The start point of the horizontal line.</param>
```

```
        /// <param name="p3">The end point of the horizontal line.</param>
        /// <returns>The point on the canvas where the intersection is located.</returns>
        private static Point? FindIntersection(Point p0, Point p1, Point p2, Point p3)
        {
            double s_x = p1.X - p0.X;
            double s_y = p1.Y - p0.Y;
            double s2_x = p3.X - p2.X;
            double s2_y = p3.Y - p2.Y;
            double denom = s_x * s2_y - s2_x * s_y;

            if (denom == 0)
            {
                // The line supplied are collinear.
                return null;
            }
            bool isDenomPositive = denom > 0;

            double s3_x = p0.X - p2.X;
            double s3_y = p0.Y - p2.Y;
            double s_numer = s_x * s3_y - s_y * s3_x;

            if (s_numer < 0 == isDenomPositive)
            {
                // There is no intersection between the two lines.
                return null;
            }

            double t_numer = s2_x * s3_y - s_x * s3_y;

            if (t_numer < 0 == isDenomPositive)
            {
                // There is no intersection between the two lines.
                return null;
            }

            if (s_numer > denom == isDenomPositive || t_numer > denom == isDenomPositive)
            {
                // There is no intersection between the lines.
                return null;
            }

            // There must be an intersection between the two lines.
            double t = t_numer / denom;
            // Finding the position and returning the point of intersection that has been
            // found between the two lines.
            Point? result = new Point(p0.X + t * s_x, p0.Y + t * s_y);
            return result;
        }

        /// <summary>
        /// Computes all of the intersections within the logic gate diagram being drawn.
        /// This is done by iterating through all of the line segments which make up a
        /// wire. It should be noted that the intersections are not drawn, they are only
        /// added to the necessary wires' list of points.
        /// </summary>
        private void DrawIntersections()
        {
            // List that stores sets of points which create horizontal lines.
            // This is all of the horizontal segments of the wires.
            var horizontalLines = new List<Point>();
            // List that stores all of the vertical line segments of the wires.
            var verticalLines = new List<Point>();
```

```
            Point? intersection;
            Wire tmp;
            // Add all of the horizontal and vertical points to the respective lists.
            for (var j = 0; j < wires.Length - 1; j++)
            {
                horizontalLines.AddRange(wires[j].GetPoints(true));
                verticalLines.AddRange(wires[j].GetPoints(false));
            }

            for (var i = 0; i < verticalLines.Count - 1; i += 2)
            {
                for (var c = 0; c < horizontalLines.Count - 1; c += 2)
                {
                    // Calculating the intersection, null => an intersection does not exist
                    // between these two lines.
                    intersection = FindIntersection(verticalLines[i], verticalLines[i + 1],
horizontalLines[c], horizontalLines[c + 1]);
                    // Ensuring the intersection exists and the intersection is not formed
                    // a horizontal and vertical line of the same wire before the intersection
                    // is added.
                    if (intersection != null && FindWire(verticalLines[i]) !=
FindWire(horizontalLines[c]))
                    {
                        tmp = FindWire(verticalLines[i]);
                        tmp.AddBridge(intersection);
                    }
                }
            }
        }

        /// <summary>
        /// Carries out a linear search on the wires drawn, given a point.
        /// </summary>
        /// <param name="p">The point p for which identifies the wire being searched. </param>
        /// <returns>The wire which contains the provided point. </returns>
        /// <exception cref="Exception">A wire containing the supplied point could not be
found.
        /// </exception>
        private Wire FindWire(Point p)
        {
            List<Point> points;
            foreach (Wire wire in wires)
            {
                // Get all points of the wire, both horizontal and vertical.
                points = wire.GetPoints(null);
                if (points.Contains(p))
                {
                    return wire;
                }
            }
            throw new Exception("Could not find wire.");
        }

        /// <summary>
        /// Draws the wires to create the logic gate diagram. The states of the gates has
        /// not been set so the colour of the wires are at their default state.
        /// </summary>
        /// <param name="root">The root node of the binary tree. </param>
        private void DrawWires(Element root)
        {
            var q = new Queue<Element>();
            // The number of wires is always the number of nodes as every node has
```

```
            // a wire that is drawn from the nodes output.
            wires = new Wire[GetNumberOfNodes(root)];
            Element tmp;
            int i = 0;
            // Traversing the nodes using a breadth first traveral to add the wires
            // in that order.
            q.Enqueue(root);
            while (q.Count != 0)
            {
                tmp = q.Dequeue();
                // Traversing the left child if it exists.
                if (tmp.leftChild != null)
                {
                    // Adding the wire created into the array.
                    wires[i] = DrawWiresForLeftChildren(tmp);
                    i++;
                    // Traversing the left child because it exists.
                    q.Enqueue(tmp.leftChild);
                }
                // Traversing the right child if it exists.
                if (tmp.rightChild != null)
                {
                    // Adding the created wire into the array.
                    wires[i] = DrawWiresForRightChildren(tmp);
                    i++;
                    // Traversing the right child because it exists.
                    q.Enqueue(tmp.rightChild);
                }
            }
            // Intersections have not been considered up until this point.
            // Compute all of the intersections of the currently drawn wires.
            DrawIntersections();
            // Adding the wires to the canvas as all of the points have been calculated.
            for (var j = 0; j < wires.Length - 1; j++)
            {
                wires[j].RenderLine();
            }
        }

        /// <summary>
        /// Small function that colours all of the wires when a change of state has
        /// occurred. This is to show the user the states transmit throughout the
        /// diagram.
        /// </summary>
        private void ColourWires()
        {
            foreach (Wire wire in wires)
            {
                // Every wire is connected to a logic gate and so we can get the
                // element behind it to colour the state of the wire appropriately.
                LogicGate logicGate = wire.GetGate();
                Element node = logicGate.GetGate();
                // Setting the colour of the wire.
                if (node.GetState() == 1)
                {
                    wire.SetColour(Brushes.Green);
                }
                else
                {
                    wire.SetColour(Brushes.Red);
                }
            }
```

```
        }

        /// <summary>
        /// Adds a node to the diagram canvas based off of the position of the node
        /// within the tree. It also readjusts the canvas to make the ScrollViewer
        /// visible when the diagram being drawn is too large.
        /// </summary>
        /// <param name="currentNode">The node behind the visual node being added to
        /// the canvas. </param>
        /// <param name="heightOfTree">The height of the binary tree being drawn.</param>
        /// <param name="depthWithinTree">The layer within the tree that the node being
        /// drawn is on. </param>
        /// <param name="positionWithinLayer">The horizontal position within the layer
        /// of the tree that the node is on. </param>
        private void DrawNode(Element currentNode, int heightOfTree, int depthWithinTree, int
positionWithinLayer)
        {
            LogicGate logicGate;
            double xPosition;
            double yPosition;
            // Ensuring that the node exists.
            if (currentNode.GetInstances() != 0)
            {
                // Calculating the x-position of the node on the canvas. This is the
                // same value in most cases. Unless the node is an input, but this
                // does not make a huge difference to the methods usage here.
                xPosition = CalculateNodeXposition(currentNode, heightOfTree, depthWithinTree);
                // Check to show that the child of the NOT gate is drawn in parallel
                if (currentNode.parent != null && currentNode.parent.GetElementName() ==
"not_gate" && currentNode.GetElementName() == "input_pin")
                {
                    // The y-position is simply the same as the parent.
                    yPosition = Canvas.GetTop(currentNode.parent.GetLogicGate());
                }
                else
                {
                    // Otherwise calculate the y-position based off of the position of
                    // the node within the binary tree.
                    yPosition = CalculateNodeYposition(heightOfTree, depthWithinTree,
positionWithinLayer);
                }
                // Creating a visual gate because the node is not a repeated input.
                logicGate = new LogicGate(currentNode);
                // Linking the visual and non-visual node together.
                currentNode.SetLogicGate(logicGate);
                // Setting the position of the logic gate onto the canvas, readjusting the
                // canvas size for the scroll viewer and then adding the visual element to
                // the canvas.
                Canvas.SetLeft(logicGate, xPosition);
                Canvas.SetTop(logicGate, yPosition);
                c.Height = Math.Max(yPosition, c.Height) + 30;
                c.Width = Math.Max(xPosition, c.Width) + 30;
                double position = logicGate.GetInputPoint2().Y + 50;
                Canvas.SetBottom(logicGate, position);
                Canvas.SetRight(logicGate, logicGate.GetInputPoint2().X);
                // Setting a large Zindex value so that the wires are not drawn over the
                // logic gates on the canvas as the gates are drawn first.
                Panel.SetZIndex(logicGate, 3);
                c.Children.Add(logicGate);
            }
        }
```

```
        /// <summary>
        /// A modified breadth first traversal that is responsible for placing all of the
        /// logic gates onto the diagram canvas. The modification is that the traversal
        /// also tracks the horizontal position within a given depth of the binary tree.
        /// This allows the y-position to be calculated.
        /// </summary>
        /// <param name="root">The root node of the binary tree. This is the last logic
        /// gate.</param>
        /// <param name="heightOfTree">The height of the tree given. This is the maximum
        /// depth of the tree. </param>
        private void DrawNodes(Element root, int heightOfTree)
        {
            var q = new Queue<Element>();
            q.Enqueue(root);
            int depthWithinTree = 0;
            // The horizontal position within a given depth of the tree.
            int positionWithinLayer = 0;
            int sizeOfQ;
            Element currentNode;
            while (q.Count != 0)
            {
                sizeOfQ = q.Count;
                // Travelling along a particular layer of the binary tree.
                while (sizeOfQ != 0)
                {
                    currentNode = q.Peek();
                    // Draw node at the current position of the breadth first traversal.
                    DrawNode(currentNode, heightOfTree, depthWithinTree, positionWithinLayer);
                    q.Dequeue();
                    // Travelling horizontally along the tree.
                    positionWithinLayer++;
                    // If the left child exists then add it to the queue to be reached.
                    if (currentNode.leftChild != null)
                    {
                        q.Enqueue(currentNode.leftChild);
                    }
                    // If the right child exists then add it to the queue to be reached.
                    if (currentNode.rightChild != null)
                    {
                        q.Enqueue(currentNode.rightChild);
                    }
                    sizeOfQ--;
                }
                // Next level of the tree so increase the depth and reset the position
                // within the layer to the left-most node => 0.
                depthWithinTree++;
                positionWithinLayer = 0;
            }
        }

        /// <summary>
        /// Carries out an iterative postorder traversal on the binary tree. This
        /// is to set the states of the elements in the tree to the corresponding
        /// cell in the truth table, given the states of the inputs in the diagram.
        /// </summary>
        /// <param name="root">The root node of the binary tree. </param>
        private void AssignGateStates(Element root)
        {
            // Get the correct row of the truth table based off of the state of the
            // inputs in the diagram.
            string tableRow = GetTruthTableRow();
            var s = new Stack<Element>();
```

```
            int i = 0;
            int state;
            // Iterative postorder traversal.
            while (true)
            {
                while (root != null)
                {
                    s.Push(root);
                    s.Push(root);
                    // Traverse down the left sub-tree until no longer.
                    root = root.leftChild;
                }
                if (s.Count == 0)
                {
                    // The traversal is complete as no nodes remain on the stack.
                    return;
                }
                root = s.Pop();
                if (s.Count != 0 && s.Peek() == root)
                {
                    // Traverse to the right sub-tree.
                    root = root.rightChild;
                }
                else
                {
                    // Assign the state to the element.
                    state = tableRow[i] - 48;
                    root.SetState(state);
                    i++;
                    root = null;
                }
            }
        }

        /// <summary>
        /// Draws the wire from the output node to the root node of the binary tree.
        /// This shows the visual connection to the tree and the output node to the
        /// user, which makes the diagram more complete.
        /// </summary>
        private void DrawOutputWire()
        {
            var wire = new Wire(c);
            // Setting the start position of the wire as the input of the output pin
            // and the end of the wire as the output point of the last gate in the
            // binary tree.
            wire.SetStart(outputNode.GetLogicGate().GetInputForOutput());
            wire.SetEnd(rootNode.GetLogicGate().GetOutputPoint());
            wire.SetGate(rootNode.GetLogicGate());
            // Setting the coordinates of the wire and drawing it onto the canvas.
            wire.SetPoints();
            wire.RenderLine();
            // Adding the wire to the wire list so that it can be coloured where
            // necessary.
            wires[^1] = wire;
        }

        /// <summary>
        /// Method for drawing the output node onto the diagram canvas. The node is
        /// not drawn within the main BFT as its state is only the state of the
        /// last gate in the diagram and so it can simply be added retroactively.
        /// </summary>
        /// <param name="heightOfTree">The height of the binary tree being drawn.
```

```
        /// This is used when calculating the position of the node.</param>
        private void DrawOutput(int heightOfTree)
        {
            // Assigning the only negative element ID. This means that it can be
            // Identified uniquely.
            outputNode = new Element(-1);
            // Assigning a logic gate as the output is always a visual gate.
            var logicGate = new LogicGate(outputNode);
            outputNode.SetLogicGate(logicGate);
            // Can use inital values to find the position of the root node. An x-offset
            // can simply be added as the y-position is the same.
            double xPosition = TranslateNode(CalculateXposition(0), heightOfTree) +
pixelsPerSquare * 8;
            double yPosition = CalculateNodeYposition(heightOfTree, 0, 0);
            // Setting the position of the logic gate and adding the gate to the canvas.
            Canvas.SetTop(logicGate, yPosition);
            Canvas.SetLeft(logicGate, xPosition);
            Canvas.SetRight(logicGate, xPosition + logicGate.ActualWidth);
            c.Children.Add(logicGate);
        }

        /// <summary>
        /// Simple public method that ties together all of the private method for
        /// diagram drawing. This method generates the tree from the validated
        /// user-entered expression. It also updates the wires into the default
        /// state of zero. This shows the user clearly the diagram is interactive.
        /// </summary>
        public void DrawDiagram()
        {
            canvasWidth = c.ActualWidth;
            GenerateBinaryTreeFromExpression(infixExpression);
            inputMap = GenerateInputMap(infixExpression, false);
            headers = GetHeaders(infixExpression, false);
            outputMap = GenerateOutputMap(infixExpression, headers, false);
            int heightOfTree = GetHeightOfTree(rootNode);
            DrawNodes(rootNode, heightOfTree);
            DrawWires(rootNode);
            DrawOutput(heightOfTree);
            DrawOutputWire();
            UpdateWires();
        }

        #endregion

        #region Truth table generation
        /// <summary>
        /// Evaluates a user-inputted infix boolean expression. This is done by first
        /// converting the expression into postfix and then using a stack.
        /// </summary>
        /// <param name="binaryCombination">A string storing the input combination to be
        /// substituted into the postfix boolean expression. </param>
        /// <param name="inputExpression">The infix expression being evaluated. </param>
        /// <returns>The result of evaluating an infix boolean expression with a particular
        /// string of inputs. </returns>
        private int EvaluateBooleanExpression(string binaryCombination, string inputExpression)
        {
            string postfix = ConvertInfixtoPostfix(inputExpression);
            int operand1;
            int operand2;
            int tmp;
            string substitutedExpression = SubsituteIntoExpression(binaryCombination, postfix);
            var evaluatedStack = new Stack<int>();
```

```
            foreach (char c in substitutedExpression)
            {
                // An operand has been found.
                if (char.IsNumber(c))
                {
                    evaluatedStack.Push(c);
                }
                else if (c == '!')
                {
                    // NOT is an unary operator so only pop one item off of the stack
                    operand1 = evaluatedStack.Pop();
                    // Applying logical is the same as applying an XOR of 1 as the
                    // operand is only every one bit in length.
                    evaluatedStack.Push(operand1 ^ 1);
                }
                else if (c == '^')
                {
                    // Pop two items for the XOR as it is a binary operator.
                    operand1 = evaluatedStack.Pop();
                    operand2 = evaluatedStack.Pop();
                    // Carry out the operation and push the result to the stack.
                    tmp = EvaluateSingleOperator(operand1, operand2, c);
                    evaluatedStack.Push(tmp + 48);
                }
                else
                {
                    // Pop two items for the gate as it is a binary operator.
                    operand1 = evaluatedStack.Pop();
                    operand2 = evaluatedStack.Pop();
                    // Carry out the operation and push the result to the stack.
                    tmp = EvaluateSingleOperator(operand1, operand2, c);
                    evaluatedStack.Push(tmp);
                }
            }
            // The final item on the stack is the result of the evaluation.
            return evaluatedStack.Pop();
        }

        /// <summary>
        /// Simple function that calculates the supplied operation with the suppplied
        /// operands. This exists because of the typing issue between operations and
        /// the char representing the operation being applied. This function is only
        /// used for the binary operators.
        /// </summary>
        /// <param name="o1">The state of the first operand (input).</param>
        /// <param name="o2">That state of the second operand (input). </param>
        /// <param name="operation">The boolean operation being applied to the two
        /// operands. </param>
        /// <returns>The result of the operation with the given operands.</returns>
        private static int EvaluateSingleOperator(int o1, int o2, char operation)
        {
            int result = 0;
            // AND gate.
            if (operation == '.')
            {
                result = o1 & o2;
            }
            // OR gate.
            else if (operation == '+')
            {
                result = o1 | o2;
            }
```

```
            // XOR gate
            else if (operation == '^')
            {
                result = o1 ^ o2;
            }
            return result;
        }

        /// <summary>
        /// Substitutes binary input into a boolean expression. This is for evaluation of a
        /// postfix boolean expression. It should be noted that the position of the bit
corresponds
        /// to the letter it is substituted for. Ie, A => leftmost bit, B=> second bit along.
        /// This means that the inputs within the expressions must be sequential within the
        /// expression.
        /// </summary>
        /// <param name="binaryCombination">The binary combination being substituted into
        /// the boolean expression. </param>
        /// <param name="inputExpression">The boolean expression that the values are being
        /// substituted into. </param>
        /// <returns>The boolean expression with the inputs replaced with the corresponding
        /// bits in the binary combination.</returns>
        private static string SubsituteIntoExpression(string binaryCombination, string
inputExpression)
        {
            string binaryDigit;
            // Tokenising the expression.
            foreach (char c in inputExpression)
            {
                // Token must be an input but not a constant in order for it to be substituted.
                if (char.IsLetter(c))
                {
                    // Finding the correct bit from the index of the letter. This is the reason
                    // for sequential inputs in the expression.
                    binaryDigit = binaryCombination[c - 65].ToString();
                    // Substituting for all of the same inputs.
                    inputExpression = inputExpression.Replace(c.ToString(), binaryDigit);
                }
            }
            // The completely substituted expression.
            return inputExpression;
        }

        /// <summary>
        /// Converts an integer n, into its binay string representation. This is padded to
        /// ensure that it fits into a truth table.
        /// </summary>
        /// <param name="n">The integer being converted into its binary representation.</param>
        /// <param name="booleanExpression">The boolean expression that the binary number
        /// will be used in its truth table. </param>
        /// <param name="isUnique">Boolean value for whether the repeated inputs are being
        /// considered within the count of the number of inputs. </param>
        /// <returns>The binary string representation of an integer n for a truth table.
        /// This is padded to ensure that it fits into the columns of the table.</returns>
        private static string ConvertIntintoBinaryString(int n, string booleanExpression, bool
isUnique)
        {
            // Finding the number of inputs to ensure the correct level of padding is used.
            int numOfInputs = GetNumberOfInputs(booleanExpression, isUnique);
            int remainder;
            string binaryString = "";
            // Repeat until n can no longer be divided two anymore.
```

```
            while (n > 0)
            {
                // Finding the remainder when dividing by two. This is the next most
                // significant bit.
                remainder = n % 2;
                // Divide n by two to get next bit.
                n /= 2;
                // Prepending as the remainder represents the most significant bit.
                binaryString = remainder.ToString() + binaryString;
            }
            // Padding with zero to ensure the correct length.
            return binaryString.PadLeft(numOfInputs, '0');
        }

        /// <summary>
        /// Used when colouring the wires when the diagram is interacted with. This returns
        /// the row of the truth table where the inputs are the same as the state of the
        /// inputs within the logic gate diagram.
        /// </summary>
        /// <returns>The entire row of the truth table based off of the states of the inputs
        /// within the drawn logic gate diagram. </returns>
        private string GetTruthTableRow()
        {
            // The row of the truth table to return is the same as the index within the
            // input map of the same truth table.
            return outputMap[Array.IndexOf(inputMap, inputStates)];
        }

        /// <summary>
        /// Gets the number of inputs within the supplied boolean expression. This count
        /// can either be only the unique inputs counted or any input within the expression.
        /// It is important to note that constants are included in the count.
        /// </summary>
        /// <param name="booleanExpression">A boolean expression entered by the user.</param>
        /// <param name="isUnique">Boolean value, whether or not non-unique are
counted.</param>
        /// <returns>The total number of unique or non-unique inputs within the boolean
expression.
        /// </returns>
        private static int GetNumberOfInputs(string booleanExpression, bool isUnique)
        {
            int numberOfInputs = 0;
            // String to track the inputs that have already been counted in the unique
            // input count.
            string alreadyCounted = "";
            // Tokenising the expression.
            foreach (char token in booleanExpression)
            {
                // Ensuring the token is an input. This either be a letter or a number.
                if (char.IsLetter(token) || char.IsNumber(token))
                {
                    if (isUnique)
                    {
                        // Extra check to see if the input has already been
                        // counted towards the total.
                        if (!alreadyCounted.Contains(token))
                        {
                            alreadyCounted += token;
                            numberOfInputs++;
                        }
                    }
                    else
```

```
                    {
                        numberOfInputs++;
                    }
                }
            }
            return numberOfInputs;
        }

        /// <summary>
        /// Finds the number of operators within a boolean expression.
        /// </summary>
        /// <param name="booleanExpression">The boolean expression being computed.</param>
        /// <returns>The number of boolean operators within a supplied boolean
expression.</returns>
        private int GetNumberOfOperators(string booleanExpression)
        {
            int numberOfOperators = 0;
            // Tokenising the expression.
            foreach (char token in booleanExpression)
            {
                if (booleanOperators.Contains(token))
                {
                    numberOfOperators++;
                }
            }
            return numberOfOperators;
        }

        /// <summary>
        /// Computes all of the input combinations for a truth table. These can be used to
        /// computer the rest of the binary values in the truth table.
        /// </summary>
        /// <param name="inputExpression">The boolean expression being tabulated. </param>
        /// <param name="isUnique">Whether or not the map is for a table with unique input
        /// headers. </param>
        /// <returns>All possible binary input combinations given a boolean
expression.</returns>
        private static string[] GenerateInputMap(string inputExpression, bool isUnique)
        {
            int numberOfInputs = GetNumberOfInputs(inputExpression, isUnique);
            // There is always 2^n different input combinations where n is the number of
            // unique inputs.
            int numberOfRows = (int)Math.Pow(2, numberOfInputs);
            string[] inputMap = new string[numberOfRows];
            for (var i = 0; i < numberOfRows; i++)
            {
                // The input combination is simply the row of the table converted into
                // its respective binary string.
                inputMap[i] = ConvertIntintoBinaryString(i, inputExpression, isUnique);
            }
            return inputMap;
        }

        /// <summary>
        /// Computes the complete set of values for a truth table. This is done by
        /// </summary>
        /// <param name="inputExpression">The boolean expression being tabulated. </param>
        /// <param name="headers">The headers of the truth table. </param>
        /// <param name="isUnique">Whether the table has unique inputs or not. This
        /// decides whether or not the table is being drawn or used for diagram interactivity.
        /// </param>
        /// <returns>The complete table of values for the truth table. </returns>
```

```
        private string[] GenerateOutputMap(string inputExpression, string[] headers, bool
isUnique)
        {
            // Generating all of the different input combinations for the truth table.
            // These will be substituted into the boolean expression.
            string[] inputMap = GenerateInputMap(inputExpression, isUnique);
            int numberOfRows = inputMap.Length;
            string inputCombination;
            string[] outputMap = new string[inputMap.Length];
            for (var i = 0; i < numberOfRows; i++)
            {
                // Getting the input combination for the truth table.
                inputCombination = inputMap[i];
                // Computing the row of the truth table.
                outputMap[i] += GetOutputRow(headers, inputCombination);
            }
            return outputMap;
        }

        /// <summary>
        /// Computes a single row of the truth table based off of the headers of the
        /// truth table and a supplied binary input combination such as "0100" for a
        /// four input truth table.
        /// </summary>
        /// <param name="headers">The headers of the truth table. This is used to
        /// determine a particular value in the column of the table. </param>
        /// <param name="inputCombination">The binary representation of the row
        /// number in the truth table. </param>
        /// <returns>A single row of the truth table. </returns>
        private string GetOutputRow(string[] headers, string inputCombination)
        {
            string outputRow = "";
            // Evaluating each header to get the complete row. Each character in the
            // string is a column of the truth table.
            foreach (string header in headers)
            {
                // Subtracting 48 to convert the character into an integer value.
                outputRow += EvaluateBooleanExpression(inputCombination, header) - 48;
            }
            return outputRow;
        }

        /// <summary>
        /// Utility function to generate headers for boolean expressions.
        /// </summary>
        /// <param name="inputExpression">The boolean expression that the headers
        /// are generated from. </param>
        /// <param name="isDisplay">Whether or not the headers will be displayed to the user.
</param>
        /// <returns>The headers from the supplied boolean expression. </returns>
        private string[] GetHeaders(string inputExpression, bool isDisplay)
        {
            // Headers are always generated from the postfix expression.
            string postfix = ConvertInfixtoPostfix(inputExpression);
            string[] headers;
            int numberOfInputs = GetNumberOfInputs(postfix, false);
            int numberOfOperators = GetNumberOfOperators(postfix);
            if (isDisplay)
            {
                headers = GenerateDisplayOperatorHeaders(postfix, numberOfInputs,
numberOfOperators);
            }
```

```
            else
            {
                headers = GeneratePostOrderHeaders(postfix, numberOfInputs, numberOfOperators);
            }
            return headers;
        }

        /// <summary>
        /// Creates the headers for the truth table that is drawn onto the truth table canvas.
        /// </summary>
        /// <param name="inputExpression">The postfix expression that the headers
represent.</param>
        /// <param name="numberOfInputs">The number of inputs within the postfix
expression.</param>
        /// <param name="numberOfOperators">The number of operators within the postfix
expression.</param>
        /// <returns>The truth table headers that are displayed to the user. </returns>
        private static string[] GenerateDisplayOperatorHeaders(string inputExpression, int
numberOfInputs, int numberOfOperators)
        {
            // Display headers are ultimately a subset of the postorder headers.
            string[] postorderHeaders = GeneratePostOrderHeaders(inputExpression,
numberOfInputs, numberOfOperators);
            // Sorting into alphabetical order.
            Array.Sort(postorderHeaders);
            // Sorting the headers by length. This makes the simplest sub-expressions first.
            // These are always the shorter sub-expressions as they are lower down the tree.
            Array.Sort(postorderHeaders, (x, y) => x.Length.CompareTo(y.Length));
            postorderHeaders = postorderHeaders.Distinct().ToArray();
            // Filtering out the duplicate subexpressions and inputs as they are always the
            // same within the table.
            return postorderHeaders;
        }

        /// <summary>
        /// Generates headers for colouring the wires when the user clicks on the inputs.
        /// These represent the subexpression at a certain point within the tree. This is
        /// also, the postorder traversal of the tree. This is done by evaluating the
expression.
        /// </summary>
        /// <param name="postfix">The postfix expression that the header will
represent.</param>
        /// <param name="numberOfInputs">The number of non-unique inputs within the given
boolean expression. </param>
        /// <param name="numberOfOperators">The number of operators within the given boolean
expression. </param>
        /// <returns>An array representing the postorder headers of a postfix boolean
expression. </returns>
        private static string[] GeneratePostOrderHeaders(string postfix, int numberOfInputs,
int numberOfOperators)
        {
            var subExpressionStack = new Stack<string>();
            // The number of headers is always the number of non-unique inputs + the number
            // of operators as this is the number of nodes in the tree.
            string[] headers = new string[numberOfOperators + numberOfInputs];
            string subexpression;
            string operand1;
            string operand2;
            int i = 0;
            // Tokenising the expression.
            foreach (char c in postfix)
            {
```

```
                // When an operand is found then pop it and add it to the header array as
                // An input is a header in itself.
                if (char.IsLetter(c) || char.IsNumber(c))
                {
                    subExpressionStack.Push(c.ToString());
                    headers[i] = c.ToString();
                    i++;
                }
                // An operator has been found.
                else
                {
                    if (c == '!')
                    {
                        // As a NOT gate is a unary operator, pop one item off of the stack.
                        operand1 = subExpressionStack.Pop();
                        subexpression = $"({c}{operand1})";
                    }
                    else
                    {
                        // All other operators are binary operators so pop two items off of
                        // the stack and push the result pack onto the stack.
                        operand1 = subExpressionStack.Pop();
                        operand2 = subExpressionStack.Pop();
                        // Add brackets for the sub-expression and for the next headers.
                        subexpression = $"({operand2}{c}{operand1})";
                    }
                    // Pushing the result pack onto the stack and add to the headers array
                    // as a subexpression is a header in itself.
                    subExpressionStack.Push(subexpression);
                    headers[i] = subexpression;
                    i++;
                }
            }
            return headers;
        }

        /// <summary>
        /// Small method that simplify calculates the width of cell in the truth table
        /// based of the header.
        /// </summary>
        /// <param name="header">The header the cell is in the column of. </param>
        /// <returns>The width of the truth table based off of the column the cell is in.
        /// The is based off of the header. </returns>
        private double CalculateCellWidth(string header)
        {
            // Default size of the truth table cell.
            double cellWidth = minCellWidth;
            // If the header is not an input.
            if (header.Length != 1)
            {
                // If the header is an expression larger than (A.B).
                if (header.Length > 5)
                {
                    cellWidth = header.Length * 11 + 15;
                }
                else
                {
                    cellWidth = header.Length * 10 + 15;
                }
            }
            return cellWidth;
        }
```

```
        /// <summary>
        /// Draws the headers for the truth table.
        /// </summary>
        /// <param name="c">The canvas being drawn to. </param>
        /// <param name="headers">The headers of the truth table. These represent the stages
        /// of the post order traversal of the logic gate diagram. </param>
        private void DrawTruthTableHeaders(Canvas c, string[] headers)
        {
            Label cell;
            var border = new Thickness(2);
            var font = new FontFamily("Consolas");
            double cellWidth;
            // Setting the initial postion of the table on the canvas.
            double xPosition = initialXofTable;
            foreach (string header in headers)
            {
                // Calculating the width of the cell based off of the header.
                cellWidth = CalculateCellWidth(header);
                cell = new Label
                {
                    // Defining the style of the cell.
                    HorizontalContentAlignment = HorizontalAlignment.Center,
                    Width = cellWidth,
                    BorderBrush = Brushes.LightGray,
                    BorderThickness = border,
                    Background = Brushes.White,
                    FontFamily = font,
                    FontSize = 18,
                    Content = header
                };
                // Adding the cell to the canvas and incrementing to get the position
                // of the next cell.
                Canvas.SetTop(cell, initialYofTable);
                Canvas.SetLeft(cell, xPosition);
                c.Children.Add(cell);
                xPosition += cellWidth;
            }
            // Readjusting the x size of the canvas so that the scroll viewer works.
            c.Width = Math.Max(xPosition, c.Width) + 30;
        }

        /// <summary>
        /// Draws the body of the truth table. This is the inputs and outputs at each stage
        /// of the table.
        /// </summary>
        /// <param name="c">The canvas being drawn to. </param>
        /// <param name="headers">The headers of the table. Used to calculate the cell widths.
</param>
        /// <param name="outputMap">The data used to fill the cells of the table. </param>
        private void DrawTruthTableBody(Canvas c, string[] headers, string[] outputMap)
        {
            Label cell;
            var border = new Thickness(2);
            var font = new FontFamily("Consolas");
            double cellWidth;
            // Initial position of the body of the truth table.
            double xPosition = initialXofTable;
            double yPosition = 50;
            foreach (string row in outputMap)
            {
                for (var i = 0; i < headers.Length; i++)
```

```
            {
                // Calculating the width of the cell based off of the size of the header.
                cellWidth = CalculateCellWidth(headers[i]);
                // Defining the style of the table cell.
                cell = new Label
                {
                    // Defining the style of the cell in the truth table.
                    HorizontalContentAlignment = HorizontalAlignment.Center,
                    Width = cellWidth,
                    BorderBrush = Brushes.LightGray,
                    BorderThickness = border,
                    Background = Brushes.White,
                    FontFamily = font,
                    FontSize = 18,
                    Content = row[i]
                };
                // Adding the cell to the canvas and incrementing the position of the x
                // to get to the next position.
                Canvas.SetTop(cell, yPosition);
                Canvas.SetLeft(cell, xPosition);
                c.Children.Add(cell);
                xPosition += cellWidth;
            }
            // Resetting the x-position on the canvas, so that the table is aligned to
            // the left side of the canvas.
            xPosition = 20;
            // Incrementing the y-position on the canvas and so, go to the next row of
            // the table.
            yPosition += 30;
        }
        // Readjusting the size of the canvas so that the scrollviewer works with
        // large tables. This makes extremely large tables very easy to view.
        c.Height = Math.Max(yPosition, c.Height) + 30;
    }

    /// <summary>
    /// Removes the outer most set of brackets from the headers. This reduces the total
    /// width of each header making the table more manageable to look through. It also
    /// makes the headings clearer as there is less going on.
    /// </summary>
    /// <param name="headers"></param>
    /// <returns></returns>
    private static string[] TrimBrackets(string[] headers)
    {
        for (var i = 0; i < headers.Length; i++)
        {
            // Ensuring that the header does not represent an input which does not have a
            // set of brackets.
            if (headers[i].Length != 1)
            {
                // Slicing off the first and last characters which are the brackets.
                headers[i] = headers[i][1.^1];
            }
        }
        return headers;
    }

    /// <summary>
    /// Simple public method that allows the diagram class to draw to a canvas on the
    /// user interface.
    /// </summary>
    /// <param name="c">The canvas that the truth table will be drawn on. </param>
```

```
        /// <param name="inputExpression">The boolean expression for which the truth table
        /// represents. </param>
        public void DrawTruthTable(Canvas c, string inputExpression)
        {
            c.Children.Clear();
            headers = GetHeaders(inputExpression, true);
            outputMap = GenerateOutputMap(inputExpression, headers, true);
            headers = TrimBrackets(headers);
            DrawTruthTableHeaders(c, headers);
            DrawTruthTableBody(c, headers, outputMap);
        }
        #endregion

        #region Minimisation
        /// <summary>
        /// Produces the final minimised expression is Petrick's method is not used.
        /// </summary>
        /// <param name="essentialPrimeImplicants">The essential prime implicants found by the
program.</param>
        /// <returns>The final minimised expression. </returns>
        private string ConvertEPIsToExpression(List<string> essentialPrimeImplicants)
        {
            // Connverting all of the essential prime implicants that have been found and
            // producing a final expression. Each implicant is separated by an OR gate in
            // the final expression.
            essentialPrimeImplicants = essentialPrimeImplicants.ConvertAll(new
Converter<string, string>(ConvertImplicantToExpression));
            string expression = string.Join("+", essentialPrimeImplicants);
            return expression;
        }

        /// <summary>
        /// Converts the binary form (such as -100) into terms of an expression such as
        /// (B!C!D) in this example. This is used for outputting minimised expressions when
        /// they need to be converted into their final form.
        /// </summary>
        /// <param name="epi">The prime implicant being converted. This is usually and
essential
        /// one. </param>
        /// <returns>The expression form of the prime implicant. </returns>
        private string ConvertImplicantToExpression(string epi)
        {
            string tmp = "";
            char input;
            for (var i = 0; i < epi.Length; i++)
            {
                input = (char)(i + 65);
                // If the bit is a one then the output must be in the expression.
                if (epi[i] == '1')
                {
                    tmp += input;
                }
                // If the bit is a zero then the complement is added to the expression => !A.
                else if (epi[i] == '0')
                {
                    tmp += $"!{input}";
                }
                // Each bit within the implicant is separated by an AND gate within the
                // expression. AND gates should not be added at the end or the start of the
                // implicant.
                if (epi.Length == 2)
                {
```

```
                    if (i == 0 && epi[i] != '-')
                    {
                        tmp += ".";
                    }
                }
                else
                {
                    if (i != 0 && i < epi.Length - 1 && epi[i] != '-')
                    {
                        tmp += ".";
                    }
                }
            }
            // Add brackets to preserve pretty formatting.
            return $"({tmp})";
        }

        /// <summary>
        /// Creates the minterm coverage strings for the prime implicants. These string show
        /// which prime implicants cover which minterms. This is used in finding the essential
        /// prime implicants and also, they are used in Petricks method for finding the product
of sums.
        /// </summary>
        /// <param name="regex">The prime implicant chart being filled.</param>
        /// <param name="minterms">The minterms for the user entered boolean
expression.</param>
        private static void SetRegexPatterns(Dictionary<string, string> regex, List<string>
minterms)
        {
            Match res;
            foreach (string regexPattern in regex.Keys.ToList())
            {
                foreach (string minterm in minterms)
                {
                    // If minterm matches the form of the prime implicants, shown by Regex,
                    // then a one can be written to the coverage string showing that the
                    // implicant covers this minterm.
                    res = Regex.Match(minterm, regexPattern);
                    if (res.Success)
                    {
                        regex[regexPattern] += "1";
                    }
                    //The implicant does not cover this minterm.
                    else
                    {
                        regex[regexPattern] += "0";
                    }
                }
            }
        }

        /// <summary>
        /// Prepares the prime implicant so that the minterm coverage strings can be created
        /// from the regex. This is also the point in which the keys are added to the
dictionary.
        /// </summary>
        /// <param name="regex">The empty prime implicant chart. </param>
        /// <param name="primeImplicants">The prime implicants that have been found from
merging
        /// minterms. </param>
        private static void ConvertImplicantsIntoRegex(Dictionary<string, string> regex,
List<string> primeImplicants)
```

```
        {
            string tmp = "";
            string value = "";
            foreach (string primeImplicant in primeImplicants)
            {
                // Replacing any dashes with \d as to indicate that the character in that
                // position does not matter. Otherwise, the bit must remain.
                foreach (char c in primeImplicant)
                {
                    if (c == '-')
                    {
                        tmp += @"\d";
                    }
                    else
                    {
                        tmp += c;
                    }
                }
                // Adding the key and empty value to the dictionary. The value will be
                // populated when the regex is carried out on the minterms.
                regex.Add(tmp, value);
                tmp = "";
            }
        }

        /// <summary>
        /// Merges two minterms when finding the prime implicants. This is because a dash is
        /// added in place of the differing bit when the merge has been made.
        /// </summary>
        /// <param name="minterm1">One of the minterms being merged.</param>
        /// <param name="minterm2">One of the minterms being merged.</param>
        /// <returns>The merged minterm with the dashes present.</returns>
        /// <exception cref="Exception">The minterms are not of the same length and so cannot
        /// be merged to produce a valid result. </exception>
        private static string MergeMinterms(string minterm1, string minterm2)
        {
            string mergedMinterm = "";
            if (minterm1.Length != minterm2.Length)
            {
                throw new Exception("Incorrect length");
            }
            else
            {
                for (var i = 0; i < minterm1.Length; i++)
                {
                    // If the bit differs then replace it with a dash.
                    // Otherwise just add the bit from one of the minterms as it is the same.
                    if (minterm1[i] != minterm2[i])
                    {
                        mergedMinterm += '-';
                    }
                    else
                    {
                        mergedMinterm += minterm1[i];
                    }
                }
                return mergedMinterm;
            }
        }

        /// <summary>
        /// For a merge to take place when finding the prime implicants, the dashes in both
```

```
        /// of the minterms must align.
        /// </summary>
        /// <param name="minterm1">A minterm. </param>
        /// <param name="minterm2">The other minterm being checked with. </param>
        /// <returns>Whether or not the dashes within two minterms are in the same
position.</returns>
        /// <exception cref="Exception">If the minterms are of different lengths then the
dashes
        /// cannot be checked as you cannot iterate through one of strings and check both
minterms. </exception>
        private static bool CheckDashesAlign(string minterm1, string minterm2)
        {
            if (minterm1.Length != minterm2.Length)
            {
                throw new Exception("Incorrect length");
            }
            else
            {
                for (var i = 0; i < minterm1.Length; i++)
                {
                    // If one of the minterms is a dash and the other is then the dashes
                    // do not align and so the minterms cannot be merged together.
                    if (minterm1[i] != '-' && minterm2[i] == '-')
                    {
                        return false;
                    }
                }
                // Dashes must align and so this condition has been met.
                return true;
            }
        }

        /// <summary>
        /// A merge can only take place if only one bit differs between the two minterms.
        /// </summary>
        private static bool CheckMintermDifference(string m1, string m2)
        {
            // Removing the dashes so that bitwise operators can be used.
            int minterm1 = RemoveDashes(m1);
            int minterm2 = RemoveDashes(m2);
            // XOR identifies any bits that differ between the two minterms.
            int res = minterm1 ^ minterm2;
            // If res != 0, then one bit could differ. This checked with the AND.
            return res != 0 && (res & res - 1) == 0;
        }

        /// <summary>
        /// Converts the minterms' dashes into zeros so that the bit difference can
        /// be easily checked.
        /// </summary>
        private static int RemoveDashes(string minterm)
        {
            return Convert.ToInt32(minterm.Replace('-', '0'), 2);
        }

        /// <summary>
        /// Finds all of the minterms of the user-entered boolean expression. A minterm is
        /// any binary input combination that results in the expression evaluating to 1.
        /// </summary>
        /// <param name="expression">The user-entered boolean expression being
minimised.</param>
        /// <returns>The list of minterms of the boolean expression.</returns>
```

```
        private List<string> GetMinterms(string expression)
        {
            var minterms = new List<string>();
            int result;
            inputMap = GenerateInputMap(expression, true);
            foreach (string input in inputMap)
            {
                // Trying the evaluation and seeing if the result is a minterm.
                result = EvaluateBooleanExpression(input, expression) - 48;
                // All minterms evaluate to 1.
                if (result == 1)
                {
                    minterms.Add(input);
                }
            }
            return minterms;
        }

        /// <summary>
        /// The number prime implicants that cover a particular minterm within the prime
        /// implicant chart. This can be used to find the essential prime implicants.
        /// </summary>
        /// <param name="regex">The prime implicant chart of the boolean expression</param>
        /// <param name="minterms">The binary combinations that result in the expression
        /// evaluating to true. </param>
        /// <returns>The number of prime implicants that cover each minterm. </returns>
        private static int[] GetFrequencyTable(Dictionary<string, string> regex, List<string>
minterms)
        {
            int[] mintermCoverage = new int[minterms.Count];
            foreach (string s in regex.Values.ToList())
            {
                for (var i = 0; i < s.Length; i++)
                {
                    if (s[i] == '1')
                    {
                        mintermCoverage[i]++;
                    }
                }
            }
            return mintermCoverage;
        }

        /// <summary>
        /// Performs the column search within the prime implicant chart, this is to
        /// search for an essential prime implicant.
        /// </summary>
        /// <param name="regex">The prime implicant chart. </param>
        /// <param name="pos">The column (minterm) that we are checking within the
chart.</param>
        /// <returns>The only prime implicant that covers that minterm within the
chart.</returns>
        /// <exception cref="Exception">No prime implicant covers that minterm and so the
frequency
        /// table has been incorrectly calculated. </exception>
        private static string GetEssentialPrimeImplicant(Dictionary<string, string> regex, int
pos)
        {
            string[] implicantCoverage = regex.Values.ToArray();
            string[] implicants = regex.Keys.ToArray();
            string prime;
            // Iterating through each of the prime implicants.
```

```
            for (var i = 0; i < implicantCoverage.Length; i++)
            {
                // Getting the minterm coverage for the prime implicant.
                prime = implicantCoverage[i];
                // If the specified column is a 1 then the essential prime implicant has been
found.
                if (prime[pos] == '1')
                {
                    return implicants[i];
                }
            }
            throw new Exception("Item could be found");
        }

        /// <summary>
        /// Filters through the coverages of each of the prime implicants searching for the
        /// essential prime implicants. These are the prime implicants that are the only
        /// implicant to cover a minterm.
        /// </summary>
        /// <param name="regex">The prime implicant chart produced from the regex and
        /// the prime implicants. </param>
        /// <param name="minterms">The binary combinations that result in the expression
        /// evaluating to 1. </param>
        /// <returns>Every prime implicant within the prime implicant chart that is essential/
</returns>
        private static List<string> GetEssentialPrimeImplicants(Dictionary<string, string>
regex, List<string> minterms)
        {
            // Find the number of ones within each column of the prime implicant chart.
            int[] bitFrequencyTable = GetFrequencyTable(regex, minterms);
            var essentialPrimeImplicants = new List<string>();
            string epi;
            for (var i = 0; i < bitFrequencyTable.Length; i++)
            {
                // This means that there is only one implicant in this column that covers it
                // and so, an essential prime implicant has been found. Iterate through the
                // column to find the implicant with the 1 within that column.
                if (bitFrequencyTable[i] == 1)
                {
                    // Do the column search to find the prime implicant, which must
                    // be essential.
                    epi = GetEssentialPrimeImplicant(regex, i);
                    if (!essentialPrimeImplicants.Contains(epi))
                    {
                        essentialPrimeImplicants.Add(epi);
                    }
                }
            }
            return essentialPrimeImplicants;
        }

        /// <summary>
        /// Finds the prime implicants of the boolean expression. This comes from the minterms
        /// of the boolean expression. This is done using a recursive merging process where
        /// merges take place. Once no merges have taken place on the prim implicants then they
        /// all have been found and the method ends.
        /// </summary>
        /// <param name="mintermList">The binary input combinations that result in the
        /// boolean expression evaluating to one. </param>
        /// <returns>The list of prime implicants of the expression. </returns>
        private static List<string> GetPrimeImplicants(List<string> mintermList)
        {
```

```
            var primeImplicants = new List<string>();
            bool[] merges = new bool[mintermList.Count];
            int numberOfMerges = 0;
            string mergedMinterm;
            string minterm1;
            string minterm2;
            for (var i = 0; i < mintermList.Count; i++)
            {
                for (var c = i + 1; c < mintermList.Count; c++)
                {
                    minterm1 = mintermList[i];
                    minterm2 = mintermList[c];
                    // A merge can be made only if the dashes of the minterms align and
                    // there is only one bit of difference between the two minterms.
                    if (CheckDashesAlign(minterm1, minterm2) &&
CheckMintermDifference(minterm1, minterm2))
                    {
                        mergedMinterm = MergeMinterms(minterm1, minterm2);
                        primeImplicants.Add(mergedMinterm);
                        numberOfMerges++;
                        // Mark the terms that have been merged so that they do not persist
                        // to the next stage of the merging process.
                        merges[i] = true;
                        merges[c] = true;
                    }
                }
            }
            // Filtering out the terms that have been merged so that the minterms do not
            // remain when the next stage occurs.
            for (var j = 0; j < mintermList.Count; j++)
            {
                if (!merges[j] && !primeImplicants.Contains(mintermList[j]))
                {
                    primeImplicants.Add(mintermList[j]);
                }
            }
            // If no more merges can be made on the list of implicants then all of the prime
            // implicants for the expression must have been found. Otherwise, recurse and try
            // merging again.
            if (numberOfMerges == 0)
            {
                return primeImplicants;
            }
            else
            {
                return GetPrimeImplicants(primeImplicants);
            }
        }

        /// <summary>
        /// Minimises a user entered boolean expression using the Quine-McCluskey algorithm and
        /// Petrick's method. Details of the algorithms can be found on Wikipedia.
        /// </summary>
        /// <param name="expression">A user entered boolean expression.</param>
        public void MinimiseExpression(string expression)
        {
            // The input combinations that result in the expression evaluating to one.
            List<string> minterms = GetMinterms(expression);
            List<string> primeImplicants = GetPrimeImplicants(minterms);
            var PIchart = new Dictionary<string, string>();
            ConvertImplicantsIntoRegex(PIchart, primeImplicants);
            // Creating the coverages for each of the prime implicants using regex.
```

```
            SetRegexPatterns(PIchart, minterms);
            PIchart = ReplaceDashesFromRegex(PIchart);
            List<string> PIs = GetEssentialPrimeImplicants(PIchart, minterms);
            string coveredString = GetCoveredString(PIs, PIchart);
            // If a 0 remains in the covered string, then the essential prime implicants
            // do not cover all of the minterms and so petrick's method must be used.
            if (coveredString.Contains('0'))
            {
                minimisedExpression = DoPetriksMethod(PIchart, PIs, primeImplicants, minterms);
            }
            else
            {
                minimisedExpression = ConvertEPIsToExpression(PIs);
            }
        }

        /// <summary>
        /// Replaces the regex characters in the prime implicants with dashes. This will
        /// make processing and outputting the prime implicants easier.
        /// </summary>
        /// <param name="PIChart">The prime implicant chart produced by QM.</param>
        /// <returns>The PIChart with \d replaced for each of the keys. </returns>
        private static Dictionary<string, string> ReplaceDashesFromRegex(Dictionary<string,
string> PIChart)
        {
            var newKeys = new Dictionary<string, string>();
            string tmp;
            foreach (string k in PIChart.Keys)
            {
                tmp = k;
                tmp = tmp.Replace(@"\d", "-");
                newKeys.Add(tmp, PIChart[k]);
            }
            return newKeys;
        }

        /// <summary>
        /// Checks to make sure that the essential prime implicants cover the original boolean
        /// expression. Otherwise, Petrick's method should be used to make sure that all of the
        /// minterms are covered by the implicants.
        /// </summary>
        /// <param name="epis">The essential prime implicants currently found. </param>
        /// <param name="PIchart">The prime implicant produced from the prime
implicants.</param>
        /// <returns></returns>
        private static string GetCoveredString(List<string> epis, Dictionary<string, string>
PIchart)
        {
            int coveredString = 0;
            foreach (string s in epis)
            {
                // Apply logical OR to each of the coverages of the essential prime implicants.
                // This is to ensure at least one 1 covers a minterm.
                coveredString |= Convert.ToInt32(PIchart[s], 2);
            }
            return coveredString.ToString();
        }

        /// <summary>
        /// Carries out the process to prepare the PI chart for Petricks' method. The first
        /// step is to remove any rows of prime implicants. The minterm that makes the
        /// implicant essential should also be removed.
```

```
        /// </summary>
        /// <param name="PIchart">The prime implicant chart being reduced.</param>
        /// <param name="epis">The essential prime implicants of the expression. </param>
        /// <param name="minterms">The minterms covered by the expression. </param>
        /// <returns>The updated prime implicant chart with essential prime implicants
removed.</returns>
        private static Dictionary<string, string> RemoveEPIs(Dictionary<string, string>
PIchart, List<string> epis, List<string> minterms)
        {
            string value;
            int bit;
            // The number of ones that cover each minterm.
            int[] freq = GetFrequencyTable(PIchart, minterms);
            foreach (string implicant in PIchart.Keys)
            {
                // A prime implicant has been found.
                if (epis.Contains(implicant))
                {
                    value = PIchart[implicant];
                    // Finding the position of the bit that makes the implicant essential.
                    // This is the column that being removed.
                    bit = GetSignificantBit(implicant, freq);
                    // Removing the column of the prime implicant chart.
                    TrimMinterm(PIchart, bit);
                    PIchart.Remove(implicant);
                }
            }
            return PIchart;
        }

        /// <summary>
        /// Removes the bit of a specified column when an essential prime implicant has been
        /// found. This is for chart reduction in Petrick's method.
        /// </summary>
        /// <param name="PIchart">The prime implicant chart produced by QM.</param>
        /// <param name="pos">The column of the prime implicant chart being removed.</param>
        /// <returns>The updated dictionary with the column removed. </returns>
        private static Dictionary<string, string> TrimMinterm(Dictionary<string, string>
PIchart, int pos)
        {
            string value;
            foreach (string s in PIchart.Keys)
            {
                value = PIchart[s];
                value = value.Remove(pos, 1);
                PIchart[s] = value;
            }
            return PIchart;
        }

        /// <summary>
        /// Returns the position of the bit that makes the prime implicant essential.
        /// </summary>
        /// <param name="epi">The prime implicant being check for. </param>
        /// <param name="freq">The number of times a minterm is covered by the
implicant.</param>
        /// <returns>The position within the coverage that makes the implicant essential.
</returns>
        private static int GetSignificantBit(string epi, int[] freq)
        {
            for (var i = 0; i < freq.Length; i++)
            {
```

```
                if (freq[i] == 1 && epi[i] == '1')
                {
                    return i;
                }
            }
            return -1;
        }

        /// <summary>
        /// Method that carries out the stepwise process of Petricks's method. The process
        /// the algorithm is defined on the methods' Wikipedia page.
        /// </summary>
        /// <param name="PIchart">The prime implicant chart found during QM. </param>
        /// <param name="epis">The essential prime implicants found in the PIchart.</param>
        /// <param name="primeImplicants">All of the prime implicants found during
        /// the initial merging process of QM.</param>
        /// <param name="minterms">The minterm of </param>
        /// <returns>The minimised expression produced by Petrick's method.</returns>
        private string DoPetriksMethod(Dictionary<string, string> PIchart, List<string> epis,
List<string> primeImplicants, List<string> minterms)
        {
            PIchart = RemoveEPIs(PIchart, epis, minterms);
            string minimisedExpression;
            Dictionary<char, string> termsImplicantMapping =
MapTermsToImplicants(primeImplicants);
            List<Bracket> productOfSums = GetProductOfSums(termsImplicantMapping, PIchart);
            string[] sumOfproducts = GetSumOfProducts(productOfSums);
            string minProduct = GetMinProduct(sumOfproducts);
            minimisedExpression = GetFinalExpression(termsImplicantMapping, minProduct);
            return minimisedExpression;
        }

        /// <summary>
        /// Creates the relationship between terms and prime implicants. This makes
        /// the creation of product of sums and sum of products much clearer as the
        /// prime implicants are abstracted to a single letter. It also allows for
        /// multiple string of prime implicants to represented in a concise manner.
        /// </summary>
        /// <param name="primeImplicants">The prime implicants found by the initial
        /// merge of the minterms of the boolean expression.</param>
        /// <returns>The relationship between terms and prime implicants as a
        /// dictionary. </returns>
        private static Dictionary<char, string> MapTermsToImplicants(List<string>
primeImplicants)
        {
            var mapping = new Dictionary<char, string>();
            // To ensure that unique characters are used as the terms. Take the largest
            // ascii value (highest inputs) and convert it to the next character and
            // this is the first character in the map.
            char minChar = (char)(primeImplicants[0].Length + 65);
            for (var i = 0; i < primeImplicants.Count; i++)
            {
                mapping.Add(minChar, primeImplicants[i]);
                // Incrementing so that the inputs within the map are sequential.
                minChar++;
            }
            return mapping;

        }

        /// <summary>
        /// Produces the product of sums of the prime implicant chart. This stage prepares
```

```
        /// the data so that the boolean algebra can be done on the expression.
        /// </summary>
        /// <param name="termToImplicantMap">The relationship between terms and
        /// prime implicants. </param>
        /// <param name="primeImplicantChart">The relationship between minterm coverage
        /// and the prime implicants found in the intial process of QM. </param>
        /// <returns> The product of sums of the PI chart, in the form [('K', 'L'),
etc.]</returns>
        private static List<Bracket> GetProductOfSums(Dictionary<char, string>
termToImplicantMap, Dictionary<string, string> primeImplicantChart)
        {
            var productOfSums = new List<Bracket>();
            List<Bracket> sumsToAdd;
            string primeImplicant;
            // A sum can be made if the minterm is covered by two implicants.
            // So, loop through each key to find its coverage of the minterms
            foreach (string key in primeImplicantChart.Keys)
            {
                primeImplicant = primeImplicantChart[key];
                // Iterate through each minterm.
                for (var i = 0; i < primeImplicant.Length; i++)
                {
                    // If the prime implicant covers the minterm then a possible sum
                    // could be found so search through the chart vertically.
                    if (primeImplicant[i] == '1')
                    {
                        // Get all of the possible sums within the column of the covered
                        // minterm and add them to the found sums.
                        sumsToAdd = GetSumsToAdd(primeImplicantChart, termToImplicantMap, key,
i);
                        AddSumsToList(productOfSums, sumsToAdd);
                    }
                }
            }
            return productOfSums;
        }

        /// <summary>
        /// Ensures that duplicate sums are not added the list as X.X = X and so they cancel
        /// off.
        /// </summary>
        /// <param name="productOfSums">The sums that have currently been found during
        /// the iteration through the prime implicant chart. </param>
        /// <param name="sumsToAdd">The sums that have found by searching through the
        /// column of the covered minterm. </param>
        private static void AddSumsToList(List<Bracket> productOfSums, List<Bracket> sumsToAdd)
        {
            foreach (Bracket s in sumsToAdd)
            {
                if (productOfSums.Contains(s) == false)
                {
                    productOfSums.Add(s);
                }
            }
        }

        /// <summary>
        /// Gets all of the possible sums that are within a column of the prime implicant
        /// chart. This is done by searching through and if another implicant covers the
        /// same minterm then a sum can be made.
        /// </summary>
        /// <param name="PIchart">The relationship between the prime implicants and
```

```
        /// the minterms that they cover. </param>
        /// <param name="termToImplicantMap">The relationship between terms in the boolean
        /// algebra and the prime implicants that they represent.</param>
        /// <param name="key">The prime implicant that caused the column search</param>
        /// <param name="positionWithinKey">The minterm that the prime implicant must
        /// cover in order for a sum to be created. It must not be the same implicant. </param>
        /// <returns>All of the possible sums for a prime implicant that covers a minterm.
</returns>
        private static List<Bracket> GetSumsToAdd(Dictionary<string, string> PIchart,
Dictionary<char, string> termToImplicantMap, string key, int positionWithinKey)
        {
            var sumsToAdd = new List<Bracket>();
            Bracket sum;
            string implicant;
            char term1;
            char term2;
            // Iterate through each prime implicant to try and find as many sums as possible.
            for (var i = 0; i < PIchart.Keys.Count; i++)
            {
                // The prime implicant that could make a sum.
                implicant = PIchart.Keys.ToArray()[i];
                // If the implicant covers the same minterm then a sum has been found and a
                // bracket can be created.
                if (PIchart[implicant][positionWithinKey] == '1')
                {
                    // Getting the letters that represent a certain prime implicant.
                    term1 = GetTermFromImplicant(termToImplicantMap, key);
                    term2 = GetTermFromImplicant(termToImplicantMap, implicant);
                    // Ensuring brackets are not made with duplicate term.
                    if (term1 != term2)
                    {
                        // A new sum is created and added to the list.
                        sum = new Bracket(term1, term2);
                        sumsToAdd.Add(sum);
                    }
                }
            }
            return sumsToAdd;
        }

        /// <summary>
        /// Searches and return the given term for a prime implicant, based off of the map
        /// created by the program.
        /// </summary>
        /// <param name="termToImplicantMap">Assignment of terms(letters) that represent
        /// prime implicants. This simplification easier to deal with. </param>
        /// <param name="implicant">The prime implicant being searched for. </param>
        /// <returns>The term that is the key to the respective prime implicant.</returns>
        /// <exception cref="Exception">The implicant that is being searched for could not
        /// be found within the termToImplicantMap </exception>
        private static char GetTermFromImplicant(Dictionary<char, string> termToImplicantMap,
string implicant)
        {
            // All of the prime implicants that are within the prime implicant.
            string[] implicants = termToImplicantMap.Values.ToArray();
            // The letter that represents the prime implicant within the simplification.
            char[] keys = termToImplicantMap.Keys.ToArray();
            for (var i = 0; i < termToImplicantMap.Values.Count; i++)
            {
                if (implicants[i] == implicant)
                {
                    // The term based off of the implicant.
```

```
                    return keys[i];
                }
            }
            throw new Exception("Could not map implicant to key");
        }

        /// <summary>
        /// Converts the product of sums (A+B)(C+D)... into the sum of products of the boolean
        /// expression. sum of products = AB + CD + etc. This allows the program to find the
        /// minimal term(solution) which can ultimately give the minimised expression.
        /// </summary>
        /// <param name="productOfSums">The product of sums found in the prime implicant
        /// chart. </param>
        /// <returns> The sum of products of the expression which is used to minimise the
input</returns>
        private static string[] GetSumOfProducts(List<Bracket> productOfSums)
        {
            Bracket b1;
            Bracket b2;
            Bracket mergedTerm;
            bool merged = true;
            // Keep trying to merge the brackets together until no more merges can be made.
            // This means that the distributive law can be applied to the brackets at that
point.
            while (merged)
            {
                merged = false;
                // Checking each sum to ensure the most merges take place.
                for (var i = 0; i < productOfSums.Count - 1; i++)
                {
                    // Start c at i + 1 because the order of brackets does not matter, so
                    // simply search through the remaining brackets.
                    for (var c = i + 1; c < productOfSums.Count; c++)
                    {
                        b1 = productOfSums[i];
                        b2 = productOfSums[c];
                        // Ensuring that the brackets are not the same. Must also consider if
                        // the terms are the same but in the opposite order.
                        if (b1.term1 == b2.term1 != (b1.term2 == b2.term2) != (b1.term1 ==
b2.term2) != (b1.term2 == b2.term1))
                        {
                            // A make the merge as it is a valid one and remove the brackets
                            // that result in the merge. This is to stop the same merges
                            // occurring again and again.
                            mergedTerm = MergeBrackets(b1, b2);
                            productOfSums.Add(mergedTerm);
                            productOfSums.Remove(b1);
                            productOfSums.Remove(b2);
                            merged = true;
                            i = c + 1;
                            c = i + 1;
                        }
                    }
                }
            }
            // Convert the merged product of sums into a string, this allows each bracket
            // to have more than two products within it.
            List<List<string>> stringProducts = ConvertBracketsToString(productOfSums);
            // Recursively apply the distributive law which does the rest of the work.
            List<List<string>> sumOfProducts = RecursiveDistributiveLaw(stringProducts);
            // The first term within the 2D list is the only bracket that remains and hence
```

```
            // can no longer be merged. It also contains all of the solutions to the
minimisation.
            return sumOfProducts[0].ToArray();
        }

        /// <summary>
        /// Carries out the first merge when creating the sum of products. This converts
        /// the product of sums found previously into the correct form for the
        /// recursive application of the distributive law to find the sum of products.
        /// </summary>
        /// <param name="b1">One set of terms being merged</param>
        /// <param name="b2">The other set of terms being merged</param>
        /// <returns></returns>
        private static Bracket MergeBrackets(Bracket b1, Bracket b2)
        {
            // Create new bracket which stores the merged terms.
            var b = new Bracket();
            // When using the distributive law, the term that is the same always remains
            // the first term of the result. The second is the AND of the remaining terms.
            if (b1.term1 == b2.term1)
            {
                b.term1 = b1.term1;
                b.term2 = b1.term2 + b2.term2;
                return b;
            }
            // The second term of the first bracket must match b2.term1 as they are in
            // alphabetical order.
            else
            {
                b.term1 = b1.term2;
                b.term2 = b1.term1 + b2.term2;
                return b;
            }
        }

        /// <summary>
        /// Converts the bracket representation into a string representation. This is because
        /// at this stage of the working, it is possible to have a bracket with more than two
        /// terms such as (AB + CD + EF). The first dimension is the bracket and the second
        /// dimension stores the terms themselves.
        /// </summary>
        /// <param name="brackets"></param>
        /// <returns></returns>
        private static List<List<string>> ConvertBracketsToString(List<Bracket> brackets)
        {
            var result = new List<List<string>>();
            List<string> tmp;
            foreach (Bracket b in brackets)
            {
                // For now, each bracket(list) will only store the two terms in the
                // struct as the distributive has not been applied yet.
                tmp = new List<string>
                {
                    b.term1,
                    b.term2
                };
                result.Add(tmp);
            }
            return result;
        }

        /// <summary>
```

```
        /// Carries out the main distribution to convert the product of sums into the sum
        /// of products. This recursively applies the distributive law until only one
        /// bracket remains which means the law can no longer be applied.
        /// </summary>
        /// <param name="brackets">The string representation of the product of sums.</param>
        /// <returns>A singular string representing the sum of products. </returns>
        private static List<List<string>> RecursiveDistributiveLaw(List<List<string>> brackets)
        {
            var lls = new List<List<string>>();
            // The distributive law can be applied as long as there are at least two brackets
            // remaining in the expression.
            if (brackets.Count > 1)
            {
                // Applies the distributive law on two brackets that contain n and m terms
                // each and adds the result to the list of remaining brackets.
                lls.Add(SingleDistributiveLaw(brackets[0], brackets[1]));
                // The brackets used within the distributive law do not persist.
                // This means that they should be removed.
                brackets.RemoveAt(0);
                brackets.RemoveAt(0);
                lls.AddRange(brackets);
                // More than two brackets remain and so repeat the process.
                return RecursiveDistributiveLaw(lls);
            }
            else
            {
                // Distributive law can no longer be applied and so return the complete
                // sum of products.
                return brackets;
            }
        }

        /// <summary>
        /// Applies the distributive law on two brackets that store m and n terms
        /// respectively. This is because (KN+KLQ+LMN+LMQ)(P+MQ) can be further
        /// distributed.
        /// </summary>
        /// <param name="b1">A bracket storing m number of terms. </param>
        /// <param name="b2">A bracket storing n number of terms that merges into b1.</param>
        /// <returns></returns>
        private static List<string> SingleDistributiveLaw(List<string> b1, List<string> b2)
        {
            var lls = new List<string>();
            // Apply the law to every term within both of the brackets to find the complete
            // expansion of the two brackets.
            for (var i = 0; i < b1.Count; i++)
            {
                for (var j = 0; j < b2.Count; j++)
                {
                    // Applying the distributive law to two products. e.g. KLQ and P
                    lls.Add(ApplyDistributiveLaw(b1[i], b2[j]));
                }
            }
            return lls;
        }

        /// <summary>
        /// Apply the distributive on two individual terms in the supplied brackets.
        /// This simply filters out the duplicate inputs as all unique inputs must remain
        /// when the law is applied.
        /// </summary>
        /// <param name="a">A product with a number of terms. </param>
```

```
        /// <param name="b">A product that is being distributed with A</param>
        /// <returns></returns>
        private static string ApplyDistributiveLaw(string a, string b)
        {
            string tempResult = a + b;
            string finalResult = "";
            foreach (char c in tempResult)
            {
                // Simply find the unique inputs to gain the result of the law.
                if (!finalResult.Contains(c))
                {
                   finalResult += c;
                }
            }
            return finalResult;
        }

        /// <summary>
        /// Finds the product within the sum of products from the distributive law that has
        /// the fewest terms within it. This allows the most minimal expression to be the
        /// result.
        /// </summary>
        /// <returns>The shortest product within the array representing the sum of
        /// products. </returns>
        private static string GetMinProduct(string[] sumOfProducts)
        {
            string minProduct = sumOfProducts[0];
            foreach (string p in sumOfProducts)
            {
                if (p.Length < minProduct.Length)
                {
                    minProduct = p;
                }
            }
            return minProduct;
        }

        /// <summary>
        /// Iterates through the smallest product from the sum of products. Each term within
        /// the product is then replaced with its respective prime implicant which can then
        /// be converted into the completely minimised expression.
        /// </summary>
        /// <param name="termToImplicantMap">The mapping between prime implicants and letters.
        /// This simplifies boolean algebra with the prime implicants.</param>
        /// <param name="minProduct">The smallest product found in the sum of products.
        /// This product will produce the most minimal result. </param>
        /// <returns>The fully minimised expression from Petrick's method. </returns>
        private string GetFinalExpression(Dictionary<char, string> termToImplicantMap, string
minProduct)
        {
            string subExpression;
            string implicant;
            string result = "";
            for (var i = 0; i < minProduct.Length; i++)
            {
                // Convert the letter into an implicant which then produces the minimal
                // expression. ie. K => -011 => !B.C.D
                implicant = termToImplicantMap[minProduct[i]];
                subExpression = ConvertImplicantToExpression(implicant);
                result += subExpression;
                // If we are not converting the last term then the implicants must be
                // separated by OR gates.
```

```
                if (i < minProduct.Length - 1)
                {
                    result += " + ";
                }
            }
            return result;
        }
        #endregion
    }
}
```

Xaml

This section covers the XAML and code-behind file for the user interface of the program.

BooleanExpressionInputDialog.xaml

This is the dialog that is displayed to the user. The window also contains a small table to remind the user of which character represents which logic gate. It is also assumed that the user knows that brackets can also be entered. The window and its XAML is shown below. It should be noted that the expression displayed within the window is an example expression to demonstrate the functionality and appearance of the dialog:

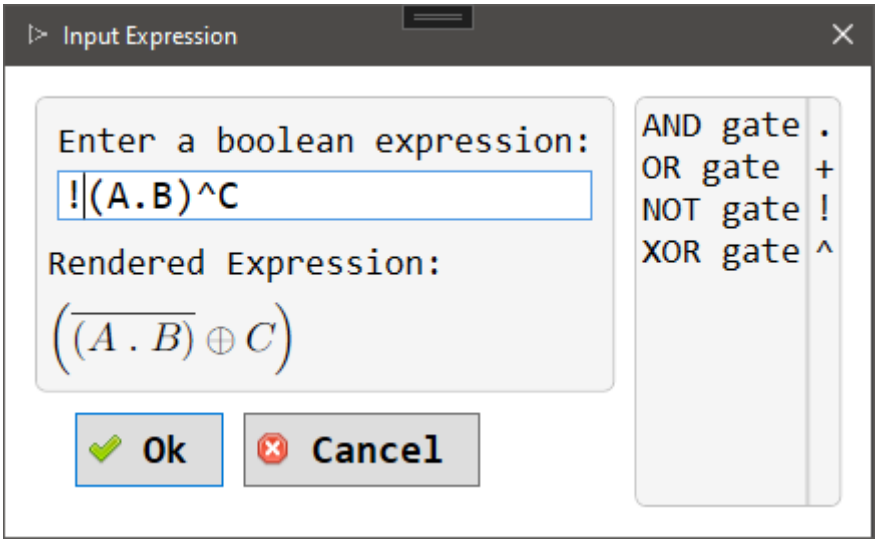


Figure 15: The boolean expression input dialog with an example expression being entered.

The XAML is given below:

```
<Window x:Class="_2BNOR_2B.BooleanExpressionInputDialog"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:controls="clr-namespace:WpfMath.Controls;assembly=WpfMath"
        xmlns:local="clr-namespace:_2BNOR_2B" xmlns:local1="clr-namespace:_2BNOR_2B.Code"
        mc:Ignorable="d"
        Title="Input Expression" SizeToContent="WidthAndHeight"
WindowStartupLocation="CenterScreen"
        Topmost="True" ContentRendered="Window_ContentRendered" ResizeMode="NoResize">
    <!-- Using the boolean renderer to show the user how the expression would look if it was
    written by hand. -->
    <Window.Resources>
        <local1:BooleanConverter x:Key="booleanConverter"/>
    </Window.Resources>

    <Grid Margin="15">
        <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="10"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <!--The table of logic gates. This is to remind the user of which gate means which
symbol. -->
        <Border Grid.Column="2" BorderBrush="Silver" Background="WhiteSmoke"
BorderThickness="1"  CornerRadius="5">
            <StackPanel Orientation="Horizontal">
                <StackPanel Margin="2.5">
                    <TextBlock FontFamily="Consolas" FontSize="18">AND gate</TextBlock>
                    <TextBlock FontFamily="Consolas" FontSize="18">OR gate</TextBlock>
                    <TextBlock FontFamily="Consolas" FontSize="18">NOT gate</TextBlock>
                    <TextBlock FontFamily="Consolas" FontSize="18">XOR gate</TextBlock>
                </StackPanel>
                <Rectangle Width="2" Fill="LightGray"></Rectangle>
                <StackPanel Margin="2.5">
                    <TextBlock FontFamily="Consolas" FontSize="18"> .</TextBlock>
                    <TextBlock FontFamily="Consolas" FontSize="18"> +</TextBlock>
                    <TextBlock FontFamily="Consolas" FontSize="18"> !</TextBlock>
                    <TextBlock FontFamily="Consolas" FontSize="18"> ^</TextBlock>
                </StackPanel>
            </StackPanel>
        </Border>

        <!-- The user inputs their expression into the following textbox and it is rendered.
        A curved border has been used to maintain the same style throughout the program. -->
        <StackPanel Grid.Row="0">
            <Border BorderBrush="Silver" Background="WhiteSmoke" BorderThickness="1"
CornerRadius="5">
                <StackPanel Margin="5">
                    <Label FontFamily="Consolas" FontSize="18">
                        <Label.Content>Enter a boolean expression:</Label.Content>
                    </Label>
                    <TextBox Margin="5,0" FontFamily="Consolas" Name="inputBox" FontSize="20"/>
                    <Rectangle Height="10"/>
                    <TextBlock FontFamily="Consolas" FontSize="18" Text="Rendered Expression:
"/>
                    <Rectangle Height="10"/>
                    <controls:FormulaControl x:Name="renderedExpression" Formula="{Binding
Path=Text, ElementName=inputBox, Converter={StaticResource booleanConverter}}"/>
                </StackPanel>
            </Border>

            <!-- Okay and Cancel buttons so the user can stop if they wish and also confirm
when
            they have entered their desired expression. -->
            <StackPanel Margin="10" Orientation="Horizontal">
                <Button IsDefault="True" Margin ="10,0" HorizontalAlignment="Center"
Click="BtnDialogOk_Click">
                    <Button.Content>
                        <StackPanel Orientation="Horizontal" Margin="5">
                            <Image Source="/Resources/tick.png"/>
                            <TextBlock Text=" Ok " FontFamily="Consolas" FontSize="20"
FontWeight="DemiBold"/>
```

```
                        </StackPanel>
                    </Button.Content>
                </Button>

                <Button IsCancel="True" Click="BtnDialogCancel_Click">
                    <Button.Content>
                        <StackPanel Orientation="Horizontal" Margin="5">
                            <Image Source="/Resources/cancel.png"/>
                            <TextBlock Text=" Cancel " FontFamily="Consolas" FontSize="20"
FontWeight="DemiBold"/>
                        </StackPanel>
                    </Button.Content>
                </Button>
            </StackPanel>
        </StackPanel>
    </Grid>
</Window>
```

BooleanExpressionInputDialog.xaml.cs

This is the code-behind file for the Boolean expression input dialog. This is responsible for receiving and the user input and getting it to the diagram class for processing. The dialog also shows the rendered expression to the user. This is so the user is aware of how the program is interpreting their expression, but it can also be helpful for someone trying to copy an expression from another source into the program.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace _2BNOR_2B
{
    /// <summary>
    /// Interaction logic for BooleanExpressionInputDialog.xaml
    /// </summary>
    public partial class BooleanExpressionInputDialog : Window
    {
        public BooleanExpressionInputDialog()
        {
            InitializeComponent();
        }

        /// <summary>
        /// The user has written an expression and they have clicked ok. This means
        /// that the expression is ready to be validated and processed if valid.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void BtnDialogOk_Click(object sender, RoutedEventArgs e)
        {
            this.DialogResult = true;
        }
```

```
        /// <summary>
        /// The user does not want to enter an expression. Respond with no entered expression.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void BtnDialogCancel_Click(object sender, RoutedEventArgs e)
        {
            this.DialogResult = false;
        }

        /// <summary>
        /// When this window is shown to the screen select the text box so the user can
        /// immediately start typing and also draw attention to the window, by focusing
        /// on it.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Window_ContentRendered(object sender, EventArgs e)
        {
            inputBox.SelectAll();
            inputBox.Focus();
        }

        /// <summary>
        /// Gets the result of the dialog. This is the users' inputted Boolean expression.
        /// </summary>
        public string Result
        {
            get { return inputBox.Text; }
        }
    }
}
```

ExpressionDisplayWindow.xaml

This is a simple window that shows the rendered Boolean expression that was used to create the diagram. The window is shown below. It should also be noted that the expression displayed within the window is also an example expression used to highlight the functionality of the window:

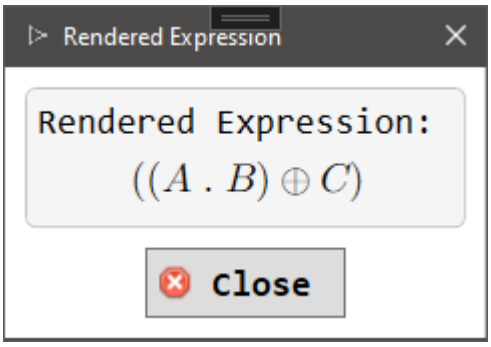


Figure 16: The expression display window for Boolean expressions.

The XAML for this window is shown below:

```
<Window x:Class="_2BNOR_2B.renderedExpressionDisplay"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_2BNOR_2B" xmlns:controls="clr-
namespace:WpfMath.Controls;assembly=WpfMath" xmlns:local1="clr-namespace:_2BNOR_2B.Code"
        mc:Ignorable="d"
```

```
        Title="Rendered Expression" Height="Auto" Width="Auto" SizeToContent="WidthAndHeight"
WindowStartupLocation="CenterScreen"
        Topmost="True" ResizeMode="NoResize">
    <!-- Using the Boolean converter. -->
    <Window.Resources>
        <local1:BooleanConverter x:Key="BooleanConverter"/>
    </Window.Resources>

    <!-- The rendered expression is displayed here. It is in a curved border to maintain the
style
    of the program. -->
    <StackPanel Margin="10">
        <Border CornerRadius="5" Background="WhiteSmoke" BorderBrush="Silver"
                BorderThickness="1">
            <StackPanel Margin="5">
                <TextBlock x:Name="HeadingText" FontFamily="Consolas" FontSize="18"/>
                <Rectangle Height="10"/>
                <controls:FormulaControl x:Name="renderedExpressionBox"
HorizontalAlignment="Center"/>
                <Rectangle Height="5"/>
            </StackPanel>
        </Border>

        <Rectangle Height="10"/>

        <!-- Cancel button so that the user can quickly close down the window. -->
        <Button Click="Button_Click" Width="100" HorizontalAlignment="Center">
            <Button.Content>
                <StackPanel Orientation="Horizontal" Margin="5">
                    <Image Source="/Resources/cancel.png"/>
                    <TextBlock Text=" Close " FontFamily="Consolas" FontSize="18"
FontWeight="DemiBold"/>
                </StackPanel>
            </Button.Content>
        </Button>
    </StackPanel>
</Window>
```

ExpressionDisplayWindow.xaml.cs

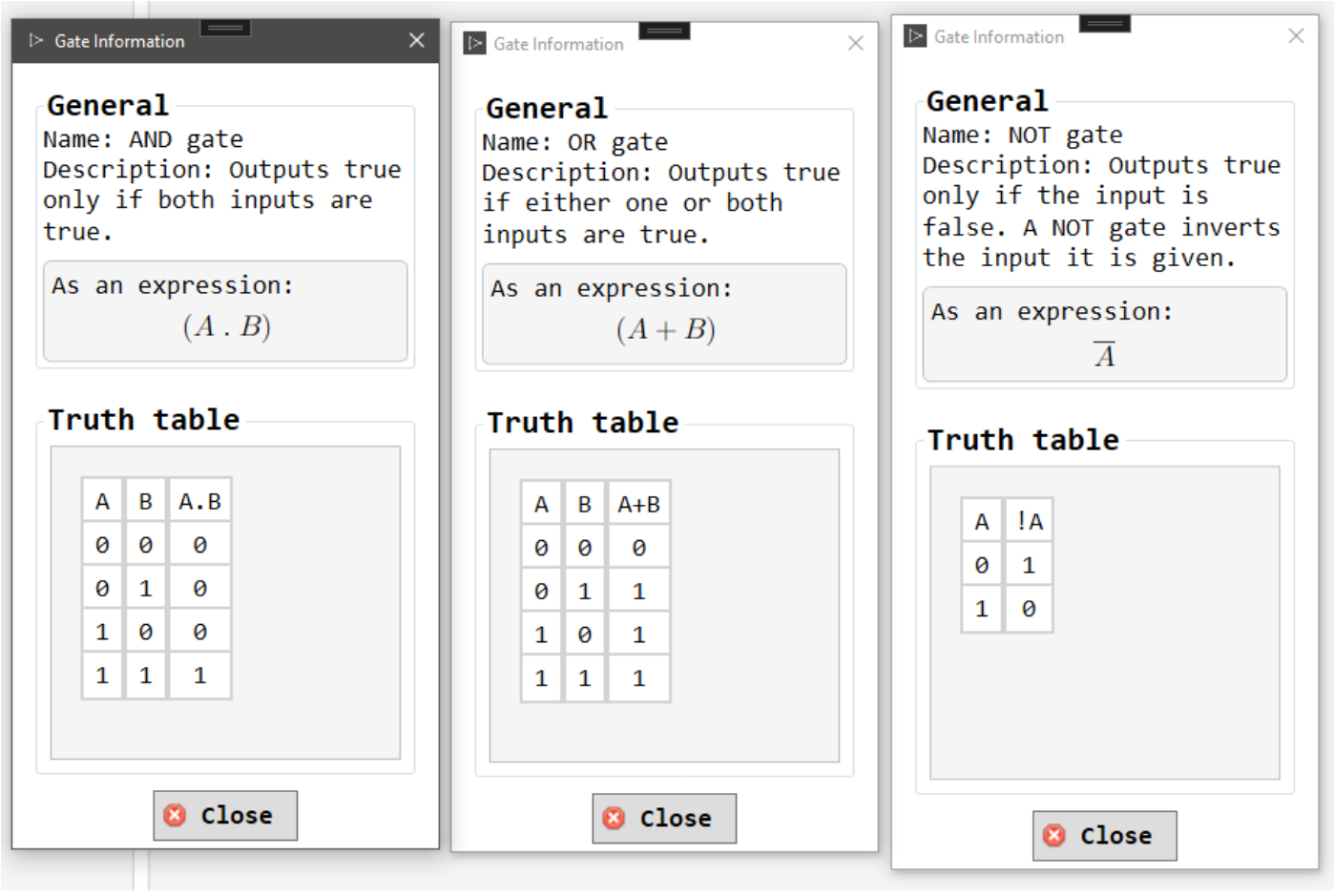
This is the code behind for displaying the Boolean expressions. This is done by instantiating the Boolean converter which is used to convert Boolean expressions in the program's own notation into LATEX so that they can be rendered by the renderer.

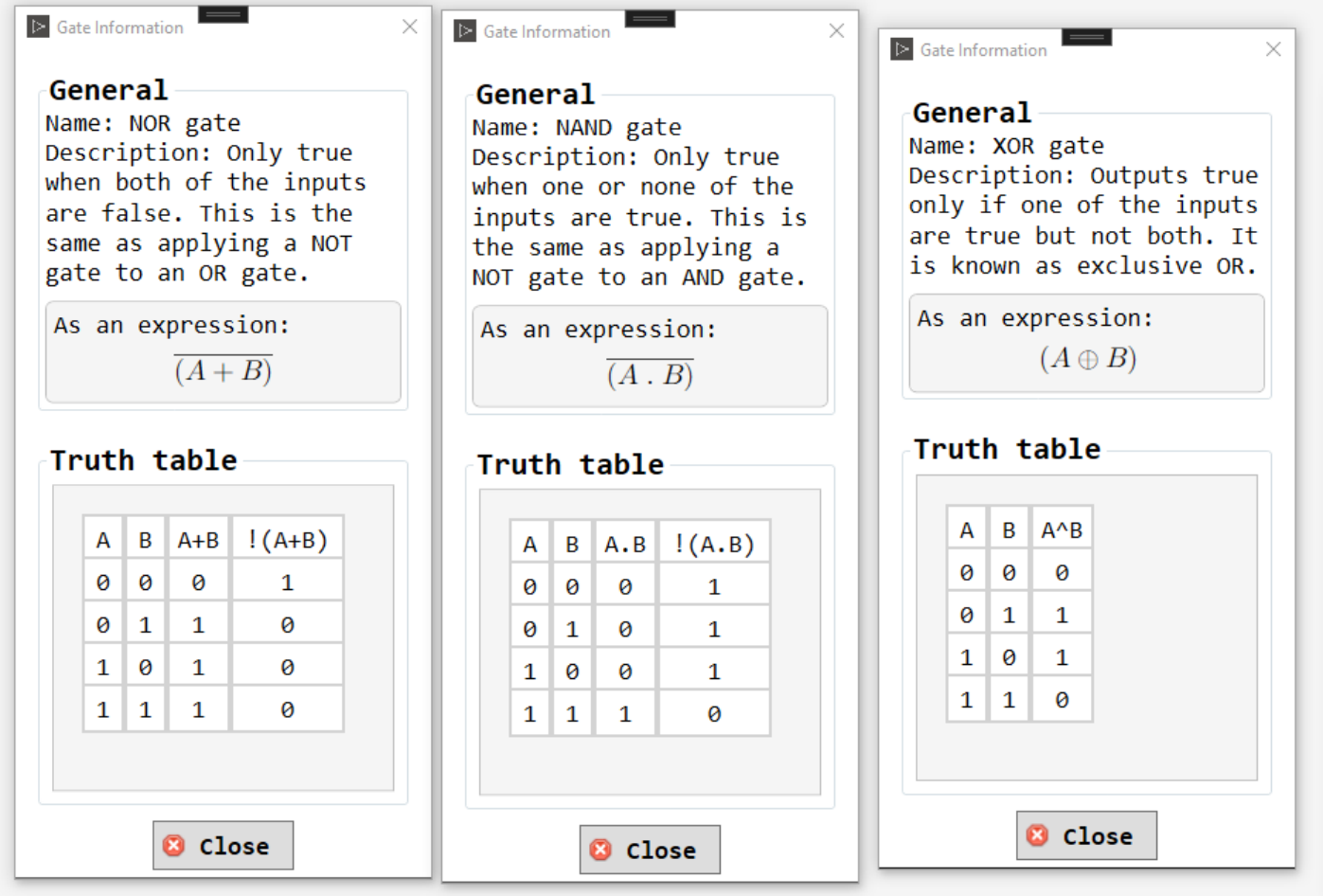
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using _2BNOR_2B.Code;

namespace _2BNOR_2B
{
    /// <summary>
```

```
    /// Interaction logic for renderedExpression.xaml
    /// </summary>
    public partial class renderedExpressionDisplay : Window
    {
        public renderedExpressionDisplay(string expression, string HeaderText)
        {
            InitializeComponent();
            // Rendering the Boolean expression with latex.
            var bc = new BooleanConverter();
            string converted = bc.ConvertString(expression);
            renderedExpressionBox.Formula = converted;
            HeadingText.Text = HeaderText;
        }

        /// <summary>
        /// A simple close button, so the user has a clear way of closing down the window.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Close();
        }
    }
}
```

GateInformationWindow.xaml

This window creates pop-up menu for the user that contains useful and general information about the six logic gates within the A-level specification. All of the pop-up menus for the gates can be seen below:





The XAML to create these windows is below:

```
<Window x:Class="_2BNOR_2B.GateInformation"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_2BNOR_2B" xmlns:controls="clr-
namespace:WpfMath.Controls;assembly=WpfMath" xmlns:local1="clr-namespace:_2BNOR_2B.Code"
        mc:Ignorable="d"
        Title="Gate Information" Height="Auto" Width="310" SizeToContent="Height"
WindowStartupLocation="CenterScreen"
        Topmost="True" ResizeMode="NoResize">
    <!-- Using the Boolean converter to render expressions for the user. -->
    <Window.Resources>
        <local1:BooleanConverter x:Key="booleanConverter"/>
    </Window.Resources>

    <StackPanel Margin="5">
        <!-- Group box for the general information about the gate. This information is filled
        in the class constructor. -->
        <GroupBox Margin="10">
            <GroupBox.Header>
                <TextBlock FontWeight="Bold" FontFamily="Consolas"
FontSize="22">General</TextBlock>
            </GroupBox.Header>

            <StackPanel>
                <TextBlock x:Name="gateName" FontSize="18" FontFamily="Consolas"/>
                <TextBlock x:Name="gateDescription" FontSize="18" FontFamily="Consolas"
                           TextWrapping="WrapWithOverflow"/>
```

```
                <Rectangle Height="10"/>
                <Border CornerRadius="5" Background="WhiteSmoke" BorderBrush="Silver"
        BorderThickness="1">
                    <StackPanel Margin="5">
                        <TextBlock Text="As an expression: " FontFamily="Consolas"
FontSize="18"/>
                        <Rectangle Height="10"/>
                        <controls:FormulaControl x:Name="renderedExpressionBox"
HorizontalAlignment="Center"/>
                        <Rectangle Height="5"/>
                    </StackPanel>
                </Border>
            </StackPanel>
        </GroupBox>
        <!-- Location of the truth table, which is rendered by the diagram class. -->
        <GroupBox Margin="10">
            <GroupBox.Header>
                <TextBlock FontWeight="Bold" FontFamily="Consolas" FontSize="22">Truth
table</TextBlock>
            </GroupBox.Header>

            <Border BorderThickness="1" BorderBrush="Silver" Margin="5">
                <Canvas x:Name="TruthTableCanvas" Background="WhiteSmoke" Height="185"/>
            </Border>

        </GroupBox>
        <!-- Quick close button so the user knows how to leave the window. -->
        <Button Click="Button_Click" Width="100" HorizontalAlignment="Center">
            <Button.Content>
                <StackPanel Orientation="Horizontal" Margin="5">
                    <Image Source="\Resources\cancel.png"/>
                    <TextBlock Text=" Close " FontFamily="Consolas" FontSize="18"
FontWeight="DemiBold"/>
                </StackPanel>
            </Button.Content>
        </Button>
    </StackPanel>
</Window>
```

GateInformationWindow.xaml.cs

This class acts as the code behind for the pop-menus on the main window. It simply retrieves the relevant gate information and displays it to the user. It should be noted that the diagram class is supplied as a parameter so that it can handle the drawing of the truth tables. This removes the need to have redundant methods in this class.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using _2BNOR_2B.Code;

namespace _2BNOR_2B
{
```

```
    /// <summary>
    /// Interaction logic for GateInformation.xaml
    /// </summary>
    public partial class GateInformation : Window
    {
        // Names of all of the different logic gates that a popup window may have to display.
        private readonly string[] names = { "AND gate", "OR gate", "NOT gate", "XOR gate",
"NAND gate", "NOR gate" };
        // The respective descriptions of the different logic gates given by the array above.
        private readonly string[] descriptions = {"Outputs true only if both inputs are true.
",
                                                  "Outputs true if either one or both inputs are true.",
                                                  "Outputs true only if the input is false. A NOT gate
inverts the input it is given.",
                                                  "Outputs true only if one of the inputs are true but
not both. It is known as exclusive OR. ",
                                                  "Only true when one or none of the inputs are true.
This is the same as applying a NOT gate to an AND gate.",
                                                  "Only true when both of the inputs are false. This is
the same as applying a NOT gate to an OR gate. "};
        // Basic expressions  that demonstrate a usage of each logic gate in its simplest form.
        // These are also used to show the truth table of particular gate.
        private readonly string[] exampleExpressions = { "A.B", "A+B", "!A", "A^B", "!(A.B)",
"!(A+B)" };

        // Supplying the diagram as a parameter as it handles all of the truth table drawing.
        public GateInformation(Diagram d, string gate)
        {
            InitializeComponent();
            gateName.Text = $"Name: {gate}";
            int i = Array.IndexOf(names, gate);
            // Getting the corresponding description for the gate information being shown
            // to the user.
            gateDescription.Text = $"Description: {descriptions[i]}";
            var bc = new BooleanConverter();
            // Showing the user a rendered version of the expression as they need to be aware
            // of the symbol for each logic gate and how they are written in expressions.
            string converted = bc.ConvertString(exampleExpressions[i]);
            renderedExpressionBox.Formula = converted;
            // Drawing the gates respective truth table.
            d.DrawTruthTable(TruthTableCanvas, exampleExpressions[i]);
        }

        /// <summary>
        /// A simple button that closes down the pop-up window.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Button_Click(object sender, RoutedEventArgs e)
        {
            Close();
        }
    }
}
```

LogicGate.xaml

This is a custom user control that represents the logic gates on the diagram canvas in the main window.

```
<UserControl x:Class="_2BNOR_2B.LogicGate"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:_2BNOR_2B"
             mc:Ignorable="d"
             d:DesignHeight="450" d:DesignWidth="450">
    <!-- Container for both the label and the image being displayed on the canvas.
    Uses a transparent background to not interfere with the main window canvas. -->
    <Canvas Background="Transparent">
        <!-- Default to using an input pin for the image. -->
        <Image Width="450" Height="450" x:Name="elementImage" Source="images\input_pin.png"
               Panel.ZIndex="0"/>
        <!-- Added a label for the input pins. This has a default width of zero as there is
        no charcter within the label. A ZIndex of one is used so that the label can be
        seen over the image of the input pin. -->
        <Label Canvas.Top="3" Canvas.Left="7" HorizontalAlignment="Center"
               VerticalAlignment="Center" FontFamily="Consolas" FontSize="24" Panel.ZIndex="1"
               x:Name="elementLabel"/>
    </Canvas>
</UserControl>
```

LogicGate.xaml.cs

This code-behind that establishes the link between the nodes within the binary tree and the visual items that are being displayed on the diagram canvas in the main window of the program. This class also contains method for getting the input and output points for the components. This is very important as it always the wires to always be drawn to the correct position as it is relative to the position of the logic gate.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using _2BNOR_2B.Code;

namespace _2BNOR_2B
{
    /// <summary>
    /// Interaction logic for logicGate.xaml
    /// </summary>
    public partial class LogicGate : UserControl
    {
        // The element (non-visual node in the tree) that links the logic gate to a node.
        // This is the link between the visual interactions and the diagram class itself.
        private readonly Element gate;
        private readonly double labelWidth;
        private int connectedWires = 0;

        public LogicGate(Element gate)
        {
            InitializeComponent();
            this.gate = gate;
            SetImage();
            this.PreviewMouseDown += LogicGate_PreviewMouseDown;
```

```
            this.MouseMove += LogicGate_MouseMove;
        }

        /// <summary>
        /// Changes the cursor when hovering over an input pin, this is to show that they
        /// are clickable and this creates a subtle hint to the user about diagram
interactivity.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void LogicGate_MouseMove(object sender, MouseEventArgs e)
        {
            // Whenever the cursor is over a logic gate and that gate is an input pin
            // then change the cursor to show that the input is clickable.
            if (gate.GetElementName() == "input_pin")
            {
                Mouse.SetCursor(Cursors.Hand);
            }
        }

        /// <summary>
        /// Used for when a logic gate will have multiple wires connected to it.
        /// This is used when shifting the repeated input. This makes the diagrams
        /// produced clearer as the wires do not overlap over each other.
        /// </summary>
        public void AddWire()
        {
            connectedWires++;
        }

        public int GetConnectedWires()
        {
            return connectedWires;
        }

        /// <summary>
        /// Changes the state of an input pin when it is clicked by the user. This is so
        /// the diagram is updated when it is clicked.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void LogicGate_PreviewMouseDown(object sender, MouseButtonEventArgs e)
        {
            // Only the state of an input can be changed by the user.
            if (gate.GetElementName() == "input_pin")
            {
                if (gate.GetState() == 1)
                {
                    gate.SetState(0);
                }
                else
                {
                    gate.SetState(1);
                }
            }

        }

        /// <summary>
        /// Used for connecting wires to gates.
        /// </summary>
        /// <returns>The point on the canvas where the left input is. This is for any
```

```
        /// logic gate that is the left child of another. </returns>
        public Point GetInputPoint1()
        {
            double xPosition = Canvas.GetLeft(this);
            // Shift down as the input is not in the top left corner of the image.
            double yPosition = Canvas.GetTop(this) + 10;
            var tmp = new Point(xPosition, yPosition);
            return tmp;
        }

        /// <summary>
        /// Used for connecting wires to gates.
        /// </summary>
        /// <returns>The point on the canvas where the left input is. This for any
        /// logic gate that is the right child of another. </returns>
        public Point GetInputPoint2()
        {
            // The top left position of the logic gate.
            double xPosition = Canvas.GetLeft(this);
            double yPosition = Canvas.GetTop(this);
            // If the gate is a NOT gate then a different shift needs to be applied
            // as the input point is in the centre of the gate.
            if (gate.leftChild == null && gate.rightChild != null)
            {
                var tmp = new Point(xPosition, yPosition + 20);
                return tmp;
            }
            else
            {
                var tmp = new Point(xPosition, yPosition + 30);
                return tmp;
            }
        }

        /// <summary>
        /// Used for when drawing a wire from the rootnode of the tree to the output
        /// node of the diagram.
        /// </summary>
        /// <returns>The point on the canvas where the wire from the output of the rootnode
        /// connects the output pin on the canvas. </returns>
        public Point GetInputForOutput()
        {
            // The top left position of the output pin.
            double xPosition = Canvas.GetLeft(this);
            double yPosition = Canvas.GetTop(this);
            // The output point is in the centre of the logic gate.
            yPosition += elementImage.Height / 2 - 1;
            var tmp = new Point(xPosition, yPosition);
            return tmp;
        }

        /// <summary>
        /// Used to connect a gate to a wire.
        /// </summary>
        /// <returns>The point on the canvas where the wire comes out of a logic gate to
        /// connect two gates together.
        /// </returns>
        public Point GetOutputPoint()
        {
            // The intial position of the output point.
            double xPosition = Canvas.GetLeft(this) + elementImage.Width + labelWidth;
            double yPosition = Canvas.GetTop(this) + elementImage.Height / 2;
```

```
            // If the gate is an input pin, then apply a shift to x-axis to deal with the
            // differently sized image.
            if (gate.GetElementName() == "input_pin")
            {
                xPosition -= 43;
                var tmp = new Point(xPosition, yPosition);
                return tmp;
            }
            else
            {
                // Apply a different shift for the other gates.
                xPosition -= 5;
                var tmp = new Point(xPosition, yPosition);
                return tmp;
            }
        }

        public Element GetGate()
        {
            return gate;
        }

        /// <summary>
        /// Sets the image for the logic gate that is displayed on the canvas. This
        /// makes the logic gate visisble.
        /// </summary>
        /// <exception cref="Exception">A relevant image could not be found for the
        /// gate being searched for. </exception>
        private void SetImage()
        {
            BitmapImage bitmap;
            Uri path;
            string imageName = gate.GetElementName();
            try
            {
                // Finding the relevant image based off of the name of the element being
                // created.
                path = new Uri($"pack://application:,,,/Resources/{imageName}.png");
                bitmap = new BitmapImage(path);
                // Applying a transformation to the image so that it fits nicely onto the
                // canvas.
                elementImage.Stretch = Stretch.Uniform;
                elementImage.Height = bitmap.Height / 5;
                elementImage.Width = bitmap.Width / 5;
                elementImage.Source = bitmap;
                // Setting the label of the gate. This is simply empty for a logic gate.
                elementLabel.Content = gate.GetLabel();
            }
            catch (Exception e)
            {
                throw new Exception($"Could not load the image for: {imageName}", e);
            }
        }
    }
}
```

MainWindow.xaml

This is the main window that the user interacts with, when using the program. A picture of the user interface is shown below. In the picture the truth table and the diagram can be seen on the two main canvases of the program. A pop-up menu is also shown which is opened when one of the respective buttons in the side panel is left-clicked. In the centre of the screen, the input dialog is shown which is where the user inputs their Boolean expression and

where it is also validated. To ensure that the program remains as clear as possible, simple visual ques, such as the shaded background of the buttons and labels are used.

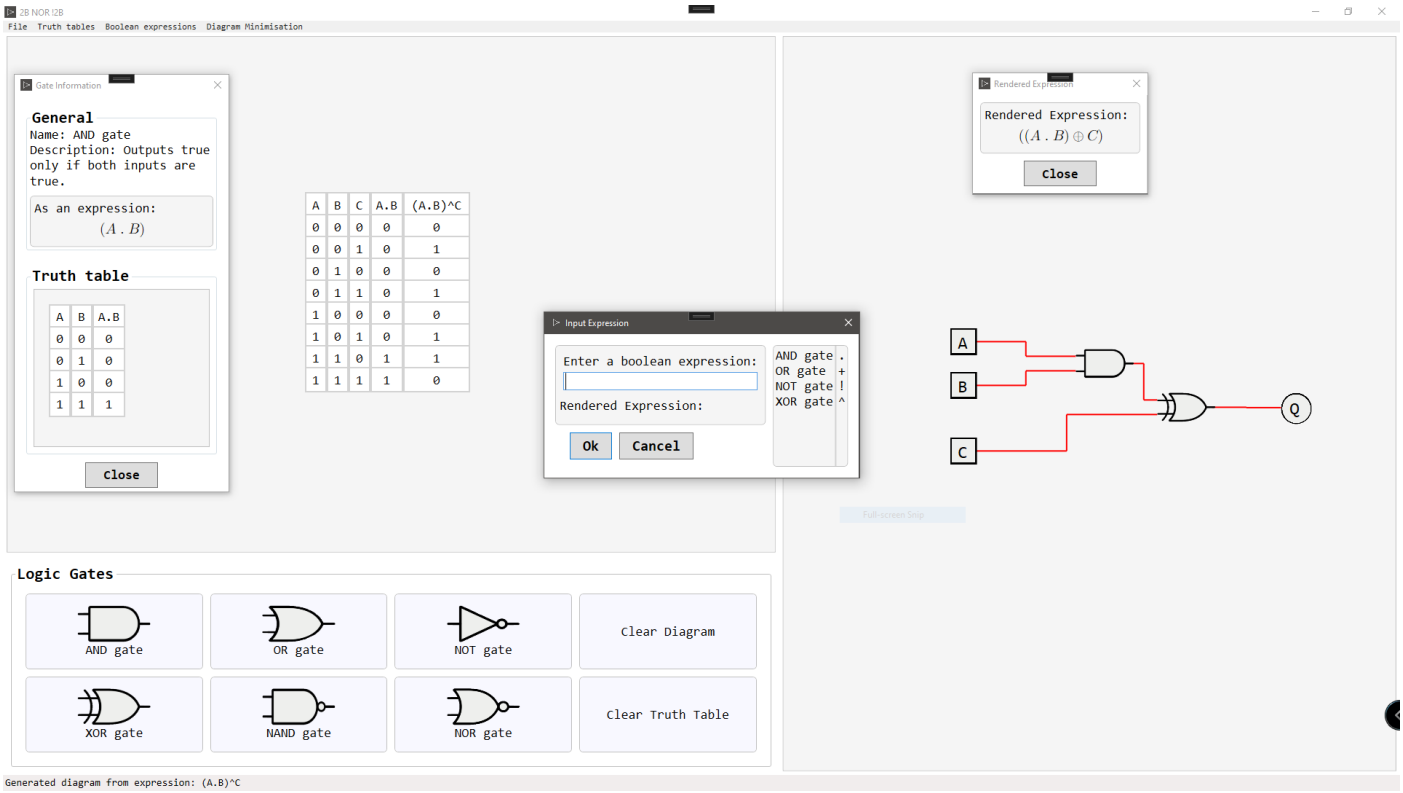


Figure 17: The window of the program. Some pop-ups and dialog of the program are also shown.

The XAML for this window is below:

```
<Window x:Class="_2BNOR_2B.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_2BNOR_2B"
        mc:Ignorable="d"
        Title="2B NOR !2B" MinHeight="600" MinWidth="1000" WindowState="Maximized">
    <Window.Resources>
        <!-- Style for the buttons on the main window. Only used here so there is no
        need to put it in the application wide resources. -->
        <Style x:Key="PopUpButtonStyle" TargetType="Button">
            <Setter Property="Background" Value="GhostWhite"/>
            <Setter Property="BorderBrush" Value="Silver"/>
            <Setter Property="Margin" Value="5"/>
            <Style.Resources>
                <Style TargetType="Border">
                    <Setter Property="CornerRadius" Value="5"/>
                </Style>
            </Style.Resources>
        </Style>
        <!-- Style for the textblock in the buttons for the pop-ups. This is used
        to ensure the styling of the program is kept consistent throughout. -->
        <Style x:Key="TextBlockStyle" TargetType="TextBlock">
            <Setter Property="FontFamily" Value="Consolas"/>
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="TextWrapping" Value="Wrap"/>
            <Setter Property="TextAlignment" Value="Center"/>
        </Style>
```

```
    </Window.Resources>

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="3.25*" MinHeight="300"/>
            <RowDefinition Height="*" MinHeight="300"/>
            <RowDefinition Height="Auto" MinHeight="20"/>
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="1.25*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <Menu Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" FontFamily="Consolas">
            <MenuItem Header="_File">
                <MenuItem Header="_Open Diagram" Click="MenuItem_LoadDiagram"/>
                <MenuItem Header="_Save Diagram" Click="MenuItem_SaveDiagram"/>
                <MenuItem Header="_Export" Click="MenuItem_ExportDiagram" />
                <Separator/>
                <MenuItem Header="Exit" Click="MenuItem_CloseApp" />
            </MenuItem>
            <MenuItem Header="_Truth tables">
                <MenuItem Header="Generate From Diagram"
Click="MenuItem_GenerateTableFromDiagram"/>
                <MenuItem Header="Generate From Expression"
Click="MenuItem_GenerateTableFromExpression"/>
            </MenuItem>
            <MenuItem Header="_Boolean expressions">
                <MenuItem Header="Create Diagram"
Click="MenuItem_GenerateDiagramFromExpression"/>
                <MenuItem Header="Find Expression"
Click="MenuItem_GenerateExpressionFromDiagram"/>
            </MenuItem>
            <MenuItem Header="_Diagram Minimisation">
                <MenuItem Header="Minmise Diagram" Click="MenuItem_MinimiseDiagram"/>
                <MenuItem Header="Minimise Expression" Click="MenuItem_MinimiseExpression"/>
            </MenuItem>
        </Menu>

        <Border Grid.Column="1" Grid.RowSpan="2" Grid.Row="1" BorderThickness="1"
BorderBrush="Silver"
                Margin="5" Background="WhiteSmoke">
            <!-- Scrollviewer so that diagrams of any size are able to be viewed by the user.
This
            removes the need for any restrictions for the size of the diagrams being drawn. -->
            <ScrollViewer CanContentScroll="True"
                          VerticalScrollBarVisibility="Auto"
                          HorizontalScrollBarVisibility="Auto">
                <!-- Using a width and height of 1, to ensure that the scrollviewer is only
enabled
                when necessary. The click event is for when the user interacts with the
diagram. -->
                <Canvas x:Name="DiagramCanvas"
                        MouseDown="DiagramCanvas_MouseDown" Height="1" Width="1"/>
            </ScrollViewer>
        </Border>

        <Border Grid.Column="0" Grid.Row="1"
                      BorderThickness="1" BorderBrush="Silver"
                      Margin="5" Background="WhiteSmoke">
```

```
            <!-- Scrollviewer so that truth tables of any size can be displayed to the user.
This
            creates as much flexibility within the program as possible. -->
            <ScrollViewer CanContentScroll="True"
                          VerticalScrollBarVisibility="Auto"
                          HorizontalScrollBarVisibility="Auto">
                <!--Setting the width and height of the canvas to 1 so that the scroll viewers
are
                visible when necessary.-->
                <Canvas x:Name="TruthTableCanvas"
                        Height="1" Width="1"/>
            </ScrollViewer>
        </Border>

        <GroupBox Margin="10" Padding="10" Grid.Column="0" Grid.Row="2" BorderBrush="Silver">
            <GroupBox.Header>
                <TextBlock FontWeight="Bold" FontFamily="Consolas" FontSize="22">Logic
Gates</TextBlock>
            </GroupBox.Header>
            <!-- Grid for the array of buttons that open up pop-menus that contain the gate
            information. -->
            <Grid Grid.Column="0" Grid.Row="2">
                <Grid.RowDefinitions>
                    <RowDefinition/>
                    <RowDefinition/>
                </Grid.RowDefinitions>

                <Grid.ColumnDefinitions>
                    <ColumnDefinition/>
                    <ColumnDefinition/>
                    <ColumnDefinition/>
                    <ColumnDefinition/>
                </Grid.ColumnDefinitions>
                <!-- Buttons that open up the pop-up menus which provide useful information to
the user
                about specific logic gates. -->
                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="0" Grid.Row="0"
                        Click="ANDInformation">
                    <StackPanel Margin="10" Width="100">
                        <Image Source="Resources/and_gate.png"/>
                        <TextBlock Style="{StaticResource TextBlockStyle}" Text="AND gate"/>
                    </StackPanel>
                </Button>

                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="1" Grid.Row="0"
                        Click="ORInformation">
                    <StackPanel Margin="10" Width="100">
                        <Image Source="Resources/or_gate.png"/>
                        <TextBlock Style="{StaticResource TextBlockStyle}" Text="OR gate"/>
                    </StackPanel>
                </Button>

                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="2" Grid.Row="0"
                        Click="NOTInformation">
                    <StackPanel Margin="10" Width="100">
                        <Image Source="Resources/not_gate.png"/>
                        <TextBlock Style="{StaticResource TextBlockStyle}" Text="NOT gate"/>
                    </StackPanel>
                </Button>

                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="0" Grid.Row="1"
                        Click="XORInformation">
```

```
                    <StackPanel Margin="10" Width="100">
                        <Image Source="Resources/xor_gate.png"/>
                        <TextBlock Style="{StaticResource TextBlockStyle}" Text="XOR gate"/>
                    </StackPanel>
                </Button>

                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="1" Grid.Row="1"
                        Click="NANDInformation">
                    <StackPanel Margin="10" Width="100">
                        <Image Source="Resources/nand_gate.png"/>
                        <TextBlock Style="{StaticResource TextBlockStyle}" Text="NAND gate"/>
                    </StackPanel>
                </Button>

                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="2" Grid.Row="1"
                        Click="NORInformation">
                    <StackPanel Margin="10" Width="100">
                        <Image Source="Resources/nor_gate.png"/>
                        <TextBlock Style="{StaticResource TextBlockStyle}" Text="NOR gate"/>
                    </StackPanel>
                </Button>

                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="4" Grid.Row="0"
                        Click="Button_Click_Diagram">
                    <TextBlock Style="{StaticResource TextBlockStyle}" Text="Clear Diagram"/>
                </Button>

                <Button Style="{StaticResource PopUpButtonStyle}" Grid.Column="4" Grid.Row="1"
                        Click="Button_Click_TT">
                    <TextBlock Style="{StaticResource TextBlockStyle}" Text="Clear Truth
Table"/>
                </Button>
            </Grid>
        </GroupBox>
        <!-- status bar which indicates the most recent interaction between the computer and
the
        user. -->
        <StatusBar Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="2">
            <StatusBarItem>
                <TextBlock x:Name="statusBar" FontFamily="Consolas" FontSize="14"/>
            </StatusBarItem>
        </StatusBar>
    </Grid>
</Window>
```

MainWindow.xaml.cs

This is the logic behind the user interface of the window. This runs the code for the buttons and is also the linking point between the user interface (user facing) and the diagram class which is responsible for all of the data processing (computer facing).

```
using _2BNOR_2B.Code;
using _2BNOR_2B.Properties;
using Microsoft.Win32;
using System;
using System.Collections.Generic;
using System.Diagnostics.Eventing.Reader;
using System.Diagnostics.Metrics;
using System.IO;
using System.Linq;
using System.Linq.Expressions;
using System.Text;
```

```
using System.Text.RegularExpressions;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Converters;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace _2BNOR_2B
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        // Stores the main diagram of the application. This creates truth tables, diagrams
        // and handles of the processing that is done in the application.
        private readonly Diagram d;
        //  String used to save/load diagrams into and out of the application.
        private string saveString = "";

        public MainWindow()
        {
            InitializeComponent();
            d = new Diagram(DiagramCanvas);
        }

        /// <summary>
        /// Generates a truth table from the diagram that is currently drawn in the diagram
        /// creation window.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_GenerateTableFromDiagram(object sender, RoutedEventArgs e)
        {
            // If the expression stored within the diagram does not exist then the diagram
            // must not exist and so error is flagged.
            if (d.GetExpression() != "")
            {
                d.DrawTruthTable(TruthTableCanvas, d.GetExpression());
                // Updating status bar to reflect changes.
                statusBar.Text = $"Generated truth table for the expression
{d.GetExpression()}.";
            }
            else
            {
                MessageBox.Show("Error generating truth table: Diagram does not exist. ",
"Truth table error", MessageBoxButton.OK, MessageBoxImage.Error);
            }
        }

        /// <summary>
        /// Clears the logic diagram currently drawn within the diagram creation window.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Button_Click_Diagram(object sender, RoutedEventArgs e)
```

```
        {
            // If the canvas has no elements (children) in it, then the diagram must not
            // exist.
            if (DiagramCanvas.Children.Count == 0)
            {
                statusBar.Text = "Please draw a diagram first. ";
            }
            else
            {
                // Clearing the diagram, canvas and notifying the user through the status
                // bar.
                d.ClearDiagram();
                DiagramCanvas.Children.Clear();
                statusBar.Text = "Cleared the current diagram. ";
            }

        }

        /// <summary>
        /// Generates and displays a truth table from a user-entered Boolean expression.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_GenerateTableFromExpression(object sender, RoutedEventArgs e)
        {
            var expressionInputDialog = new BooleanExpressionInputDialog();
            string expression;
            // Asking the user to enter a Boolean expression and storing the result.
            if (expressionInputDialog.ShowDialog() == true)
            {
                expression = expressionInputDialog.Result;
                // If the entered expression is valid then the truth table can be drawn and
                // reflecting that change within the status bar.
                if (d.IsExpressionValid(expression))
                {
                    d.DrawTruthTable(TruthTableCanvas, expression);
                    statusBar.Text = "Generated Truth table from expression: " + expression;
                }
                else
                {
                    MessageBox.Show("The expression is invalid", "Validation Error",
MessageBoxButton.OK, MessageBoxImage.Warning);
                }

            }
        }

        /// <summary>
        /// Displaying a rendered Boolean expression from the user-drawn diagram.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_GenerateExpressionFromDiagram(object sender, RoutedEventArgs e)
        {
            renderedExpressionDisplay renderedExpression;
            // The expression must exist otherwise an error has occurred.
            if (d.GetExpression() != "")
            {
                // Rendering the expression because it is more appealing
                renderedExpression = new renderedExpressionDisplay(d.GetExpression(), "Rendered
Expression: ");
                renderedExpression.Show();
```

```
            }
            else
            {
                MessageBox.Show("The diagram does not exist. ", "Expression Error",
MessageBoxButton.OK, MessageBoxImage.Error);
            }
        }

        /// <summary>
        /// Draws a logic gate diagram from a user entered Boolean expression. Also allows
        /// users to constantly enter expressions without having to clear the canvas every
        /// time they would like a new diagram.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_GenerateDiagramFromExpression(object sender, RoutedEventArgs e)
        {
            var expressionInputDialog = new BooleanExpressionInputDialog();
            string expression;
            // Asking the user for a Boolean expression and storing the result.
            if (expressionInputDialog.ShowDialog() == true)
            {
                expression = expressionInputDialog.Result;
                // Validating the entered expression and drawing it if it is.
                if (d.IsExpressionValid(expression))
                {
                    // Clearing the diagram to avoid interference.
                    DiagramCanvas.Children.Clear();
                    d.ClearDiagram();
                    d.SetExpression(expression);
                    statusBar.Text = "Generated diagram from expression: " + expression;
                    d.DrawDiagram();
                    saveString = expression;
                }
                else
                {
                    MessageBox.Show("This expression is invalid. Please try again. ");
                }
            }
        }

        /// <summary>
        /// Takes the user-drawn diagram and replaces it with the minimised form of it.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_MinimiseDiagram(object sender, RoutedEventArgs e)
        {
            string expression = d.GetExpression();
            string minimisedExpression;
            // Ensuring that the expression exists, and the tree is not null to make sure
            // that minimisation can take place. Otherwise, an error has occurred.
            if (expression != "" && d.GetTree() != null)
            {
                // Clearing the canvas so that the new diagram does not interfere with the
                // old one. Resetting the diagram for the same reason.
                DiagramCanvas.Children.Clear();
                d.ClearDiagram();
                d.MinimiseExpression(expression);
                minimisedExpression = d.GetMinimisedExpression();
                d.SetExpression(minimisedExpression);
                d.DrawDiagram();
```

```
            }
            else
            {
                MessageBox.Show("Error Minimising diagram: Diagram does not exist. ",
"Minimisation error", MessageBoxButton.OK, MessageBoxImage.Error);
            }
        }

        /// <summary>
        /// Accepts a user entered Boolean expression and returns the rendered expression
        /// of the minimised result.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_MinimiseExpression(object sender, RoutedEventArgs e)
        {
            var expressionInputDialog = new BooleanExpressionInputDialog();
            renderedExpressionDisplay renderedExpression;
            string expression;
            // Asking the user to enter a Boolean expression.
            if (expressionInputDialog.ShowDialog() == true)
            {
                expression = expressionInputDialog.Result;
                // Validating the expression and minimising if it is valid.
                if (d.IsExpressionValid(expression))
                {
                    d.SetExpression(expression);
                    d.MinimiseExpression(expression);
                    // Showing the rendered minimised expression because it is more
                    // appealing.
                    renderedExpression = new
renderedExpressionDisplay(d.GetMinimisedExpression(), "Minimised Expression: ");
                    renderedExpression.Show();
                    statusBar.Text = "Minimised expression: " + expression;
                }
                else
                {
                    // The program has found that the entered expression is invalid. The
                    // should be notified of this.
                    MessageBox.Show("You have entered an invalid expression.", "Expression
input error", MessageBoxButton.OK, MessageBoxImage.Error);
                }

            }
        }

        /// <summary>
        /// Saving a user-drawn diagram on the main window canvas to a text file, so
        /// the user can save their work and continue at a later point.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_SaveDiagram(object sender, RoutedEventArgs e)
        {
            var saveFileDialog = new SaveFileDialog()
            {
                // The only extensions the program will accept. This is to stop any
                // problems of the user giving the program erroneous file formats.
                Filter = "Text file (*.txt)|*.txt|XML (*.xml)|*.xml|Expression file
(*.2B)|*.2B",
                DefaultExt = "Expression file (*.2B)|*.2B"
            };
```

```
            // Showing the savefile dialog to find the desired file path.
            saveFileDialog.ShowDialog();
            try
            {
                // Simply try writing all of the text to a file or an error has occurred and
                // the user will be given an error message.
                File.WriteAllText(saveFileDialog.FileName, saveString);
                statusBar.Text = "Saved diagram at the path " + saveFileDialog.FileName;
            }
            catch (Exception ex)
            {
                // The file could not be loaded by the program.
                MessageBox.Show($"Error saving diagram:\n{ex.Message}", "Load File Error",
MessageBoxButton.OK, MessageBoxImage.Error);
                e.Handled = true;
            }
        }

        /// <summary>
        /// Allows the user to load saved diagrams from text files so that they can
        /// continue working on the same diagram/expression.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_LoadDiagram(object sender, RoutedEventArgs e)
        {
            var openFileDialog = new OpenFileDialog()
            {
                Filter = "Text file (*.txt)|*.txt|XML (*.xml)|*.xml|Expression file
(*.2B)|*.2B",
                DefaultExt = "Expression file (*.2B)|*.2B"
            };
            // Showing the open file dialog to get the open path.
            openFileDialog.ShowDialog();
            try
            {
                // Reading all text from the file and this the expression to be drawn
                // to the main window. Only if it is valid otherwise an error has happened.
                saveString = File.ReadAllText(openFileDialog.FileName);
                if (d.IsExpressionValid(saveString))
                {
                    d.SetExpression(saveString);
                    d.DrawDiagram();
                    statusBar.Text = "Loaded diagram from " + openFileDialog.FileName;
                }
                else
                {
                    MessageBox.Show("This expression is invalid. Please try again. ");
                }
            }
            catch (Exception ex)
            {
                // The file could not be opened by the program.
                MessageBox.Show($"Error opening file:\n{ex.Message}", "Open File Error",
MessageBoxButton.OK, MessageBoxImage.Error);
                e.Handled = true;
            }
        }

        /// <summary>
        /// Allows the user to export a png of their drawn diagram. This can be put into
        /// schoolwork or in a worksheet of some kind.
```

```
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_ExportDiagram(object sender, RoutedEventArgs e)
        {
            var saveFileDialog = new SaveFileDialog()
            {
                Filter = "PNG (*.png)|*.png",
                DefaultExt = ".png"
            };
            if (saveFileDialog.ShowDialog() != true)
            {
                return;
            }

            if (d.GetExpression() != "")
            {
                var bounds = VisualTreeHelper.GetDescendantBounds(DiagramCanvas);
                var bitmap = new RenderTargetBitmap((int)bounds.Width, (int)bounds.Height, 96,
96, PixelFormats.Pbgra32);
                var drawingVisual = new DrawingVisual();
                using (DrawingContext dc = drawingVisual.RenderOpen())
                {
                    var visualBrush = new VisualBrush(DiagramCanvas);
                    var tmp = new Point();
                    var rect = new Rect(tmp, bounds.Size);
                    dc.DrawRectangle(visualBrush, null, rect);
                }

                bitmap.Render(drawingVisual);
                var png = new PngBitmapEncoder();
                png.Frames.Add(BitmapFrame.Create(bitmap));
                using (Stream stream = File.Create(saveFileDialog.FileName))
                {
                    png.Save(stream);
                }
            }
            else
            {
                MessageBox.Show("Could not calculate bounds: Diagram does not exist. \n",
"Diagram Export Error", MessageBoxButton.OK, MessageBoxImage.Error);
                e.Handled = true;

            }
        }

        /// <summary>
        /// Closes the application.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void MenuItem_CloseApp(object sender, RoutedEventArgs e)
        {
            this.Close();
        }

        /// <summary>
        /// Updates the state of the logic gate diagram when a click has occurred on the
        /// main window canvas. This allows the diagrams to be interactive.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
```

```
        private void DiagramCanvas_MouseDown(object sender, MouseButtonEventArgs e)
        {
            // If the tree does not exist then the user should be prompted to draw a
            // diagram first. Otherwise update the state of the diagram.
            if (d.GetTree() != null)
            {
                d.UpdateWires();
                statusBar.Text = "Updated the state of the diagram.";
            }
            else
            {
                statusBar.Text = "Please draw a diagram first. ";
            }
        }

        /// <summary>
        /// Clears the truth table canvas of any previously generated truth tables.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void Button_Click_TT(object sender, RoutedEventArgs e)
        {
            //Checking if there is a truth table currently on the canvas.
            if (TruthTableCanvas.Children.Count == 0)
            {
                statusBar.Text = "Please draw a truth table first. ";
            }
            else
            {
                TruthTableCanvas.Children.Clear();
                statusBar.Text = "Cleared the current truth table. ";
            }
        }

        /// <summary>
        /// Displays information on the AND logic gate.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void ANDInformation(object sender, EventArgs e)
        {
            var info = new GateInformation(d, "AND gate");
            info.Show();
        }

        /// <summary>
        /// Displays information on the OR logic gate.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void ORInformation(object sender, EventArgs e)
        {
            var info = new GateInformation(d, "OR gate");
            info.Show();
        }

        /// <summary>
        /// Displays information on the NOT logic gate.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void NOTInformation(object sender, EventArgs e)
```

```
        {
            var info = new GateInformation(d, "NOT gate");
            info.Show();
        }

        /// <summary>
        /// Displays information on the XOR logic gate.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void XORInformation(object sender, EventArgs e)
        {
            var info = new GateInformation(d, "XOR gate");
            info.Show();
        }

        /// <summary>
        /// Displays information on the NAND logic gate.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void NANDInformation(object sender, EventArgs e)
        {
            var info = new GateInformation(d, "NAND gate");
            info.Show();
        }

        /// <summary>
        /// Displays information on the NOR logic gate.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void NORInformation(object sender, EventArgs e)
        {
            var info = new GateInformation(d, "NOR gate");
            info.Show();
        }
    }
}
```

App.xaml

This defines the application wide resources for the program.

```
<Application x:Class="_2BNOR_2B.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:_2BNOR_2B" xmlns:local1="clr-namespace:_2BNOR_2B.Code"
             StartupUri="MainWindow.xaml">
    <Application.Resources>
        <!--The following converter renders Boolean expression that are entered by the user.
        This is application wide as the converter is used in multiple windows at different
        points within the program, such as the expression input dialog and the rendered
        expression window.-->
        <local1:BooleanConverter x:Key="booleanConverter"></local1:BooleanConverter>
    </Application.Resources>
</Application>
```

Testing

The following sections covers all of the different test that have been conducted to measure the function of the program. The order of tests is given by the list below:

- Developmental testing
    - Expression validation
    - Diagram drawing
        - Drawing from expressions
        - Diagram interactivity
    - Truth table generation
        - Generation from Expressions
        - Generation from diagrams
    - Diagram and expression minimisation
        - Minimising user-entered expressions
        - Minimising user-drawn diagrams
    - File handling
        - Saving diagrams
        - Loading diagrams
        - Exporting diagrams
    - User interface testing
        - Pop-up menus
        - Expression rendering

When each test is conducted, the information will be recorded:

- Test ID: The reference number for the test. This will be structured in the same order as shown above. This allows for unique referencing for the tests. These are zero-indexed.
- Test Description: A small message indicating the subject being tested and the nature of what the test is targeting.
- Test data: Any data that must be supplied to the program to conduct the test. Examples include user-entered Boolean expressions.
- Expected outcome: The output that the program should give with the test data.
- Outcome: The actual outcome given by the program. If the result is a valid result, then the test will be coloured in green. A failed test will be indicated by a red background with the exception given. Any fixes will be indicated in blue.
- Comments [where necessary]: this is a small message added, after the test have been conducted. These are added only where necessary.

If an error occurs in one of the tests being conducted, then a solution will be provided with a complete code snippet of the updated method. Please refer to the technical solution for the original method of the same name as the technical solution was written before these tests were conducted. The test will be repeated to ensure that the problem is fixed. This will be indicated in the table.

It should also be noted that, all evidence of the test will be given after a particular group of tests have been conducted. This evidence should be referred to by the test ID.

Developmental testing

Validation

The following tests focus on the validation of Boolean expressions within the program.

Test ID	Description	Test data	Expected outcome	Outcome
0.0.0	Invalid characters	“”	Error message shown	Invalid Operation Exception: Stack empty
Test Comment: The program only considers “ “ within validation. Therefore**,** a check must be put in to ensure that the validation routine discounts empty strings.				

Solution: This creates the following validation method which discounts any empty strings.

```
        public bool IsExpressionValid(string expression)
        {
            string removedWhitespace = RemoveWhitespace(expression, "");
            if (removedWhitespace == "")
            {
                return false;
            }
            string postfix = ConvertInfixtoPostfix(expression);
            if (IsSequential(expression) && InvalidCharacters(expression) &&
ValidateBrackets(expression))
            {
                // Imposing an input limit for Petrick's method. This is a choice because
                // it means that the term to implicant map has a unique character as the
                // key.
                if (GetNumberOfInputs(expression, true) > maxNumberOfInputs)
                {
                    return false;
                }
                else
                {
                    // Ensuring that the postfix is valid postfix. Covers expressions
                    // such as "(A.)" which pass the other checks.
                    return PostfixCheck(postfix);
                }
            }
            // If the expression is of the incorrect form, then discard immediately as valid
            // postfix cannot be produced from the expression.
            else
            {
                return false;
            }
        }
```

`0.0.0	Invalid characters	“”	Error message shown	Error message shown
0.0.1	Invalid characters	“ “	Error message shown	Error message shown
0.0.2	Invalid characters	[9]	Error message shown	Error message shown
0.0.3	Invalid characters however the expression is in the correct form	[8,9]	Error message shown	Error message shown
0.0.4	Invalid characters but valid inputs	[A+B]	Error message shown	Error message shown
0.0.5	Valid expression but invalid character has been added	(A+B)_	Error message shown	Error message shown
0.0.6	Valid expression but invalid character in erroneous position	(A_+B)	Error message shown	Error message shown
0.0.7	Expression in the correct form but constant is invalid	(A+9)	Error message shown	Error message shown
0.0.8	Expression in the correct form but constant is invalid.	(9+A)	Error message shown	Error message shown
Section comment: The following tests (id’s 0.1.0 to 0.1.40) cover expressions that have invalid inputs in the expression. For an expression to be valid, the inputs must have sequential ASCII values starting from ‘A’ (65).				
0.1.0	Valid form of expression but inputs are not sequential	(A+C)	Error message shown	Error message shown
0.1.1	Valid form of expression but inputs are not sequential	(C+A)	Error message shown	Error message shown

0.1.2	Valid form of expression but inputs are not sequential	(B+C)	Error message shown	Error message shown
0.1.3	Valid form of expression but inputs do not start from 'A'	(C+B)	Error message shown	Error message shown
0.1.4	Valid form of expression but inputs do not start from 'A'	(C.D)+E	Error message shown	Error message shown
0.1.5	Valid form of expression but inputs do not start from 'A'	(E.C)+D	Error message shown	Error message shown
0.1.6	Inputs do not start from 'A' and are in reverse order.	(E.D)+C	Error message shown	Error message shown
0.1.7	Invalid case of inputs but valid form of expressions	(a+b)	Error message shown	Error message shown
0.1.8	Invalid case of one input but valid form of expression	(A+b)	Error message shown	Error message shown
0.1.9	Invalid case of one input bit valid form of expression	(a+B)	Error message shown	Error message shown
0.1.10	Invalid case of inputs and also incorrect order	(C+b)	Error message shown	Error message shown
0.1.11	Invalid case of inputs and also incorrect order	(b+C)	Error message shown	Error message shown
0.1.12	Valid inputs	(A+B)	No error message shown	No error message shown
0.1.13	Valid inputs	(B+A)	No error message shown	No error message shown
0.1.14	Valid inputs	A+B	No error message shown	No error message shown
0.1.15	Valid inputs	B+A	No error message shown	No error message shown
0.1.16	Invalid sequence of inputs, next operand should be 'B' not 'C'	A+C	Error message shown	Error message shown
0.1.17	Invalid sequence of inputs, operand 'C' should be 'B'	C+A	Error message shown	Error message shown
0.1.18	Invalid inputs as they should start from 'A'	B+C	Error message shown	Error message shown
0.1.19	Invalid inputs as they should start from 'A'	C+B	Error message shown	Error message shown
0.1.20	Invalid inputs as they should start from 'A'	C.D+E	Error message shown	Error message shown
0.1.21	Invalid inputs as they should start from 'A'. These are also in reverse order	E.D+C	Error message shown	Error message shown
0.1.22	Incorrect case of inputs	a+b	Error message shown	Error message shown
0.1.23	Incorrect case of second operand	A+b	Error message shown	Error message shown
0.1.24	Incorrect case of first operands	a+B	Error message shown	Error message shown
0.1.25	Incorrect case and order of operands. They also do not start from 'A'	C+b	Error message shown	Error message shown
0.1.26	Valid inputs but constants should not be considered	(A+1)	No error message shown	No error message shown
0.1.27	Valid inputs but constants should not be considered.	(A+0)	No error message shown	No error message shown

0.1.28	Valid inputs but constants should not be considered	(1+A)	No error message shown	No error message shown
0.1.29	Valid inputs but constants should not be considered.	(0+A)	No error message shown	No error message shown
0.1.30	Invalid inputs as they should start with 'A'	(B+0)	Error message shown	Error message shown
0.1.31	Invalid inputs as they should start with 'A'	(B+1)	Error message shown	Error message shown
0.1.32	Invalid inputs as they should start with 'A'	(0+B)	Error message shown	Error message shown
0.1.33	Invalid inputs as they should start with 'A'	(1+B)	Error message shown	Error message shown
0.1.34	Valid inputs	A+1	No error message shown	No error message shown
0.1.35	Valid inputs	A+0	No error message shown	No error message shown
0.1.36	Valid inputs	0+A	No error message shown	No error message shown
0.1.37	Invalid inputs as they should start with 'A'	B+0	Error message shown	Error message shown
0.1.38	Invalid inputs as they should start with 'A''	B+1	Error message shown	Error message shown
0.1.39	Invalid inputs as they should start with 'A'	0+B	Error message shown	Error message shown
0.1.40	Invalid inputs as they should start with 'A'	1+B	Error message shown	Error message shown

Section comment: The following tests (id's 0.2.0 to 0.2.41) cover the position and number of brackets within entered expressions. The expressions do not contain invalid characters or symbols but do focus on the order in which the brackets are entered. This is to create expression that do not create valid postfix expressions as brackets change precedence of the expression.

0.2.0	Empty set of brackets	()	Error message shown	InvalidOperationException: stack empty

Test comment: The test passes the check but does not have any operands, this is because the check for the number of operands does not consider a lower limit. To make this correct the number of operands – number of constants within the expression must be greater than or equal to 1 for a valid expression

Solution: This addition is within the main validation method and extends the number of operands check.

```
        public bool IsExpressionValid(string expression)
        {
            string removedWhitespace = RemoveWhitespace(expression, "");
            if (removedWhitespace == "")
            {
                return false;
            }
            if (IsSequential(removedWhitespace) && InvalidCharacters(removedWhitespace)
                && ValidateBrackets(removedWhitespace))
            {
                // Imposing an input limit for Petrick's method. This is a choice because
                // it means that the term to implicant map has a unique character as the
                // key.
                int numberOfInputs = GetNumberOfInputs(removedWhitespace, false) -
GetNumberOfConstants(removedWhitespace);
                if (GetNumberOfInputs(removedWhitespace, true) > maxNumberOfInputs ||
numberOfInputs < 1)
                {
                    return false;
                }
```

```
                else
                {
                    string postfix = ConvertInfixtoPostfix(removedWhitespace);
                    // Ensuring that the postfix is valid postfix. Covers expressions
                    // such as "(A.)" which pass the other checks.
                    return PostfixCheck(postfix);
                }
            }
            // If the expression is of the incorrect form, then discard immediately as valid
            // postfix cannot be produced from the expression.
            else
            {
                return false;
            }
        }
```

`0.2.0	Empty set of brackets	()	Error message shown	Error message shown
0.2.1	Empty set of brackets	( )	Error message shown	Error message shown
0.2.2	No closing bracket has been entered into the expression	(!A	Error message shown	Error message shown
0.2.3	No opening bracket has been entered into the expression	!A)	Error message shown	Error message shown
0.2.4	Brackets have been entered in the incorrect position	()!A	Error message shown	Expression passes through

Test comment: This is because currently the program is checking whether or an expression can be converted from postfix to infix an vice-versa. This means that this expression can pass the valid postfix, but the brackets are point

Solution: A new method has been written to check for the empty bracket case. Any expression that contains this is discounted. This is done using the following regular expression:

```
        private static bool CheckBracketPosition(string expression)
        {
            var bracketCheck = new Regex(@"\(\)");
            var match = bracketCheck.Match(expression);
            if (match.Success)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
```

This is implemented as follows

```
...
if (CheckBracketPosition(expression))
{
    return false;
}
else
{
    // Imposing an input limit for Petrick's method. This is a choice because
    // it means that the term to implicant map has a unique character as the
    // key.
    int numberOfInputs = GetNumberOfInputs...
```

`0.2.4	Brackets have been entered in the incorrect postion	()!A	Error message shown	Error message shown
0.2.5	Brackets have been entered in the incorrect position	!A()	Error message shown	Error message shown

0.2.6	Brackets have been entered to enclose the input	!(A)	No error message shown	No error message shown
0.2.7	Brackets do not enclose the operand	(!)A	Error message shown	Error message shown
0.2.8	Brackets do not enclose the entire expression	A(+B)	Error message shown	Expression passes through

Test comment: The regular expression written for test 0.2.4 is not restrictive enough as it allows incomplete operators through. Therefore, a further check must be carried out.

Solution: The CheckBracketPostion method has a new regular expression to cover more cases. The main validation method has also been restructured to cope with these new changes.

```
        private static bool CheckBracketPosition(string expression)
        {
            var bracketCheck = new Regex(@"\(\)");
            var match = bracketCheck.Match(expression);
            if (match.Success)
            {
                return true;
            }
            else
            {
                var gateCheck = new Regex(@"\([A-Z0-1]*[!.+^][A-Z0-1]\)");
                match = gateCheck.Match(expression);
                if (match.Success)
                {
                    return true;
                }
                else
                {
                    return false;
                }
            }
        }

        public bool IsExpressionValid(string expression)
        {
            string removedWhitespace = RemoveWhitespace(expression, "");
            if (removedWhitespace == "")
            {
                return false;
            }
            if (IsSequential(removedWhitespace) && InvalidCharacters(removedWhitespace) &&
ValidateBrackets(removedWhitespace))
            {
                // Imposing an input limit for Petrick's method. This is a choice because
                // it means that the term to implicant map has a unique character as the
                // key.
                int numberOfInputs = GetNumberOfInputs(removedWhitespace, false) -
GetNumberOfConstants(removedWhitespace);
                if (GetNumberOfInputs(removedWhitespace, true) > maxNumberOfInputs ||
numberOfInputs < 1)
                {
                    return false;
                }
                else
                {
                    string postfix = ConvertInfixtoPostfix(removedWhitespace);
                    // Ensuring that the postfix is valid postfix. Covers expressions
                    // such as "(A.)" which pass the other checks.
                    bool postfixCheck = PostfixCheck(postfix);
                    if (CheckBracketPosition(removedWhitespace))
```

```
                    {
                        return false;
                    }
                    else
                    {
                        return postfixCheck;
                    }
                }
            }
            // If the expression is of the incorrect form, then discard immediately as valid
            // postfix cannot be produced from the expression.
            else
            {
                return false;
            }
        }
```

`0.2.8	Brackets do not enclose the entire expression	A(+B)	Error message shown	Error message shown
0.2.9	Brackets do not enclose the entire expression	(A+)B	Error message shown	Error message shown
0.2.10	Brackets do not enclose the expression	A(+)B	Error message shown	Error message shown
0.2.11	Brackets enclose one operand	(A)+B	No error message shown	Error message shown

Test comment: The program is allowing this expression through because the second regular expression within the CheckBracketPosition method returns the incorrect value for its match. This is a simple fix.

Solution: Change the return value for the second regular expression being checked within the CheckBracketPositionMethod. This is done below:

```
        private static bool CheckBracketPosition(string expression)
        {
            var bracketCheck = new Regex(@"\(\)");
            var match = bracketCheck.Match(expression);
            if (match.Success)
            {
                return false;
            }
            else
            {
                var gateCheck = new Regex(@"\([A-Z0-1]*[!.+^][A-Z0-1]\)");
                match = gateCheck.Match(expression);
                if (match.Success)
                {
                    var operandCheck = new Regex(@"\([A-z0-1][\^.+][A-z0-1]\)");
                    match = operandCheck.Match(match.Value);
                    if (match.Success)
                    {
                        return true;
                    }
                    else
                    {
                        return false;
                    }
                }
                else
                {
                    return true;
                }
            }
        }
```

`0.2.11	Brackets enclose one operands	(A)+B	No error message shown	No error message shown
0.2.12	Brackets enclose one operand	A+(B)	No error message shown	No error message shown
0.2.13	Extra number of brackets	(A)+(B)	No error message shown	No error message shown
0.2.14	Extra number of brackets	((A)+(B))	No error message shown	No error message shown
0.2.15	Extra number of brackets	((A)+B)	No error message shown	No error message shown
0.2.16	One valid set of brackets, but enclosed by an invalid set	((A+B)	Error message shown	Error message shown
0.2.17	One valid set of brackets, but enclosed by an invalid set	(A+B))	Error message shown	Error message shown
0.2.18	Incomplete brackets	((A+(B)	Error message shown	Error message shown
0.2.19	Invalid brackets enclosed by valid brackets	((A(+)(B)))	Error message shown	Error message shown
0.2.20	Valid brackets but in an invalid position	A(+)(B)	Error message shown	Error message shown
0.2.21	Valid brackets but in an invalid position	(A)(+)B	Error message shown	Error message shown
0.2.22	All terms separated by brackets	(A)(+)(B)	Error message shown	Error message shown
0.2.23	Erroneous empty brackets	(A+B)()	Error message shown	Error message shown
0.2.24	Erroneous empty brackets	(A()+B)	Error message shown	Error message shown
0.2.25	Erroneous empty brackets	(()A+B)	Error message shown	Error message shown
0.2.26	Erroneous empty brackets	(A+()B)	Error message shown	Error message shown
0.2.27	Erroneous empty brackets	(A+B())	Error message shown	Error message shown
0.2.28	Erroneous empty brackets	(()A+B())	Error message shown	Error message shown
0.2.29	Erroneous empty brackets	(A()+()B)	Error message shown	Error message shown
0.2.30	Erroneous empty brackets	(()A+()B)	Error message shown	Error message shown
0.2.31	Erroneous empty brackets	(A()+B())	Error message shown	Error message shown
0.2.32	Erroneous empty brackets an missing second operand	(A()+())	Error message shown	Error message shown
0.2.33	Erroneous empty brackets	(()A+())	Error message shown	Error message shown
0.2.34	Erroneous empty brackets	(()A)+())	Error message shown	Error message shown
0.2.35	Too many closing brackets and erroneous empty brackets	(()A+((B))))	Error message shown	Error message shown
0.2.36	Too many opening brackets	((((((A+B))	Error message shown	Error message shown
0.2.37	Erroneous empty brackets and missing first operand	(()+B())	Error message shown	Error message shown
0.2.38	Erroneous empty brackets and missing first operand	(()+()B)	Error message shown	Error message shown
0.2.39	Brackets with no operators	(A)	No error message shown	No error message shown
0.2.40	Erroneous empty brackets	A()	Error message shown	Error message shown
0.2.41	Erroneous empty brackets	()A	Error message shown	Error message shown
Section comment: The following tests (id's 0.3.0 to 0.3.13) cover expressions that have incomplete operands.				
0.3.0	Incorrect number of operands	A+	Error message shown	Error message shown
0.3.1	Incorrect number of operands	+B	Error message shown	Error message shown
0.3.2	Incorrect number of operands	(A)+	Error message shown	Error message shown
0.3.3	Incorrect number of operands	+(A)	Error message shown	Error message shown
0.3.4	Incorrect number of operands	B+	Error message shown	Error message shown

0.3.5	Incorrect number of operands	(B)+	Error message shown	Error message shown
0.3.6	Incorrect number of operands	+(B)	Error message shown	Error message shown
0.3.7	Incorrect number of operands	(A.B)+	Error message shown	Error message shown
0.3.8	Incorrect number of operands	(A.)+C	Error message shown	Error message shown
0.3.9	No operands in expression	+	Error message shown	Error message shown
0.3.10	No operands in expression	()+()	Error message shown	Error message shown
0.3.11	No operands in expression	(+)	Error message shown	Error message shown
0.3.12	No operands in expression	()+	Error message shown	Error message shown
0.3.13	No operands in expression	+()	Error message shown	Error message shown
Section comment: The following tests (id's 0.4.0 to 0.4.6) cover expressions that do not fit into the previously defined categories, but they must be included for fully in-depth testing.				
0.4.0	Testing the upper limit for the number of inputs	(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L))	No error message shown	No error message shown
0.4.1	Too many unique inputs for the program	(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L)+(M))	Error message shown	Expression passes through
Test comment: The max number of inputs that the program will accept has been set to the incorrect value.				
Solution: Simply change to the property within the diagram to be one less than the original value:				

```
readonly int maxNumberOfInputs = 12;
```

`0.4.1	Too many unique inputs for the program	(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L)+(M))	Error message shown	Error message shown
0.4.2	Ensuring constants do not count towards the total number of inputs.	(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L)+(1))	No error message shown	Expression is rejected.
Test comment: The method GetNumberOfInputs also counts constants within the expression. The number of constants must be subtracted in the count.				
Solution: This simple fix is given in the new validation method below:				

```
        public bool IsExpressionValid(string expression)
        {
            string removedWhitespace = RemoveWhitespace(expression, "");
            if (removedWhitespace == "")
            {
                return false;
            }
            if (IsSequential(removedWhitespace) && InvalidCharacters(removedWhitespace) &&
ValidateBrackets(removedWhitespace))
            {
                // Imposing an input limit for Petrick's method. This is a choice because
                // it means that the term to implicant map has a unique character as the
                // key.
                int numberOfInputs = GetNumberOfInputs(removedWhitespace, false) -
GetNumberOfConstants(removedWhitespace);
                if (numberOfInputs > maxNumberOfInputs || numberOfInputs < 1)
                {
                    return false;
                }
                else
                {
                    string postfix = ConvertInfixtoPostfix(removedWhitespace);
                    // Ensuring that the postfix is valid postfix. Covers expressions
                    // such as "(A.)" which pass the other checks.
                    bool postfixCheck = PostfixCheck(postfix);
                    if (!CheckBracketPosition(removedWhitespace))
                    {
```

```
                        return false;
                    }
                    else
                    {
                        return postfixCheck;
                    }
                }
            }
            // If the expression is of the incorrect form, then discard immediately as valid
            // postfix cannot be produced from the expression.
            else
            {
                return false;
            }
        }
```

`0.4.2	Ensuring constants do not count towards the total number of inputs	(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L)+(1))	No error message shown	No error message shown
0.4.3	Constant only expression	0+1	No error message shown	Expression is rejected.

Test comment: The lower limit of the input check is causing the expression to be discounted.

Solution: Remove the lower limit check in the main validation method.

```
        public bool IsExpressionValid(string expression)
        {
            string removedWhitespace = RemoveWhitespace(expression, "");
            if (removedWhitespace == "")
            {
                return false;
            }
            if (IsSequential(removedWhitespace) && InvalidCharacters(removedWhitespace) &&
ValidateBrackets(removedWhitespace))
            {
                // Imposing an input limit for Petrick's method. This is a choice because
                // it means that the term to implicant map has a unique character as the
                // key.
                int numberOfInputs = GetNumberOfInputs(removedWhitespace, false) -
GetNumberOfConstants(removedWhitespace);
                if (numberOfInputs > maxNumberOfInputs)
                {
                    return false;
                }
                else
                {
                    string postfix = ConvertInfixtoPostfix(removedWhitespace);
                    // Ensuring that the postfix is valid postfix. Covers expressions
                    // such as "(A.)" which pass the other checks.
                    bool postfixCheck = PostfixCheck(postfix);
                    if (!CheckBracketPosition(removedWhitespace))
                    {
                        return false;
                    }
                    else
                    {
                        return postfixCheck;
                    }
                }
            }
            // If the expression is of the incorrect form, then discard immediately as valid
            // postfix cannot be produced from the expression.
            else
            {
```

```
                return false;
            }
        }
```

`0.4.3	Constant only expression	0+1	No error message shown	No error message shown
0.4.4	Constant only expression	1+0	No error message shown	No error message shown
0.4.5	Constant only expression	0+0	No error message shown	IndexOutOfRangeException

Test comment: The program is using the label to create the nodes within the GenerateBinaryTreeFromExpression method. This means that '0' (48) – 65 creates a negative index which gives the IndexOutOfRangeException.

Solution: Change the GenerateBinaryTreeFromExpression method to not use the subtraction to find the correct node. This creates the following code. The GetInputWithSameLabel method has also been changed to throw an exception an not return null.

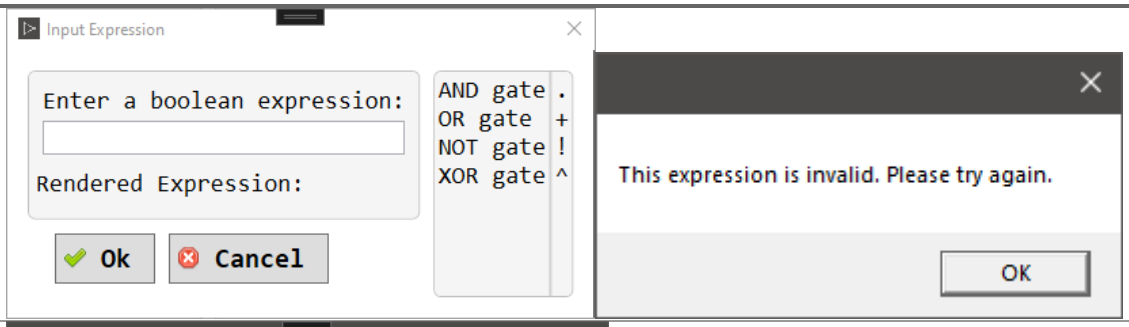
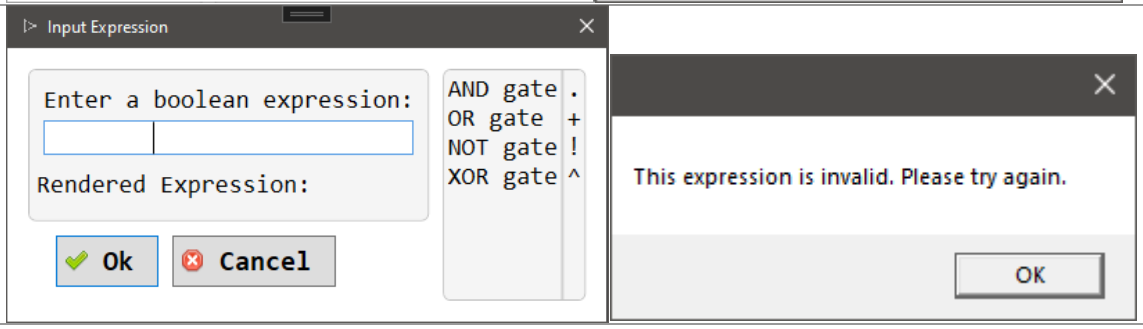
```
        private Element GetInputWithSameLabel(char label)
        {
            foreach (Element e in elements)
            {
                if (e.GetLabel() == label)
                {
                    return e;
                }
            }
            throw new Exception("Could not find element. ");
        }
        private void GenerateBinaryTreeFromExpression(string inputExpression)
        {
            string postfixExpression = ConvertInfixtoPostfix(inputExpression);
            inputs = new Element[GetNumberOfInputs(inputExpression, false)];
            var nodeStack = new Stack<Element>();
            elements = new Element[inputExpression.Length + 1];
            Element nodeToAdd;
            Element leftChild;
            Element rightChild;
            Element tmp;
            int elementID = 0;
            int i = 0;
            string elementName;
            string inputsAdded = "";
            foreach (char c in postfixExpression)
            {
                if (char.IsLetter(c) && char.IsUpper(c) || char.IsNumber(c))
                {
                    nodeToAdd = new Element(elementID, c);
                    inputs[i] = nodeToAdd;
                    i++;
                    if (char.IsNumber(c))
                    {
                        nodeToAdd.SetState(c - 48);
                    }
                    if (inputsAdded.Contains(c) == false)
                    {
                        nodeToAdd.SetInstances(1);
                        inputsAdded += c;
                    }
                    else
                    {
                        tmp = GetInputWithSameLabel(c);
                        tmp.AddInstance();
                    }
```

```
                }
                else if (c == '!')
                {
                    rightChild = nodeStack.Pop();
                    nodeToAdd = new Element("not_gate", elementID, null, rightChild);
                    nodeToAdd.SetInstances(1);
                    rightChild.parent = nodeToAdd;
                }
                else
                {
                    // Any logic gate is a binary operator, so pop two items and these are
                    // the operands for the current Boolean operation.
                    rightChild = nodeStack.Pop();
                    leftChild = nodeStack.Pop();
                    elementName = gateNames[Array.IndexOf(booleanOperators, c)];
                    nodeToAdd = new Element(elementName, elementID, leftChild, rightChild);
                    nodeToAdd.SetInstances(1);
                    leftChild.parent = nodeToAdd;
                    rightChild.parent = nodeToAdd;
                }
                nodeStack.Push(nodeToAdd);
                elements[elementID] = nodeToAdd;
                elementID++;
            }
            rootNode = nodeStack.Pop();
        }
```

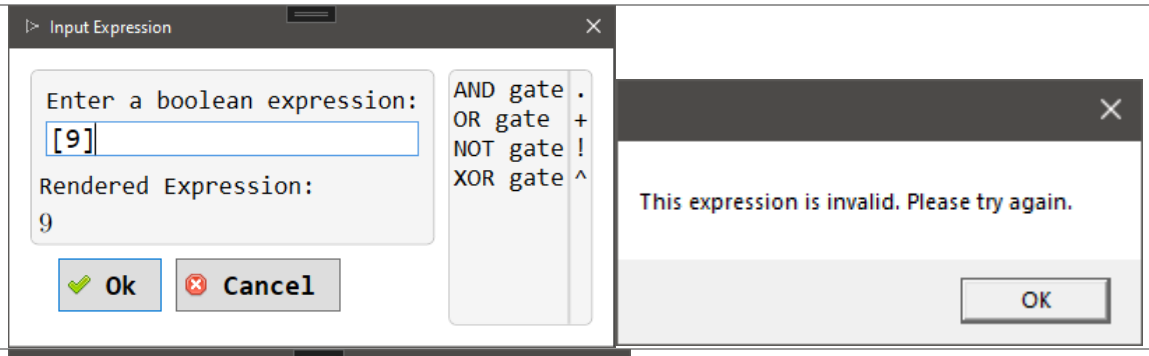
`0.4.5	Constant only expression	0+0	No error message shown	No error message shown
0.4.6	Constant only expression	1+1	No error message shown	No error message shown

Evidence of tests

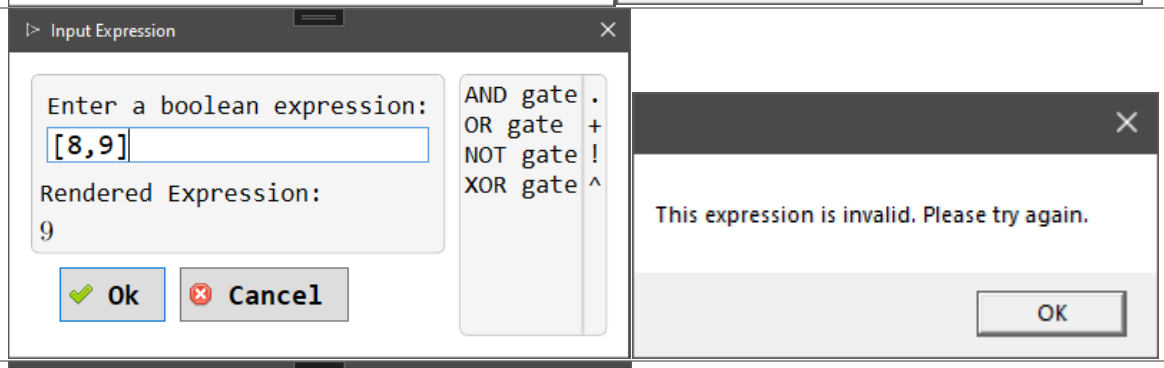
Evidence of the tests is given below. Although the error message is the same, each test has been conducted separately to each other.

Test ID	Evidence
0.0.0	
0.0.1	

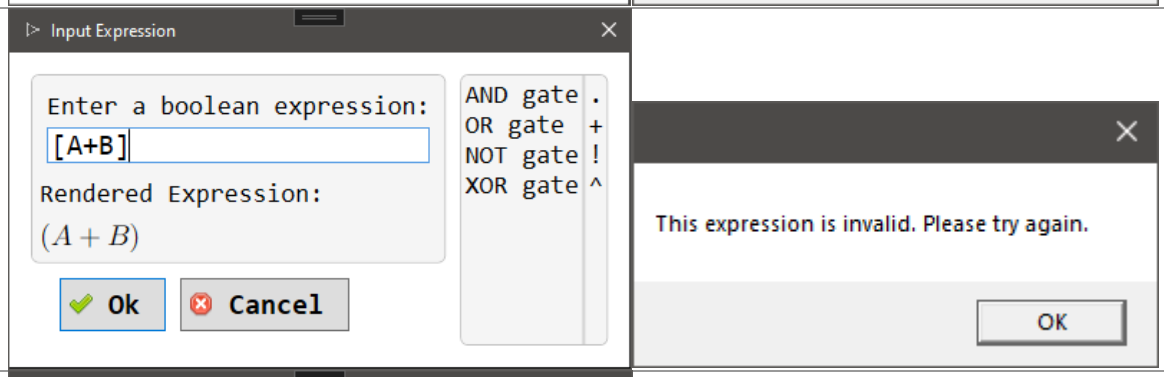
0.0.2



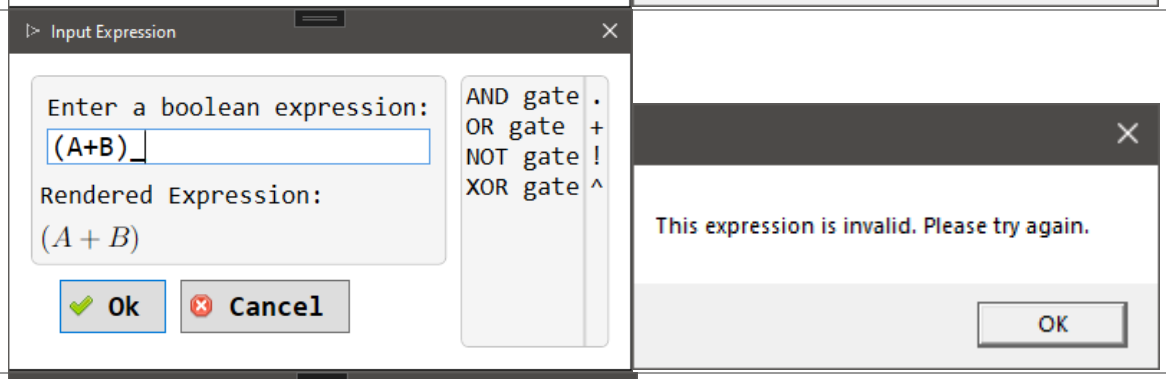
0.0.3



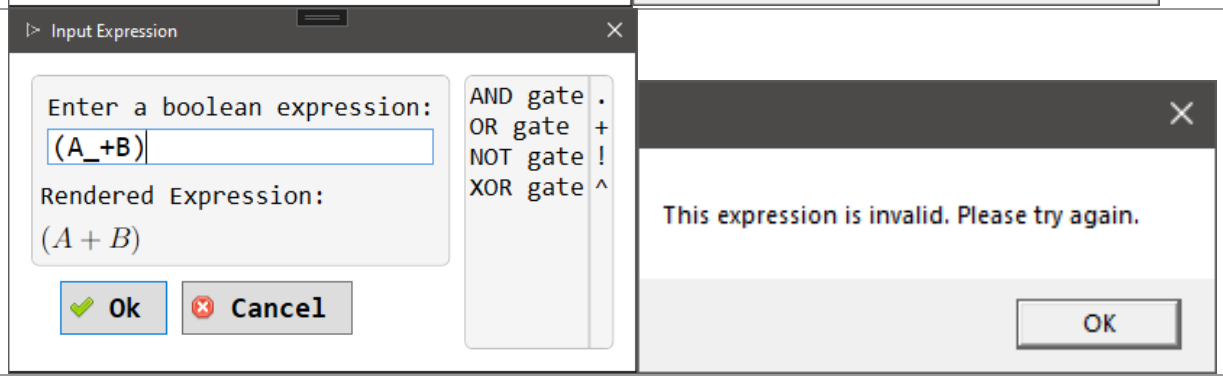
0.0.4



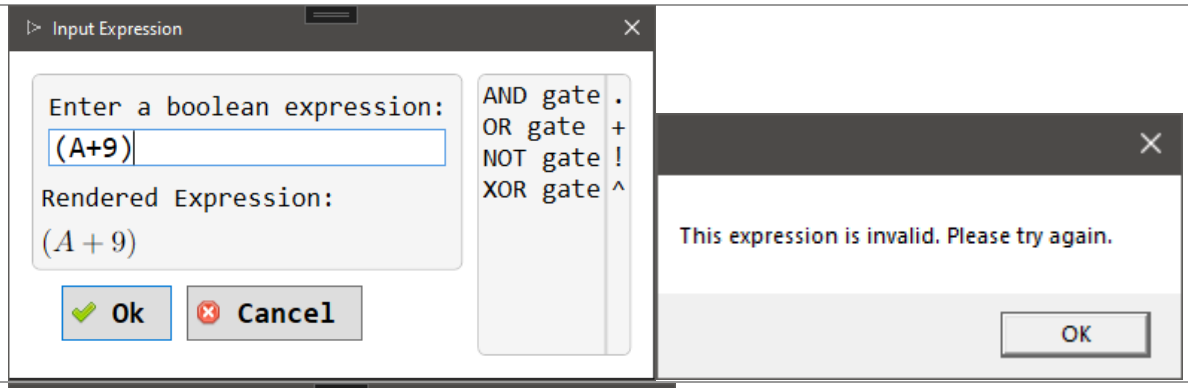
0.0.5



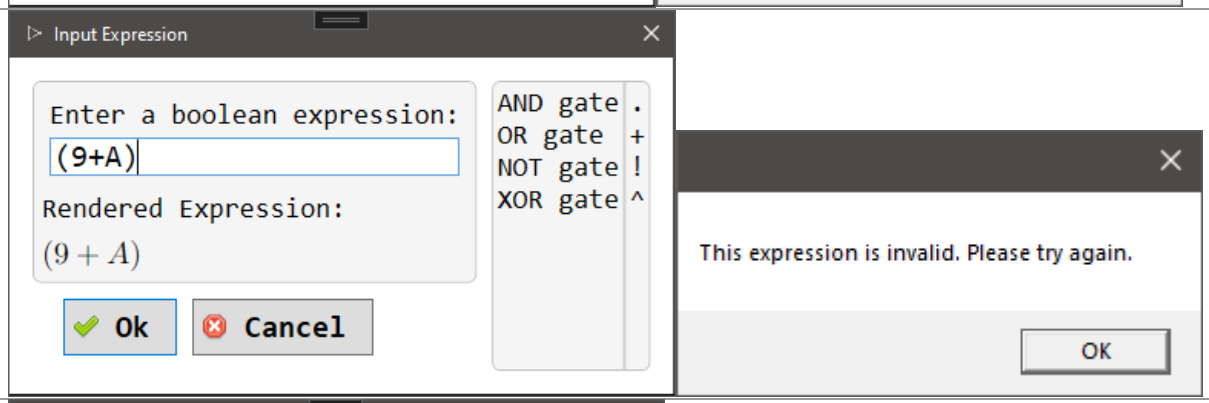
0.0.6



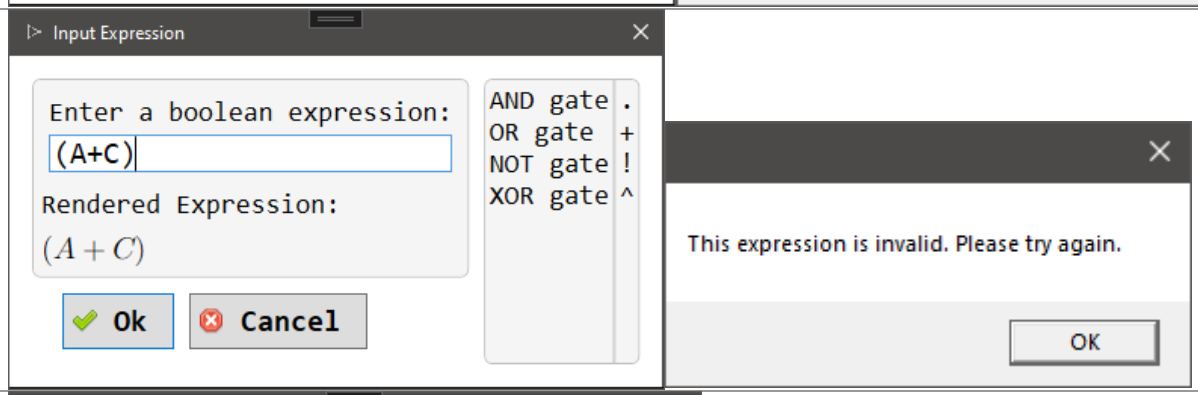
0.0.7



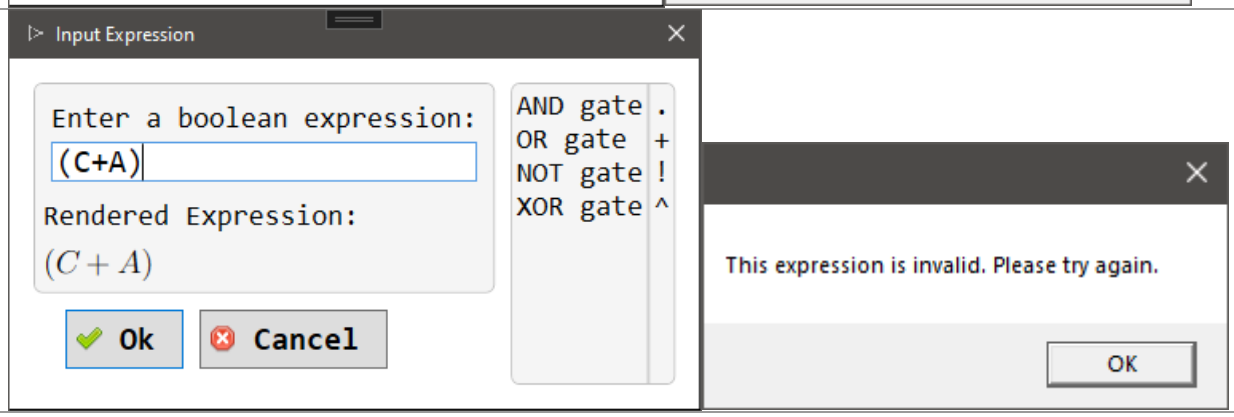
0.0.8



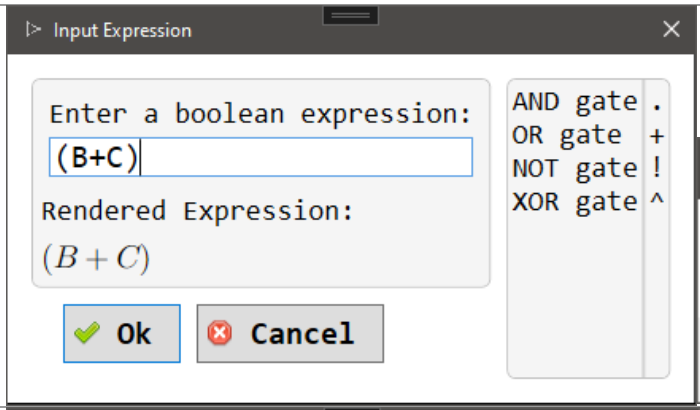
0.1.0

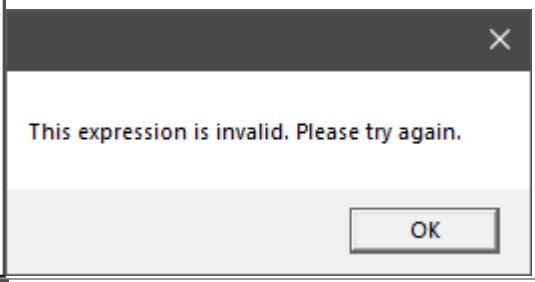


0.1.1

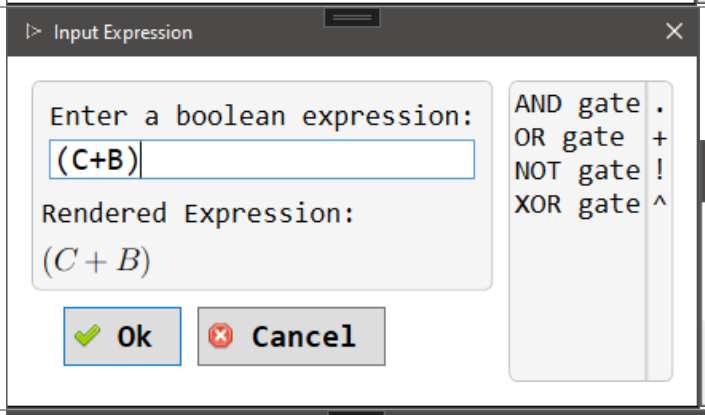


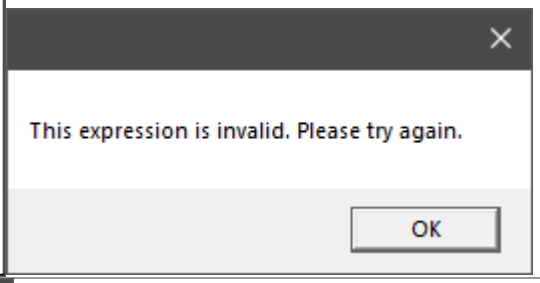
0.1.2



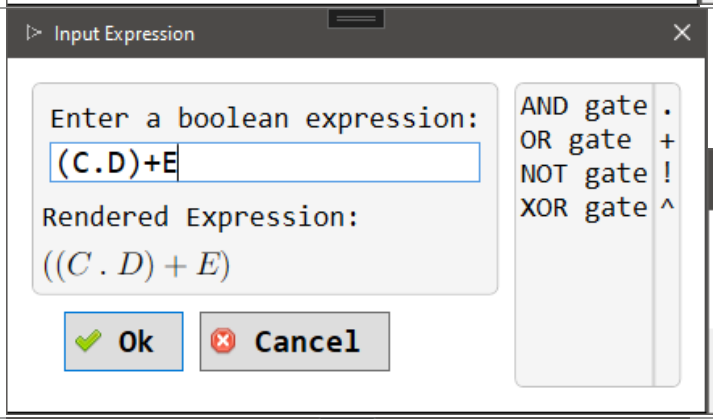


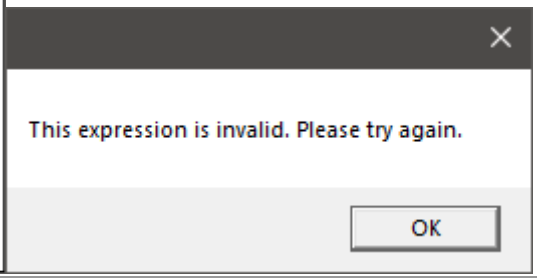
0.1.3



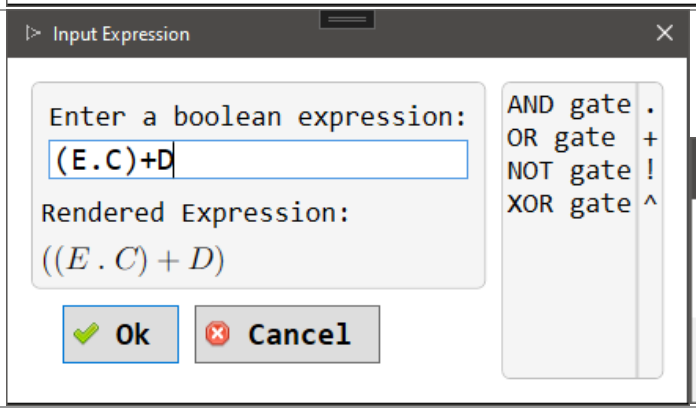


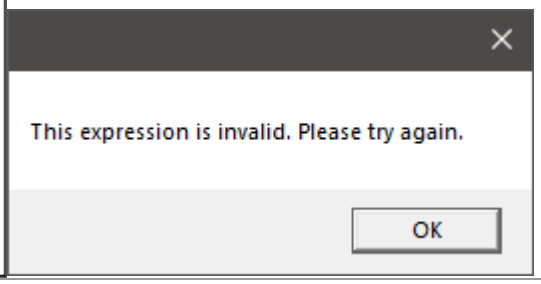
0.1.4



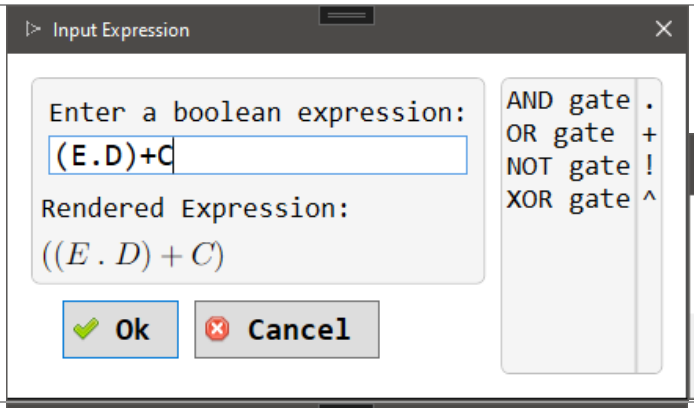


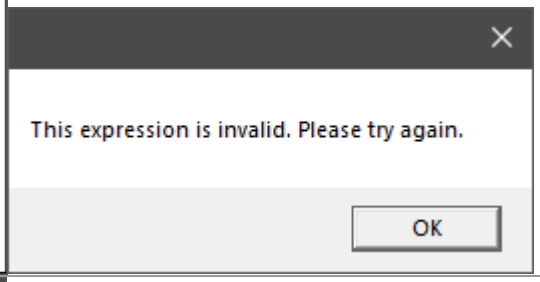
0.1.5



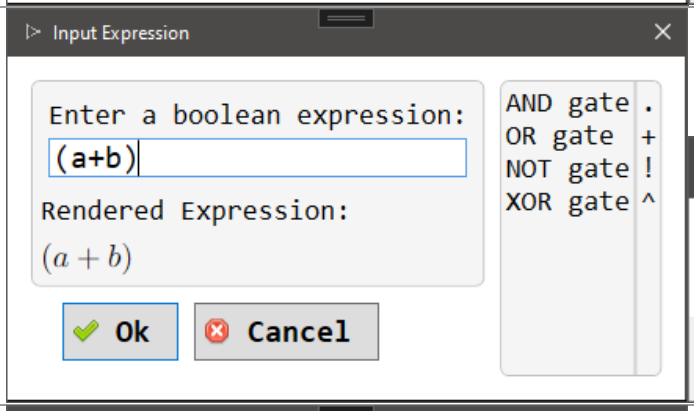


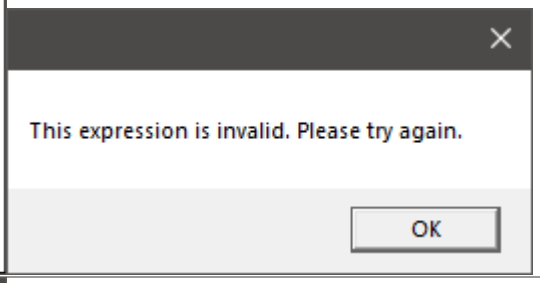
0.1.6



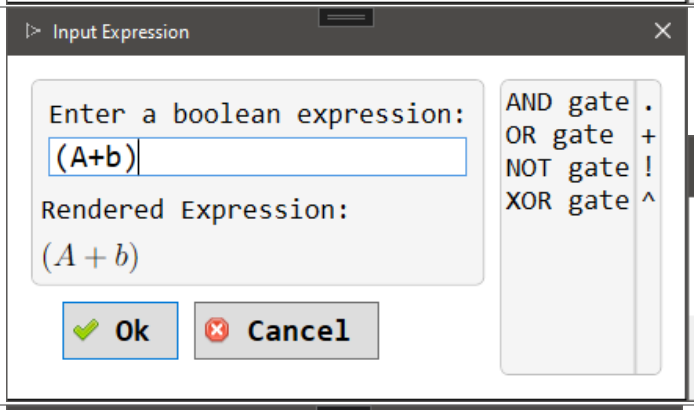


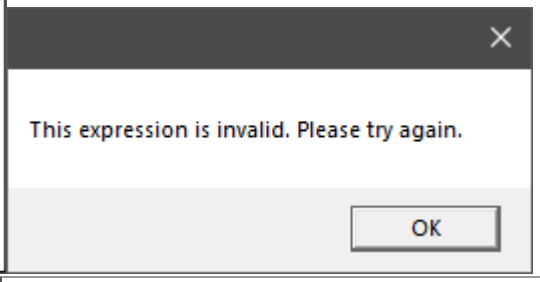
0.1.7



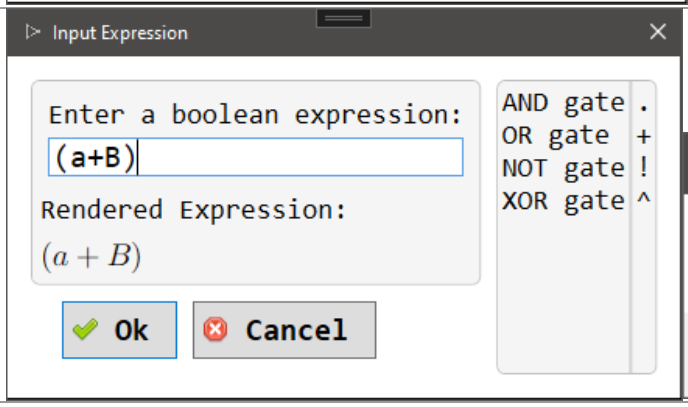


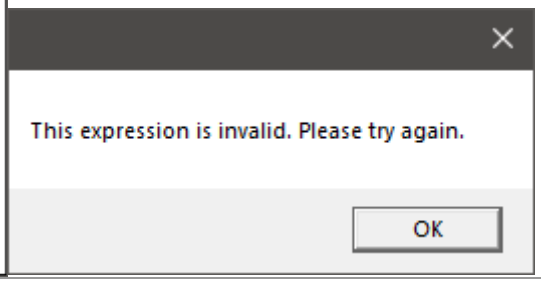
0.1.8



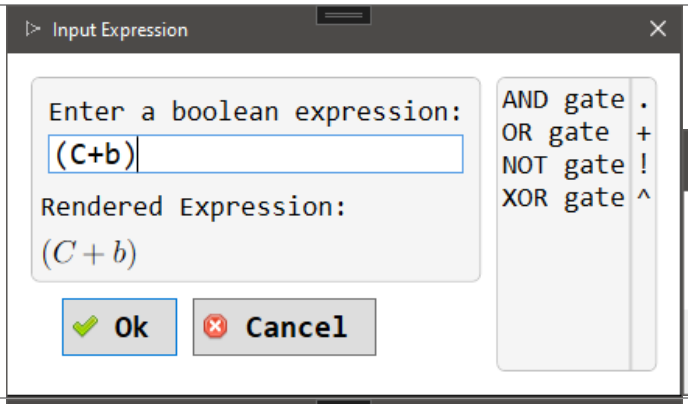


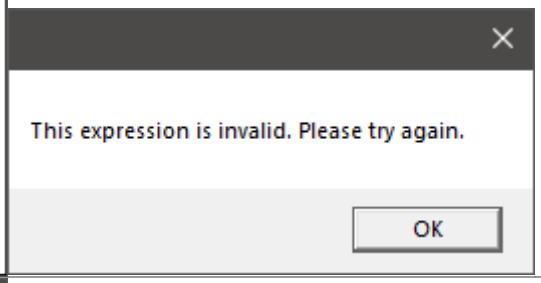
0.1.9



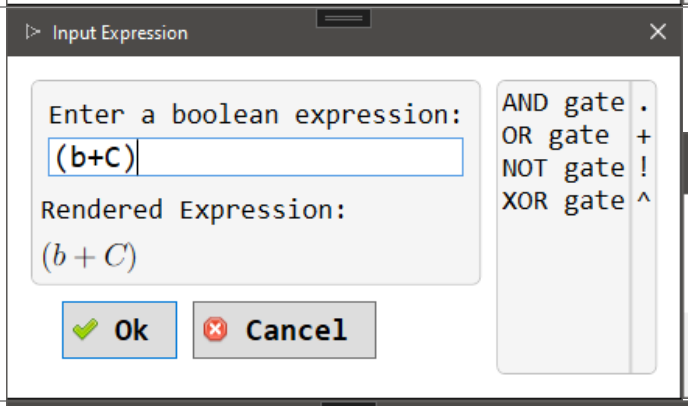


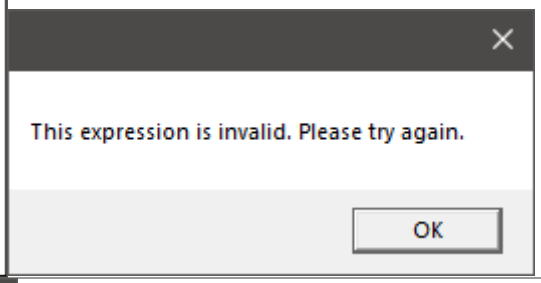
0.1.10



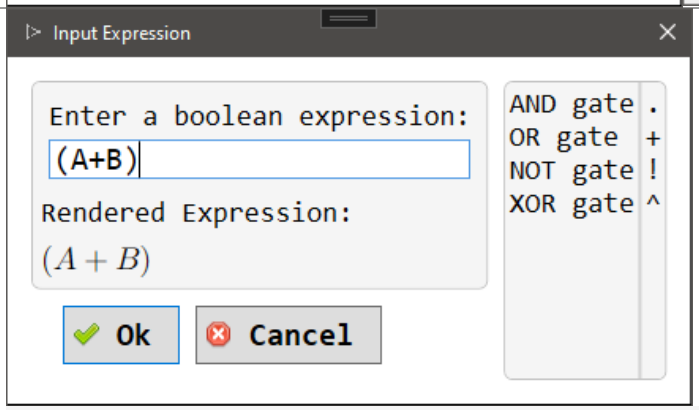


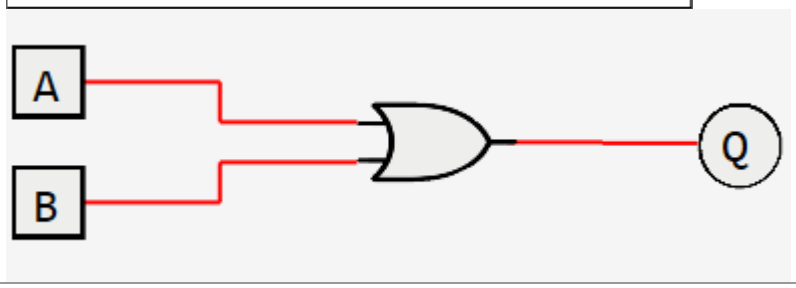
0.1.11



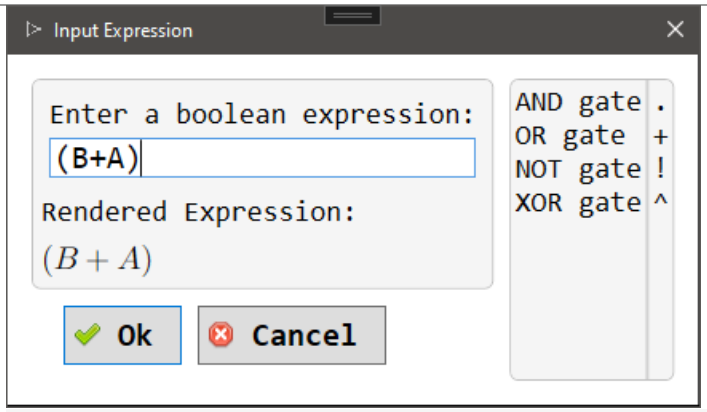


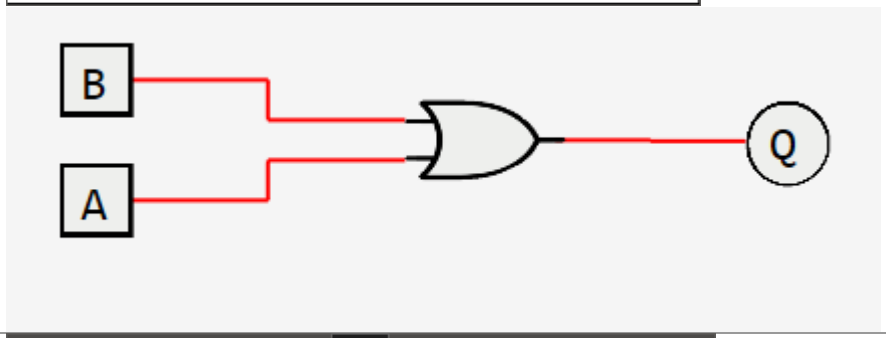
0.1.12



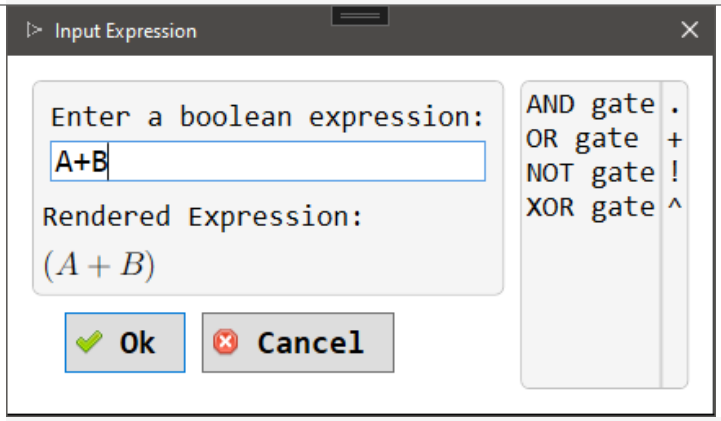


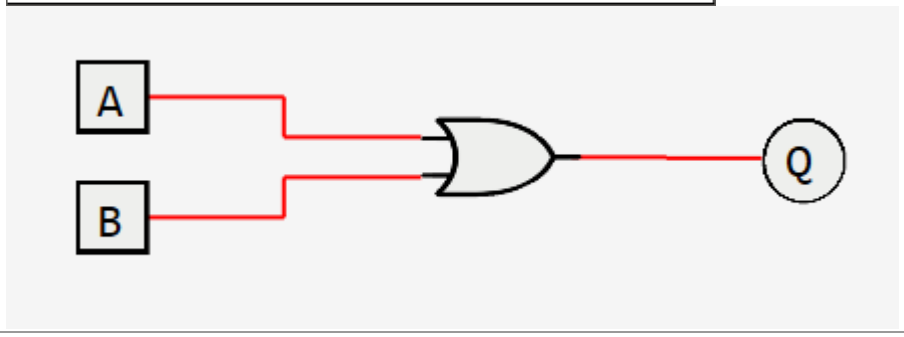
0.1.13


Input Expression
Enter a boolean expression:
(B+A)
Rendered Expression:
$(B + A)$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

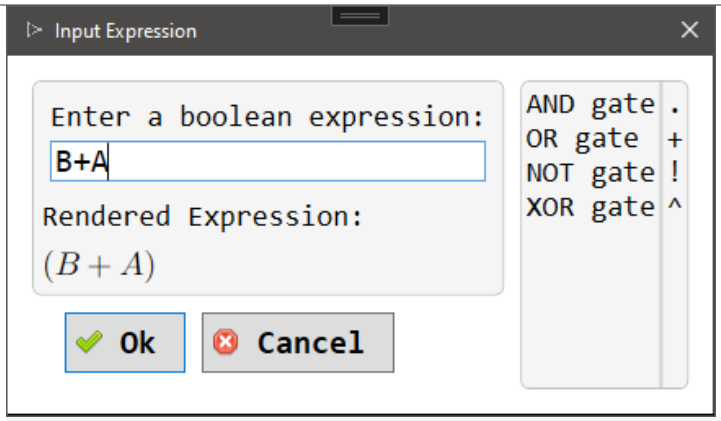

B
A
Q

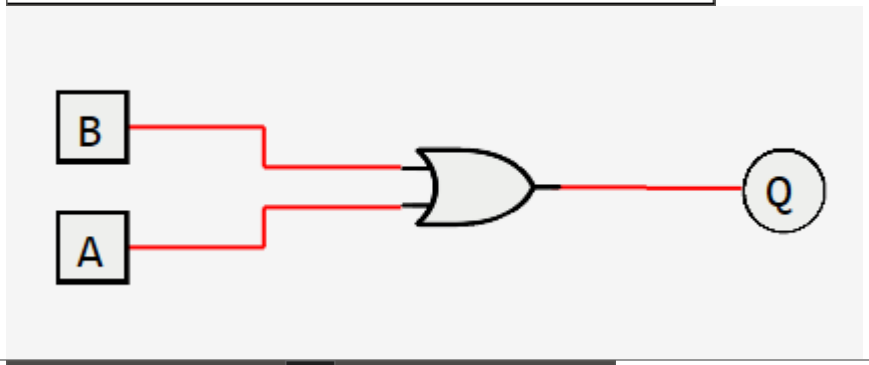
0.1.14


Input Expression
Enter a boolean expression:
A+B
Rendered Expression:
$(A + B)$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

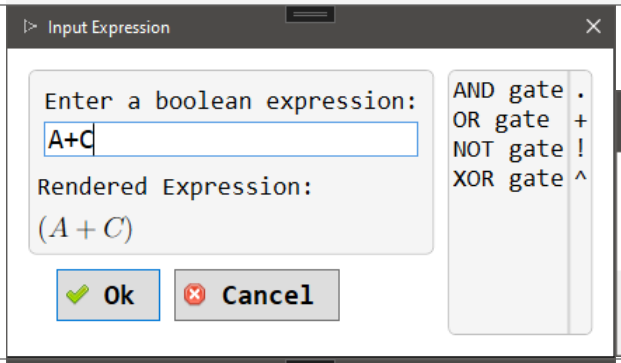

A
B
Q

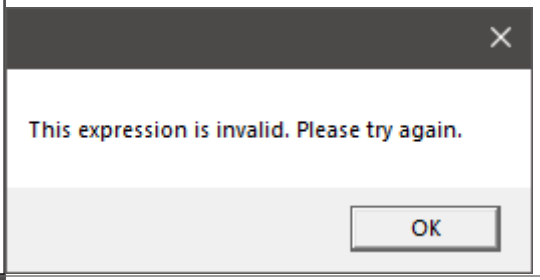
0.1.15



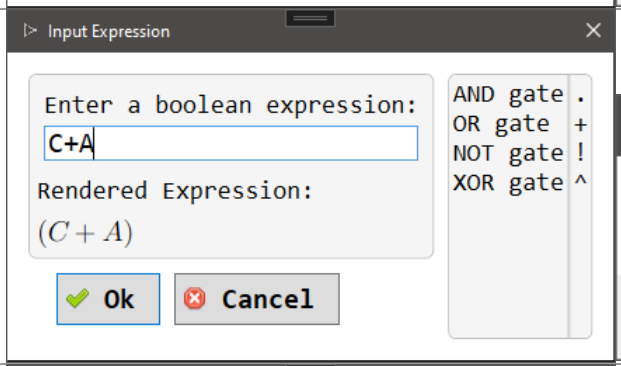


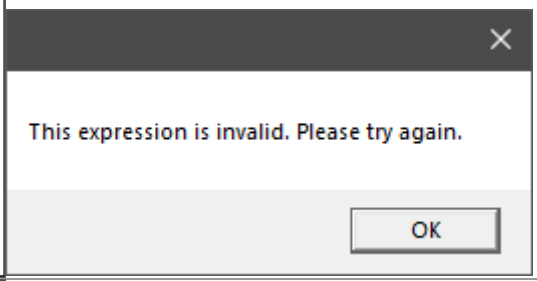
0.1.16



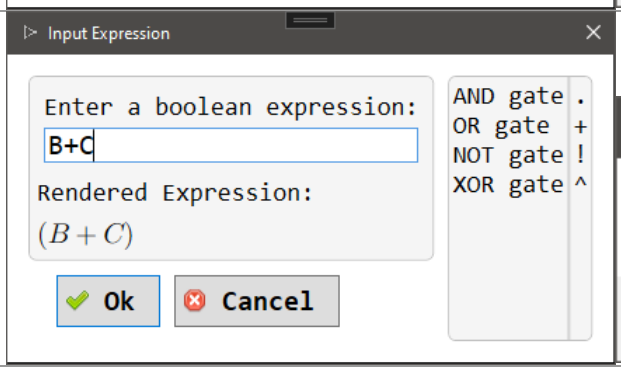


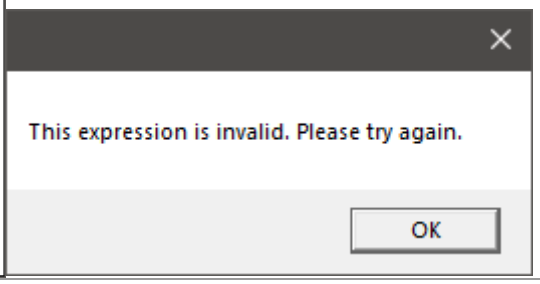
0.1.17



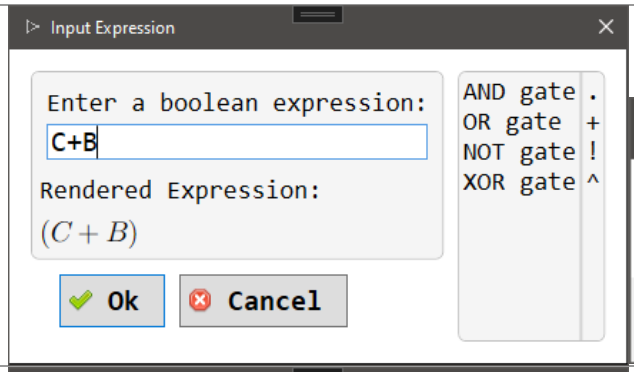


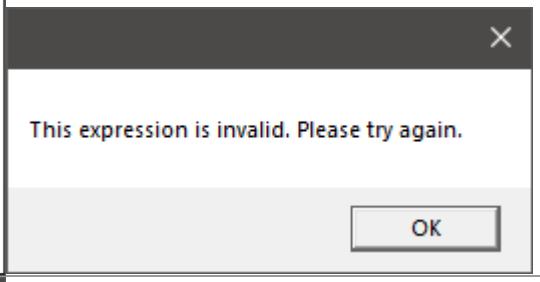
0.1.18



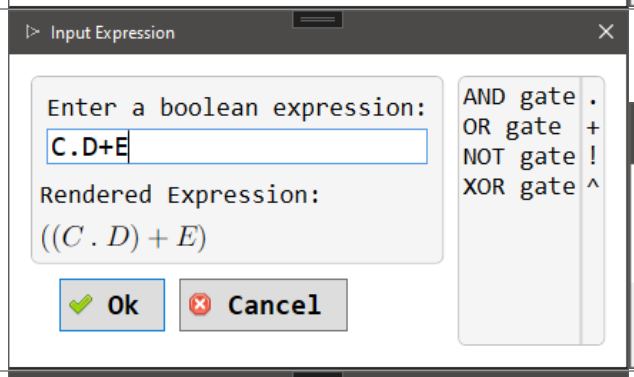


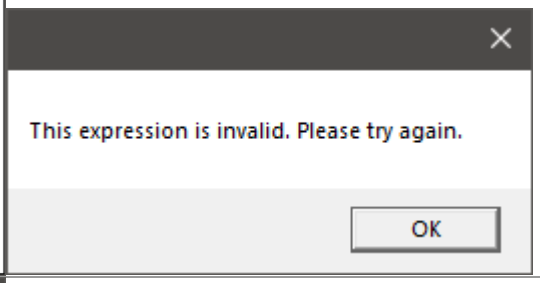
0.1.19



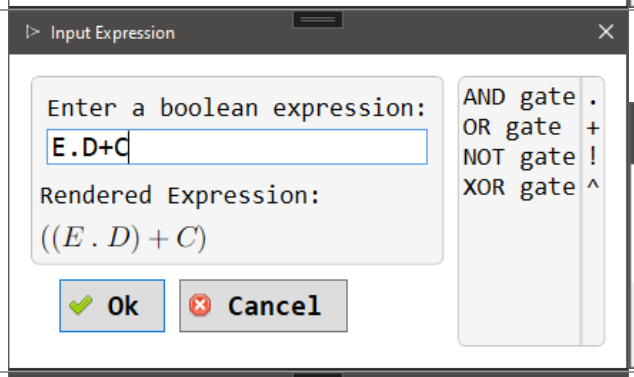


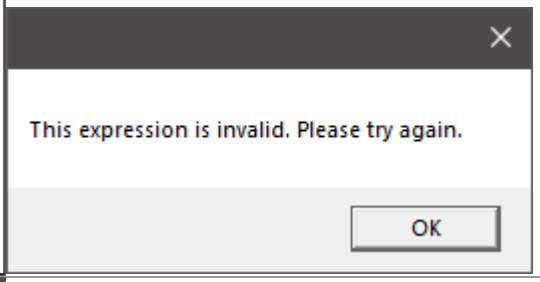
0.1.20



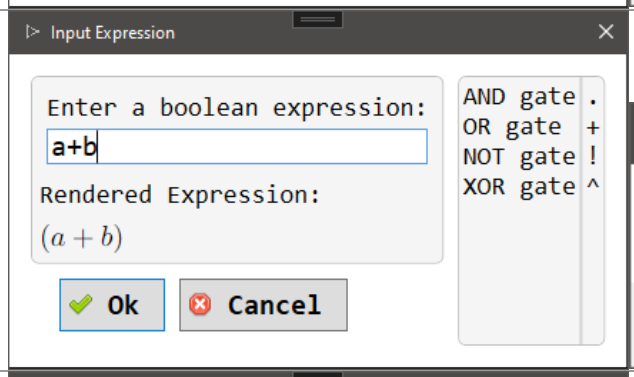


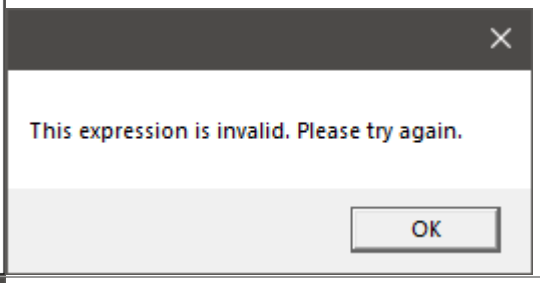
0.1.21



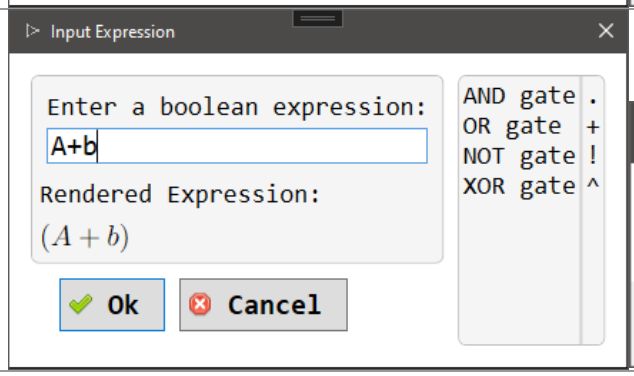


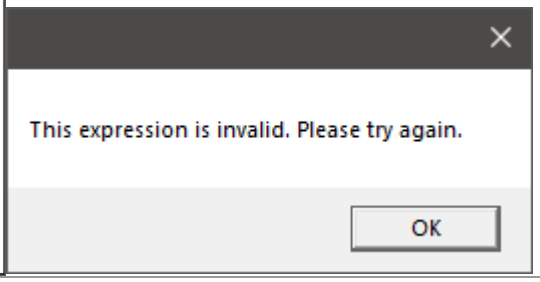
0.1.22



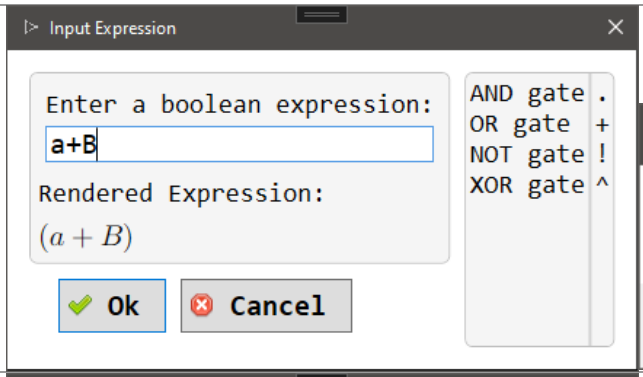


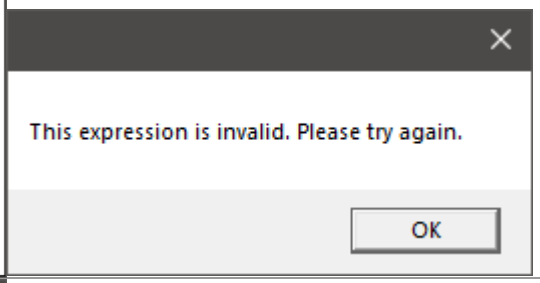
0.1.23



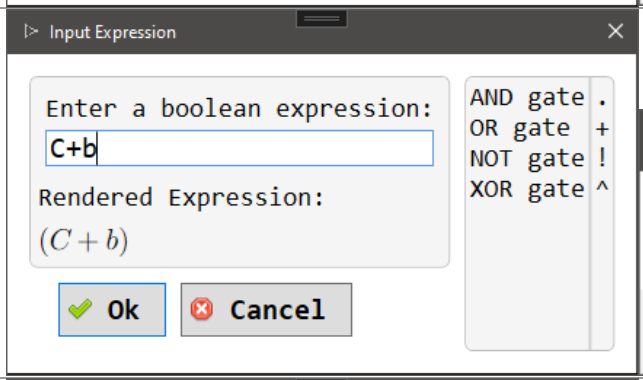


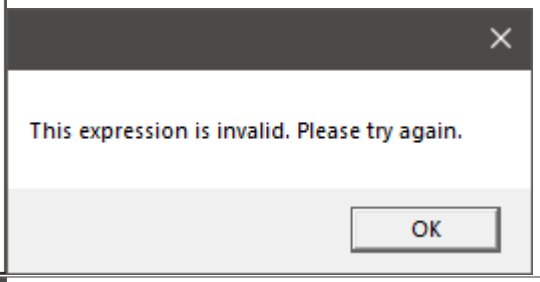
0.1.24



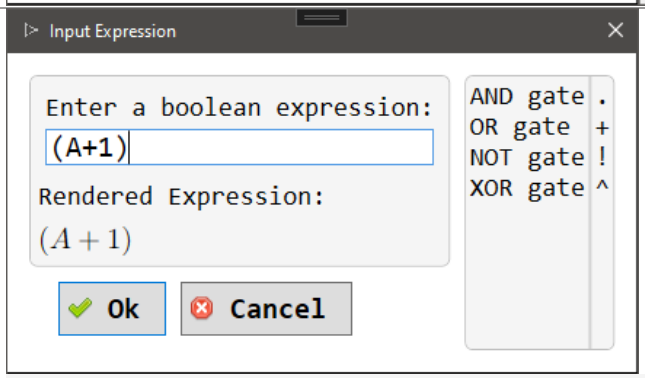


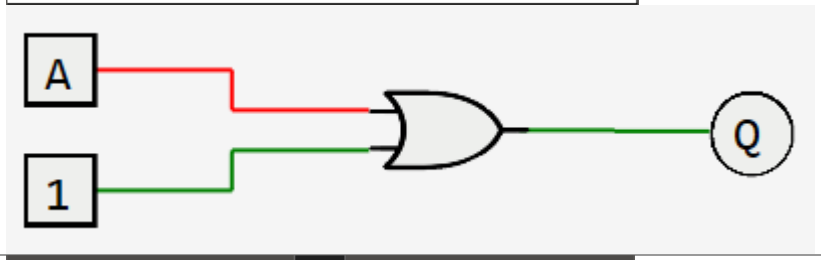
0.1.25



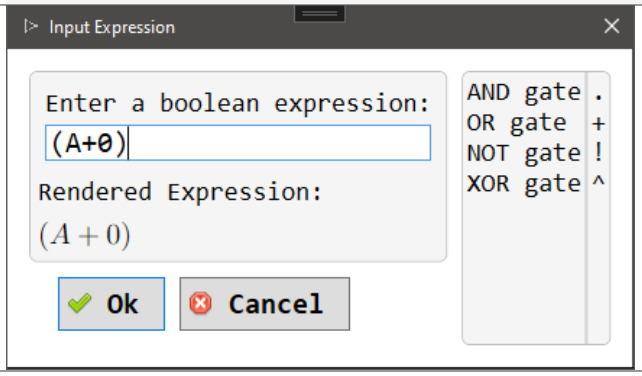


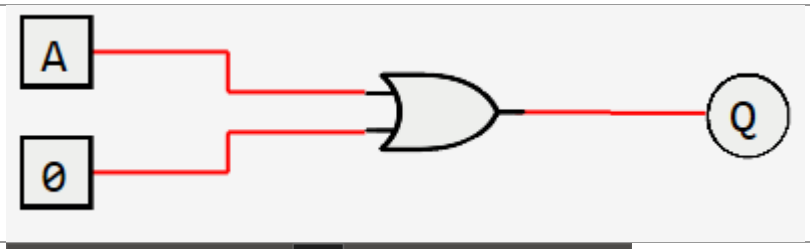
0.1.26



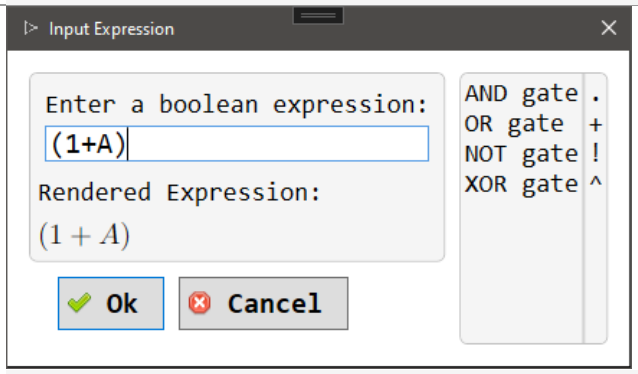


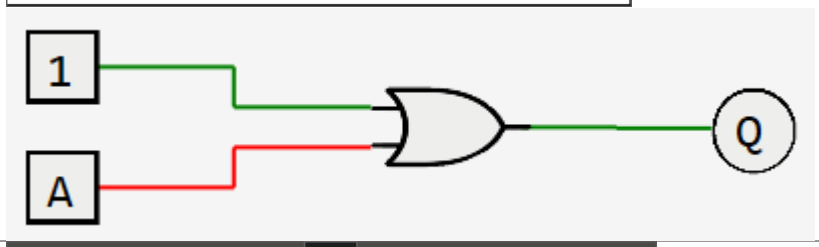
0.1.27



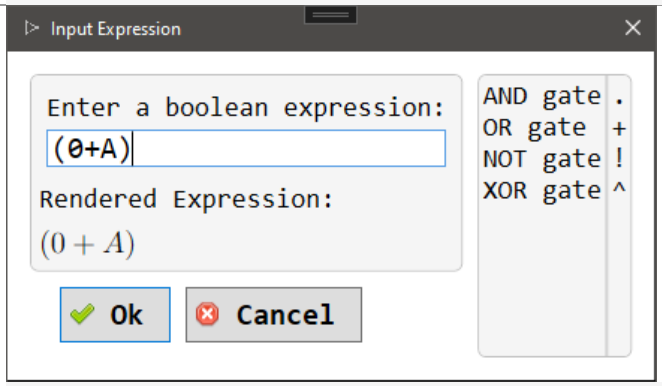


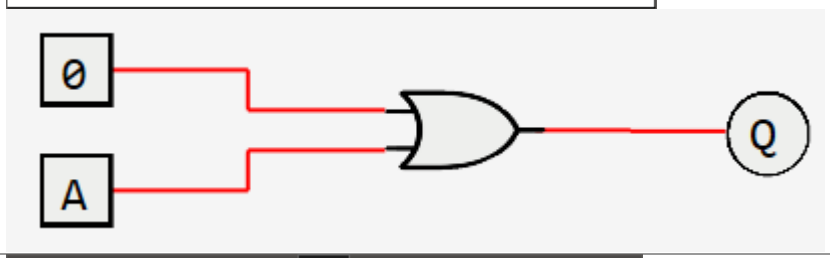
0.1.28



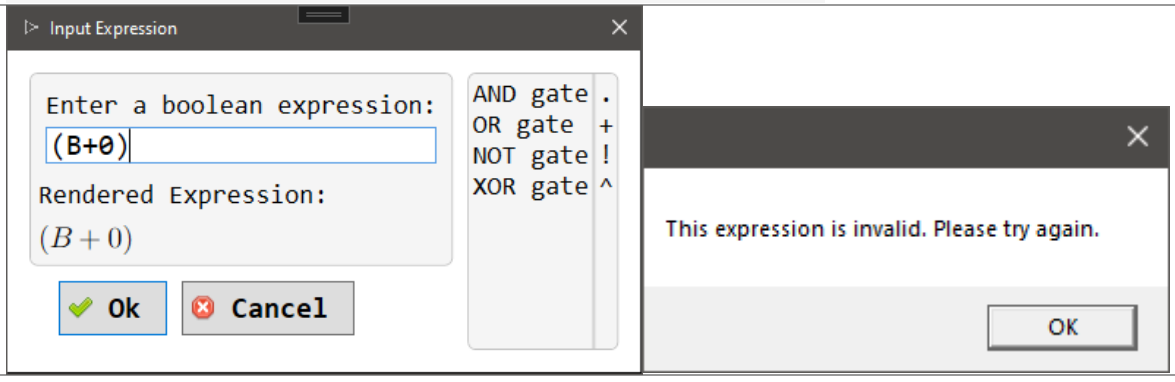


0.1.29

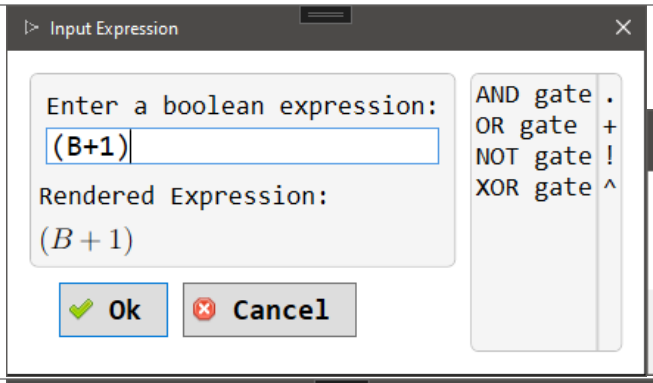


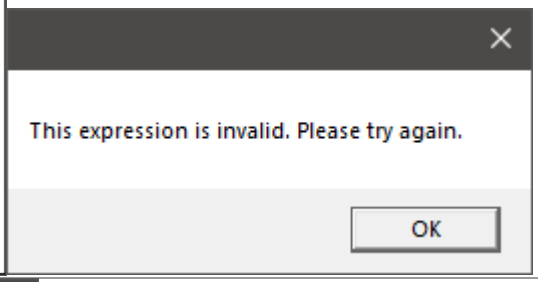


0.1.30

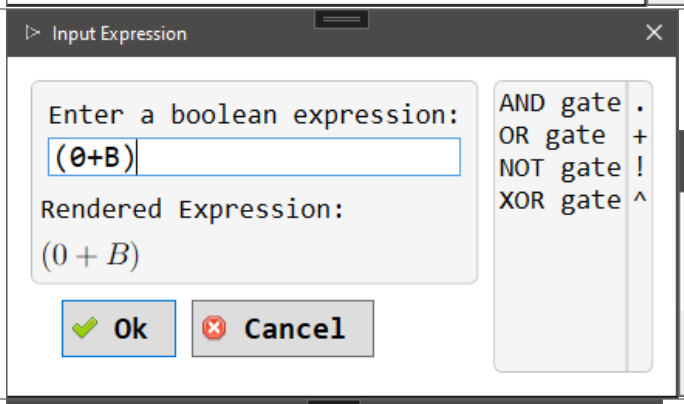


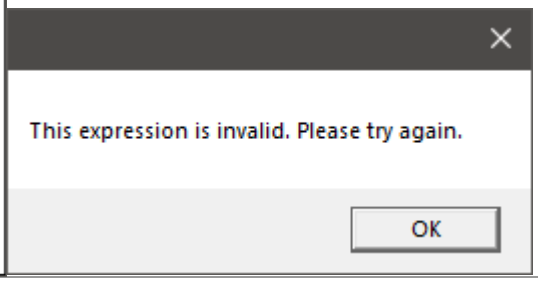
0.1.31



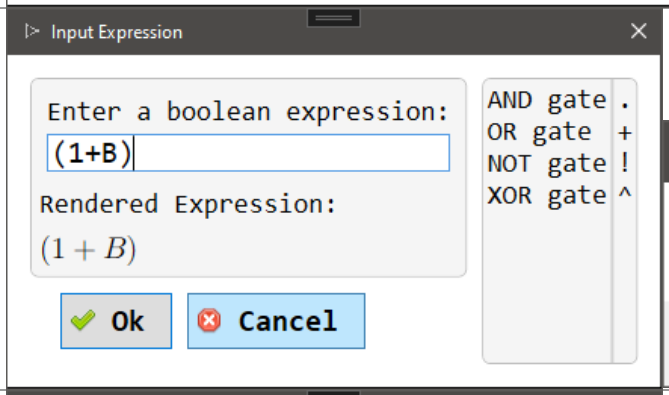


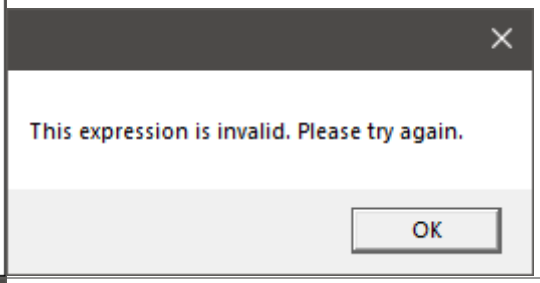
0.1.32



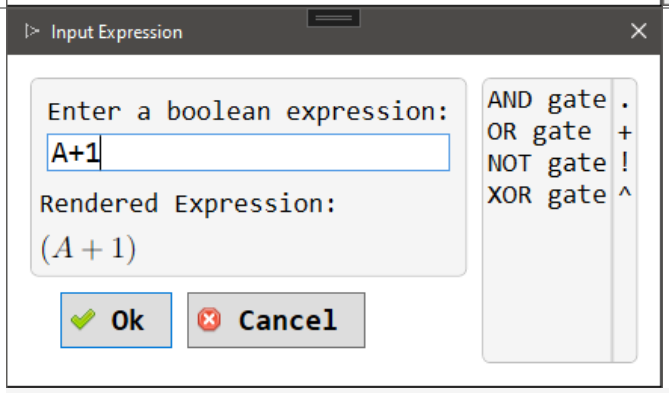


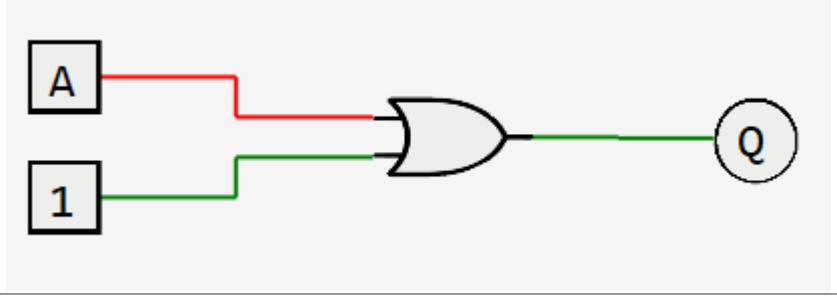
0.1.33



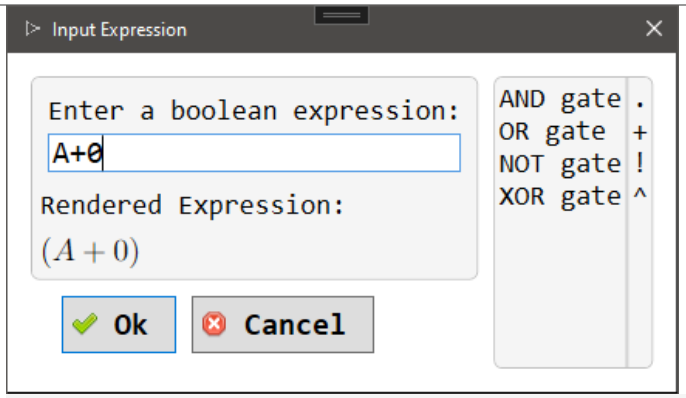


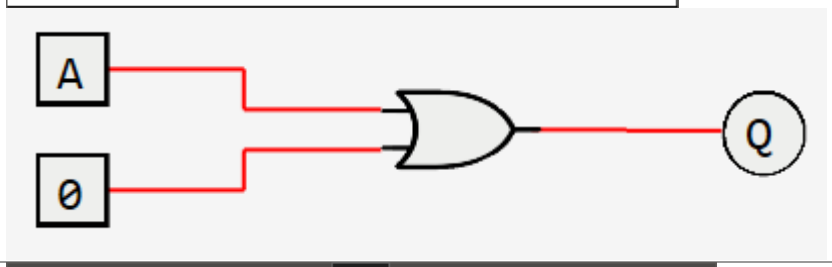
0.1.34



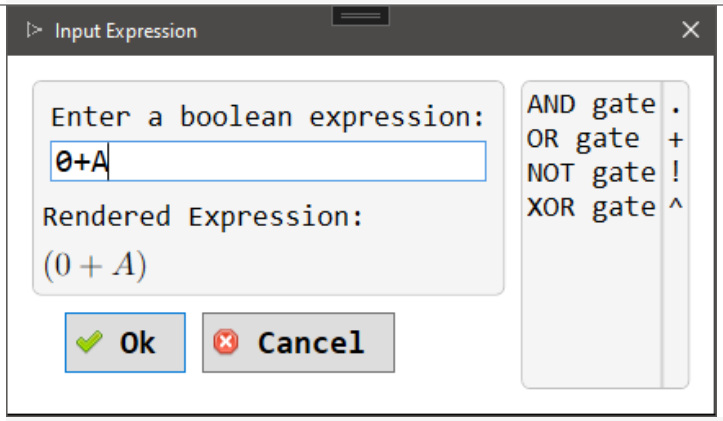


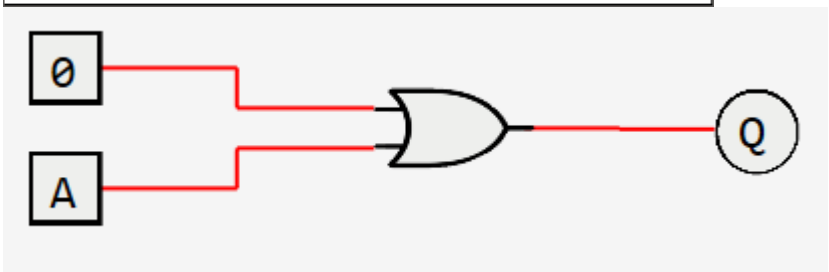
0.1.35



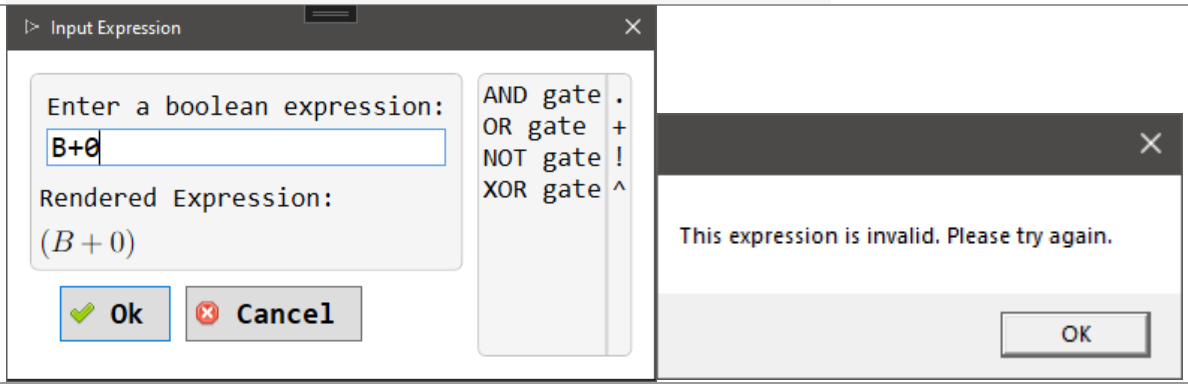


0.1.36

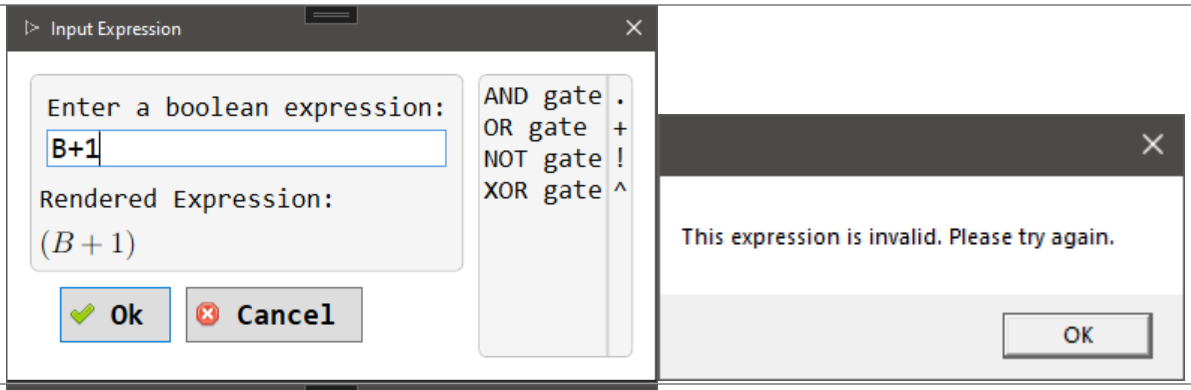




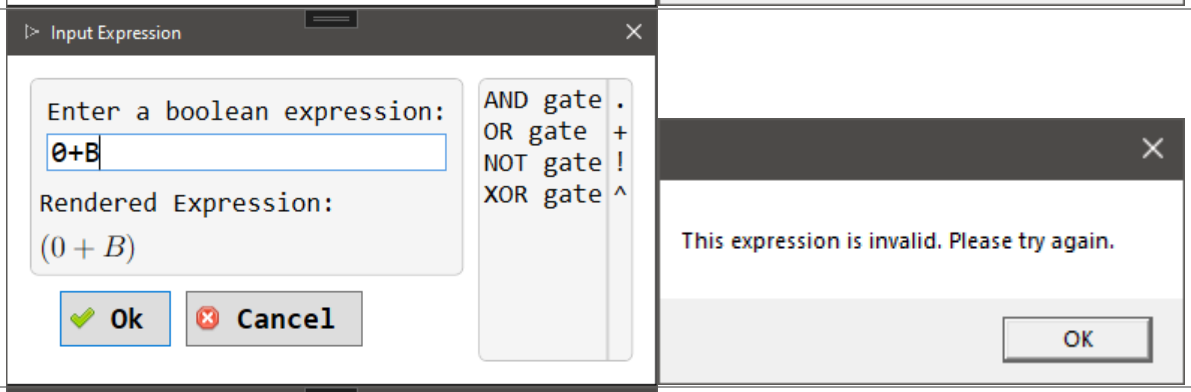
0.1.37



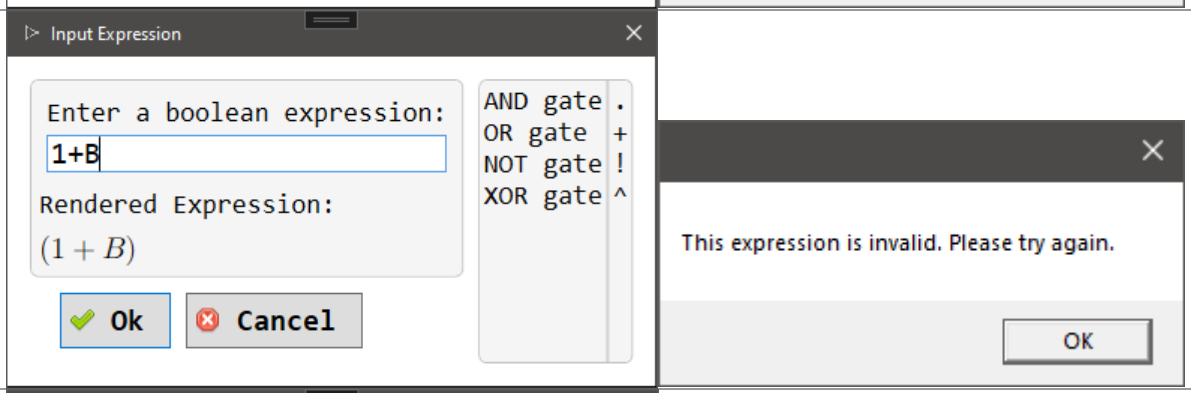
0.1.38



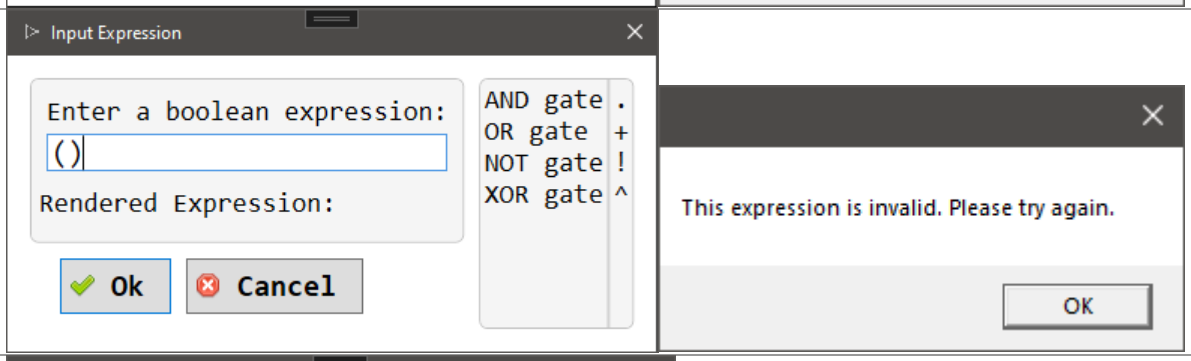
0.1.39



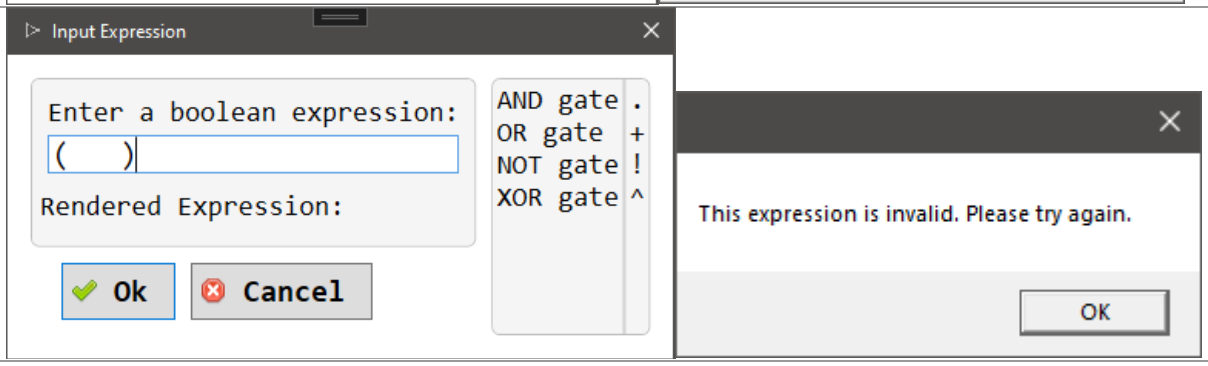
0.1.40



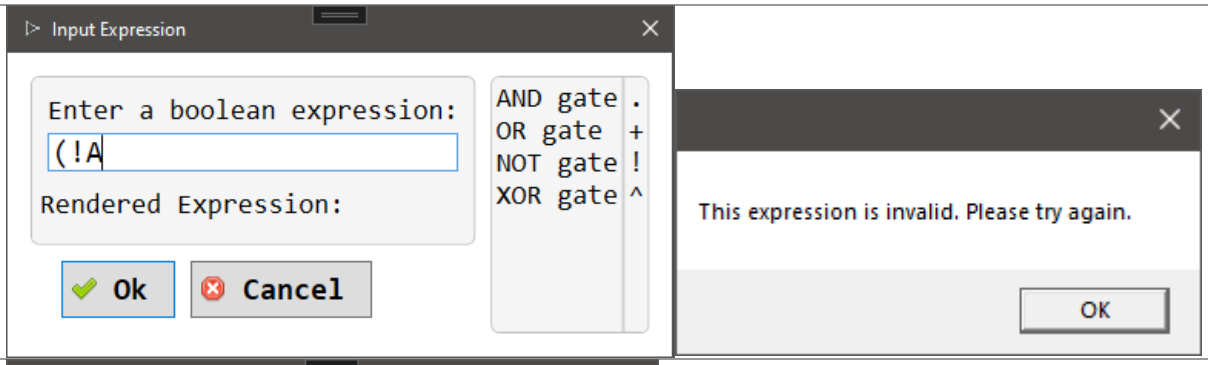
0.2.0



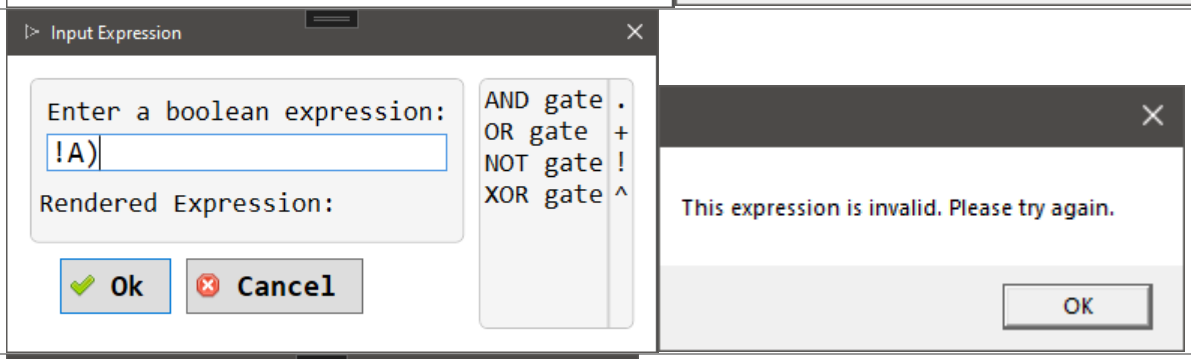
0.2.1



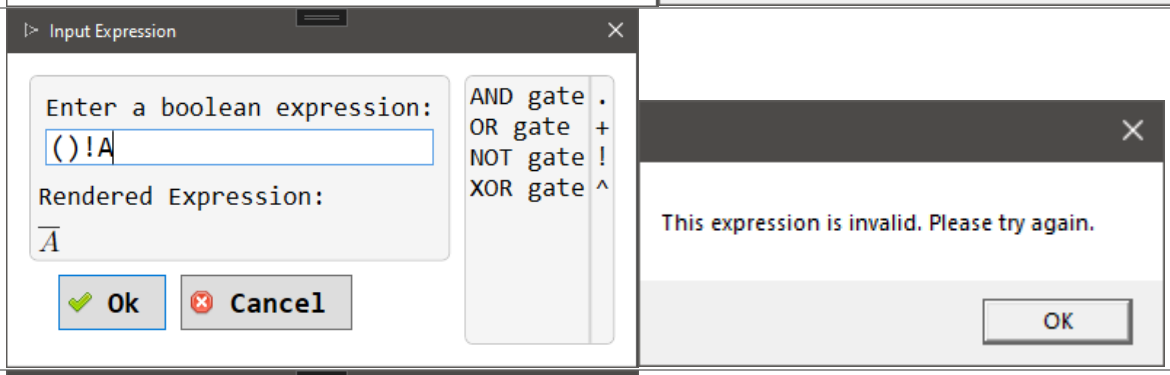
0.2.2



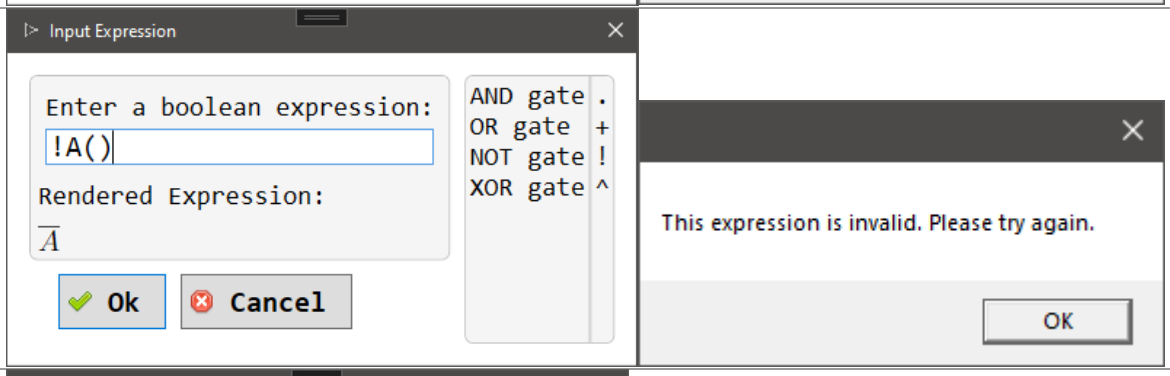
0.2.3



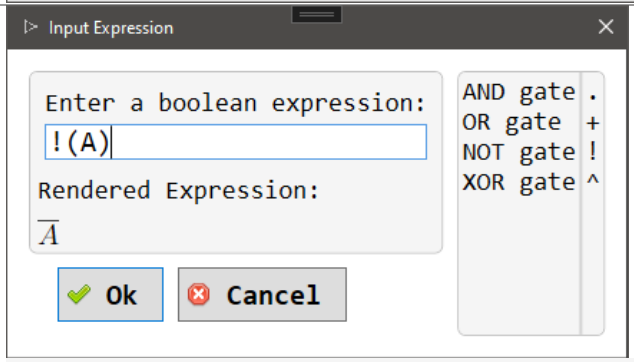
0.2.4

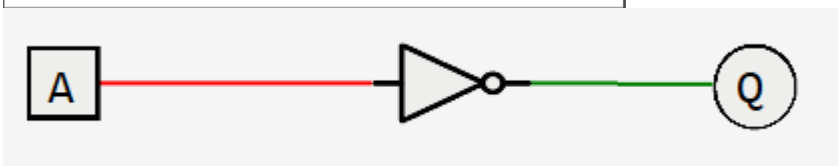


0.2.5

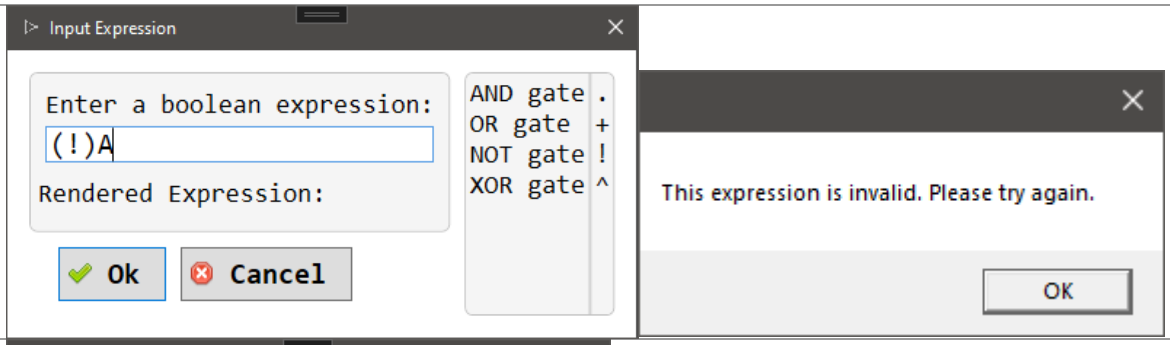


0.2.6

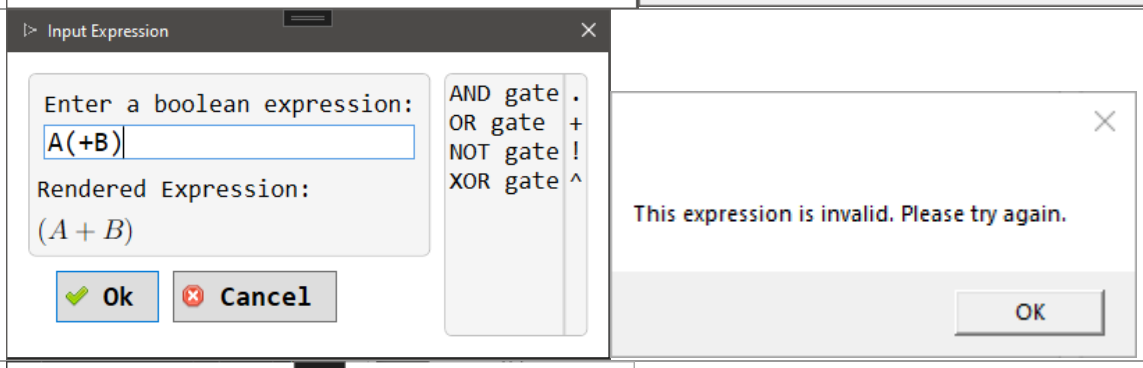




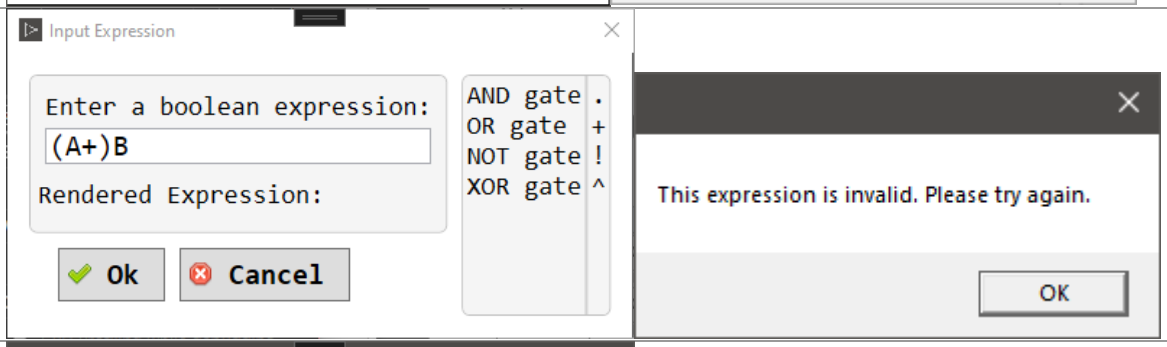
0.2.7



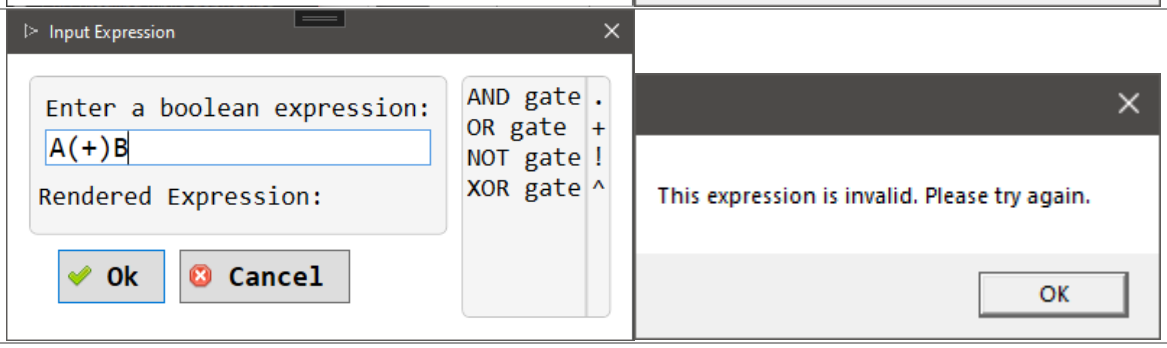
0.2.8



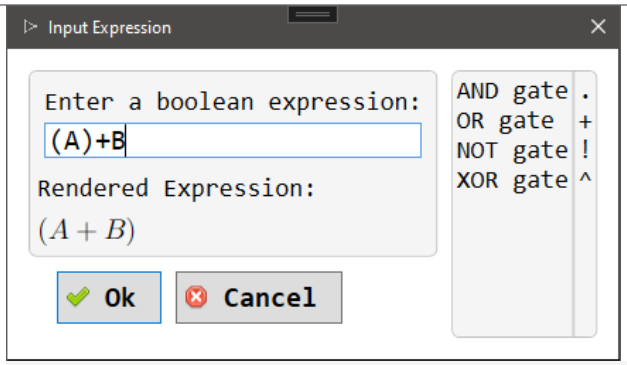
0.2.9

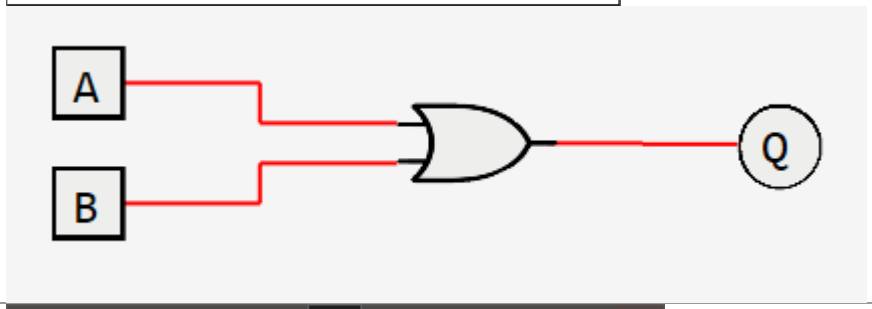


0.2.10

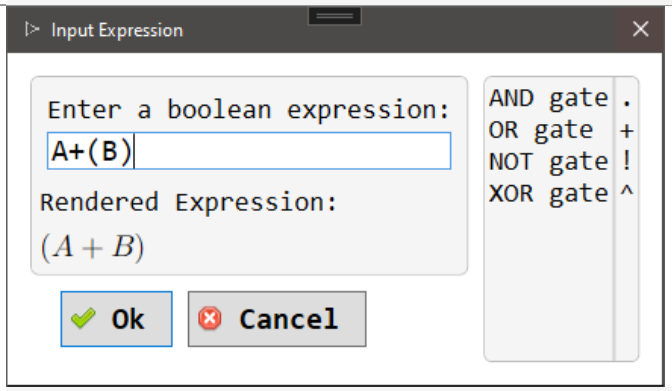


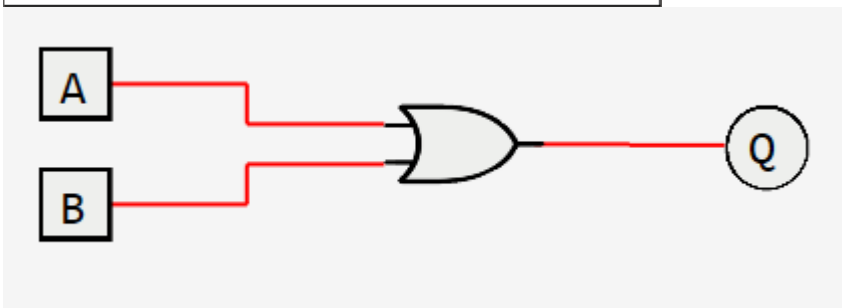
0.2.11


Input Expression
Enter a boolean expression:
(A)+B
Rendered Expression:
$(A + B)$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

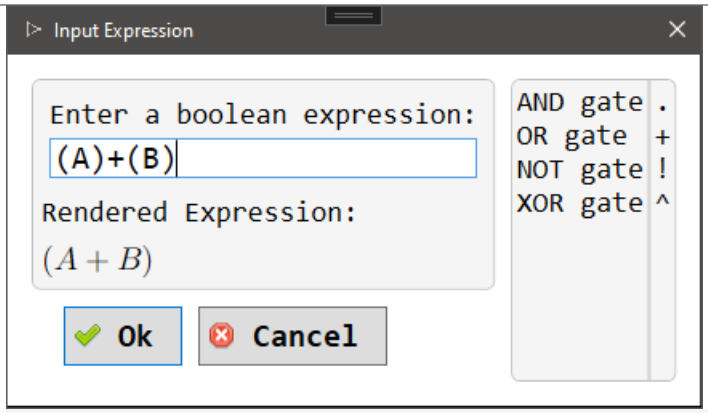

A
B
Q

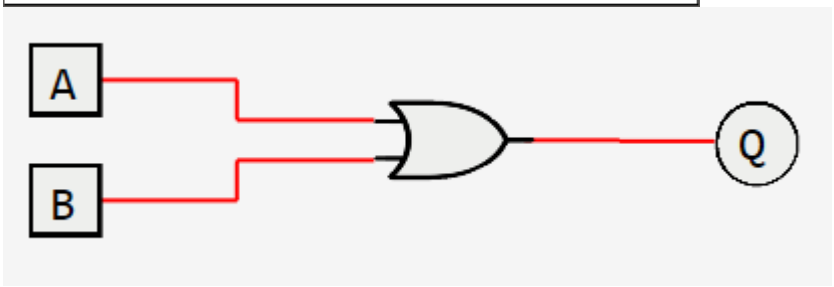
0.2.12


Input Expression
Enter a boolean expression:
A+(B)
Rendered Expression:
$(A + B)$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

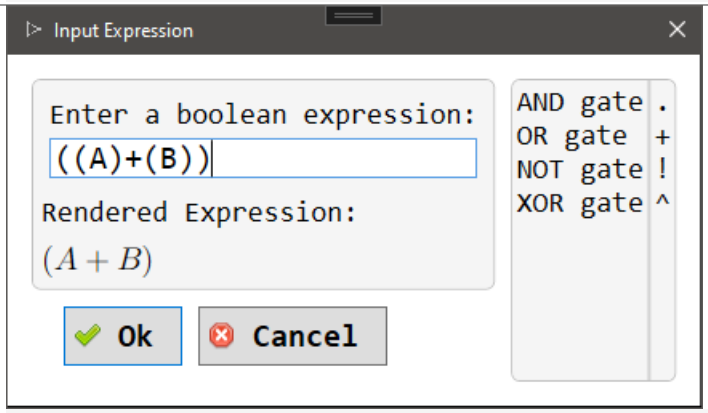

A
B
Q

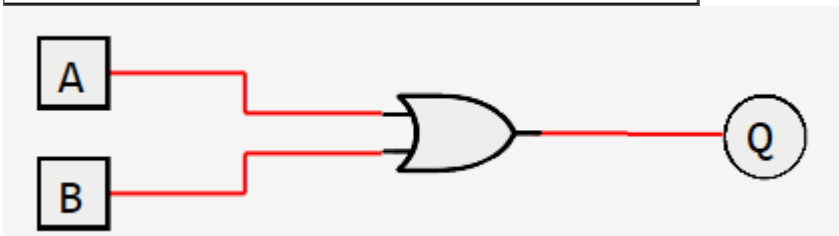
0.2.13


Input Expression
Enter a boolean expression:
(A)+(B)
Rendered Expression:
$(A + B)$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

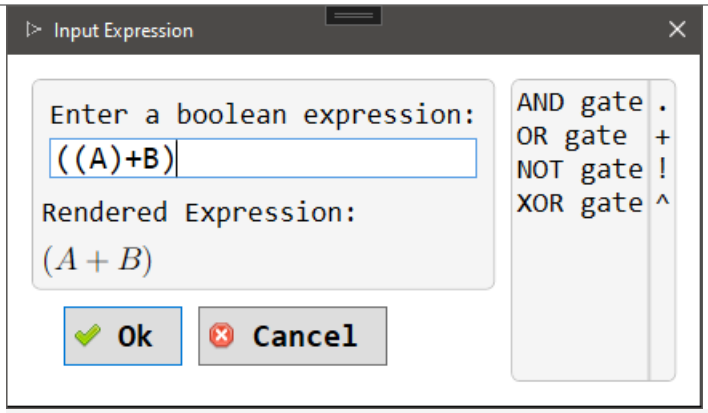

A
B
Q

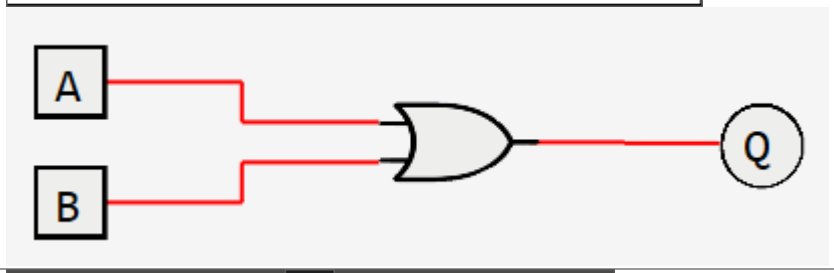
0.2.14


Input Expression
Enter a boolean expression:
((A)+(B))
Rendered Expression:
$(A + B)$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

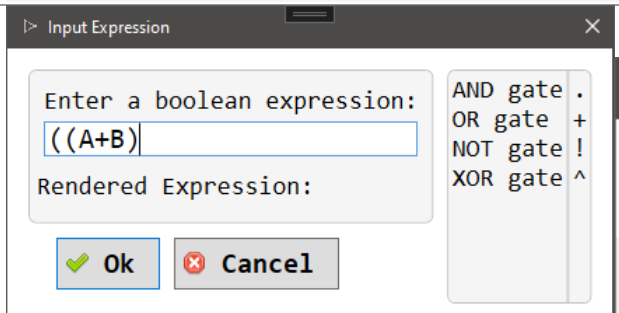

A
B
Q

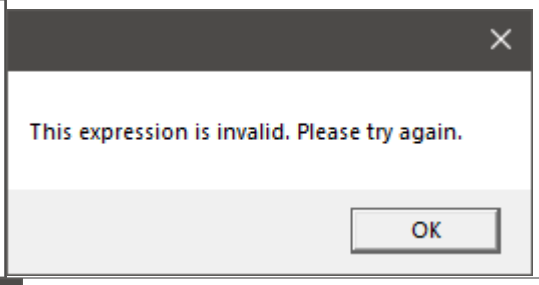
0.2.15



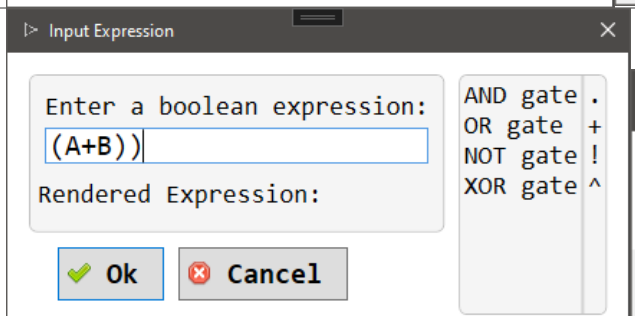


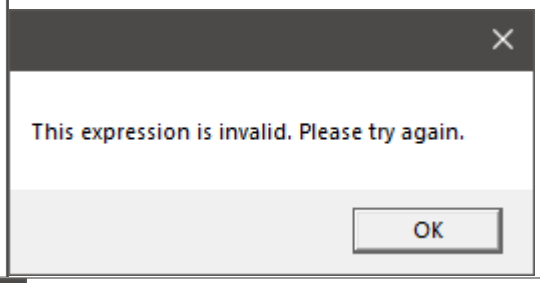
0.2.16



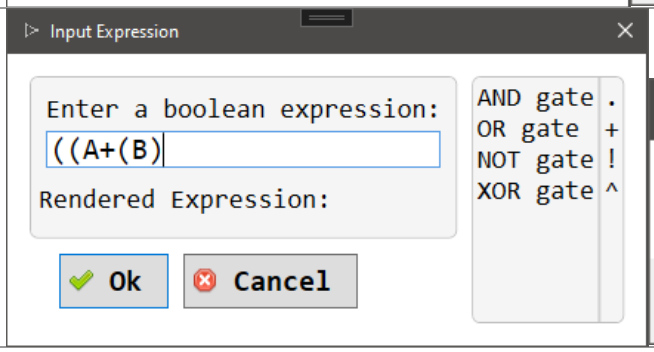


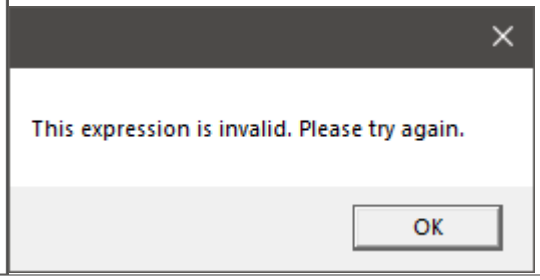
0.2.17



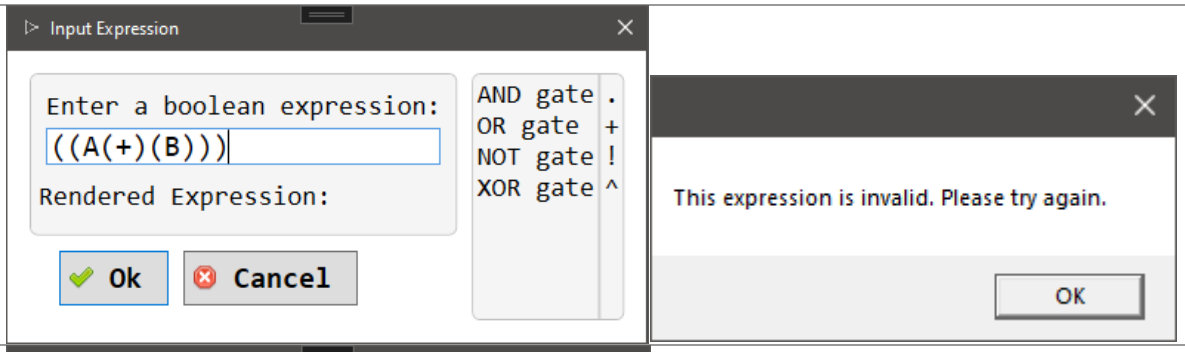


0.2.18

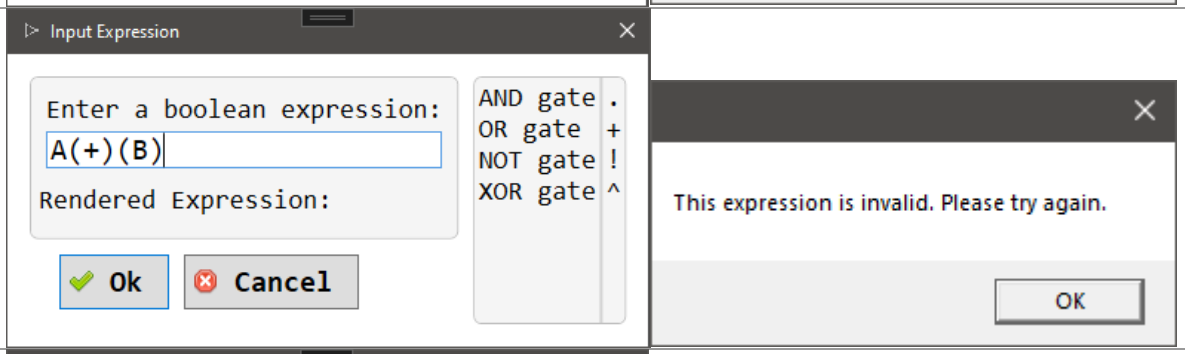




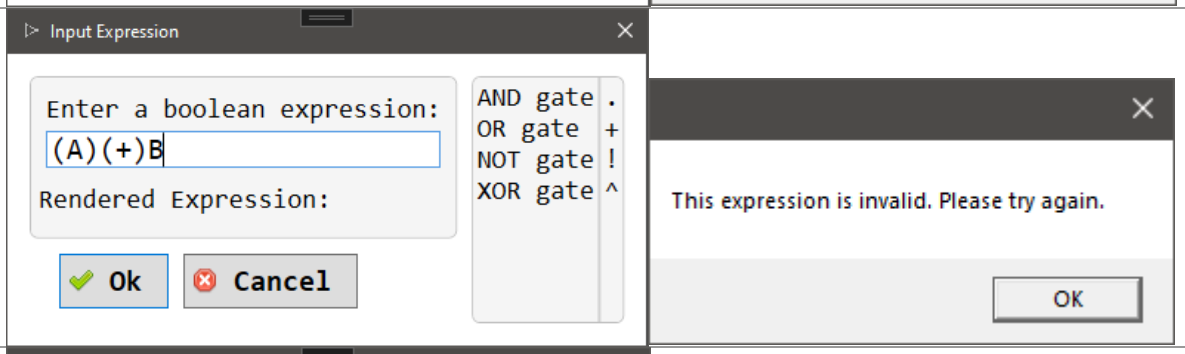
0.2.19



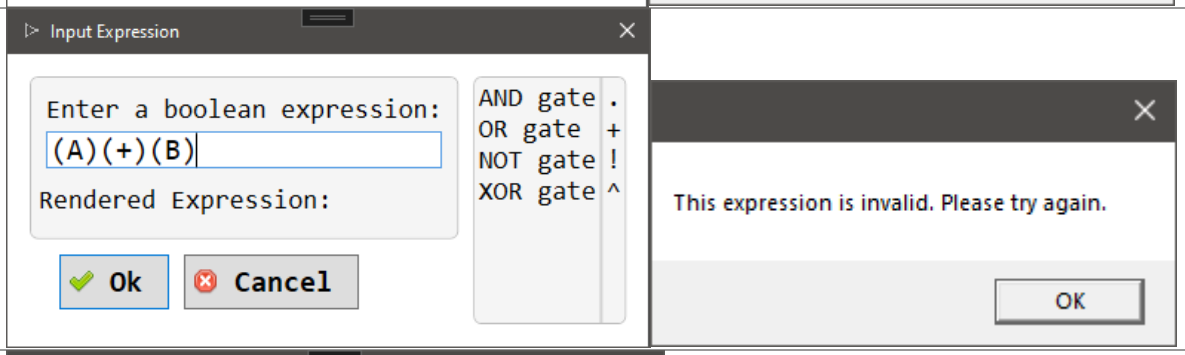
0.2.20



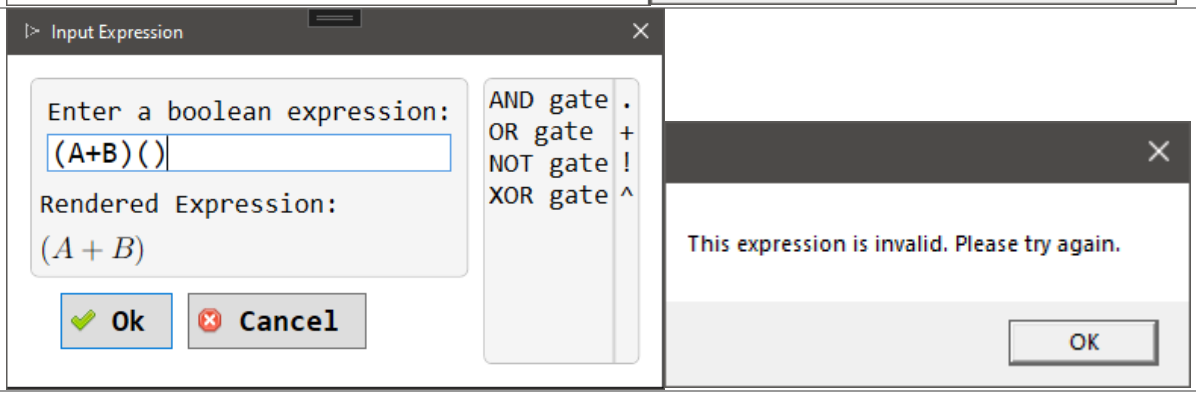
0.2.21



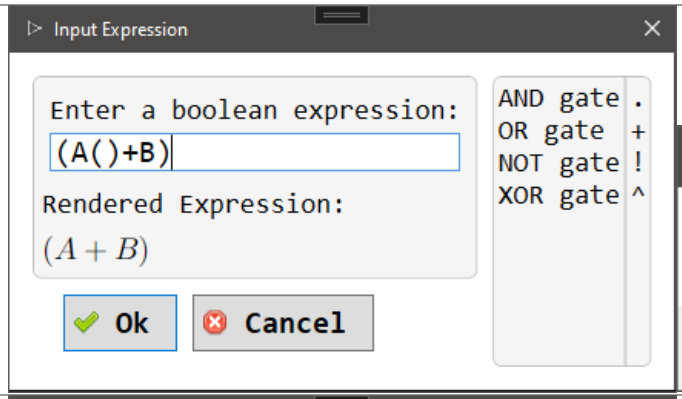
0.2.22

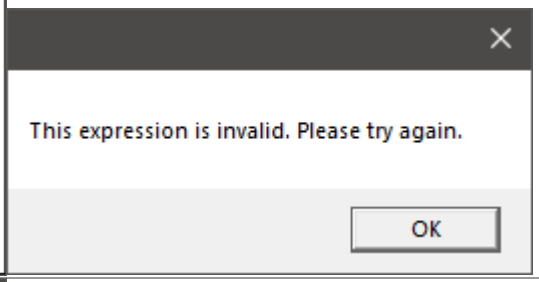


0.2.23

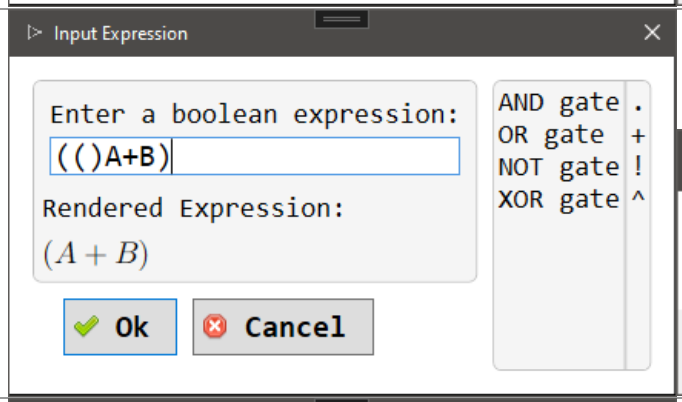


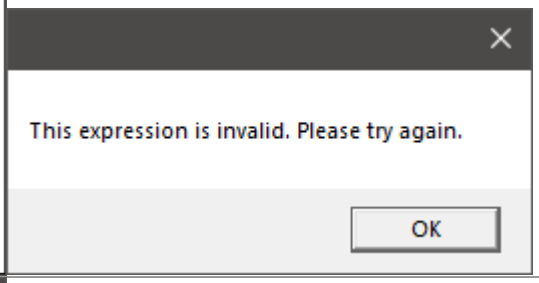
0.2.24



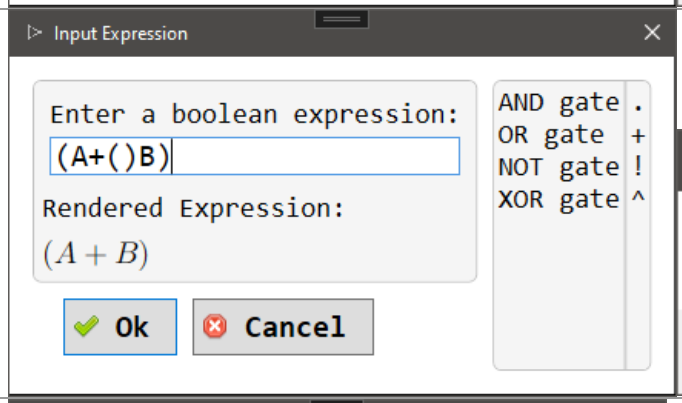


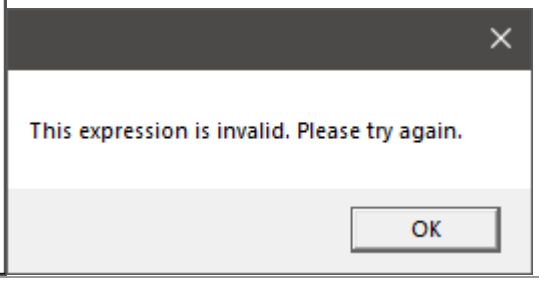
0.2.25



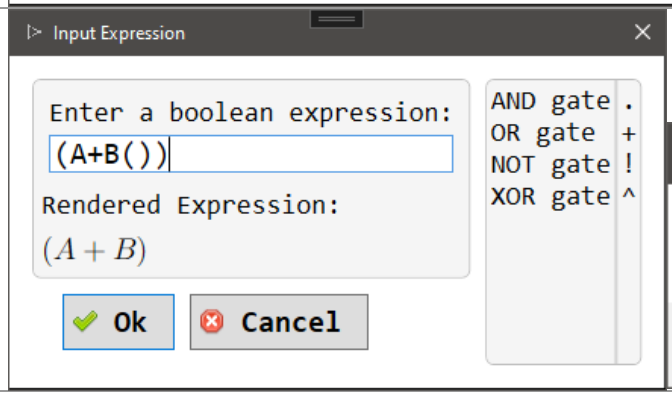


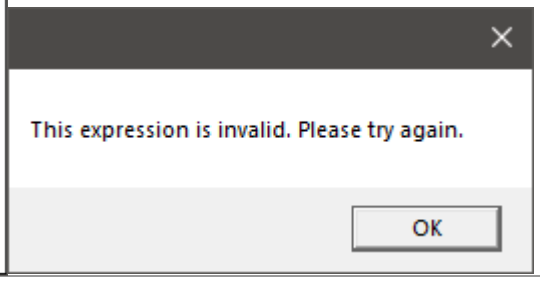
0.2.26



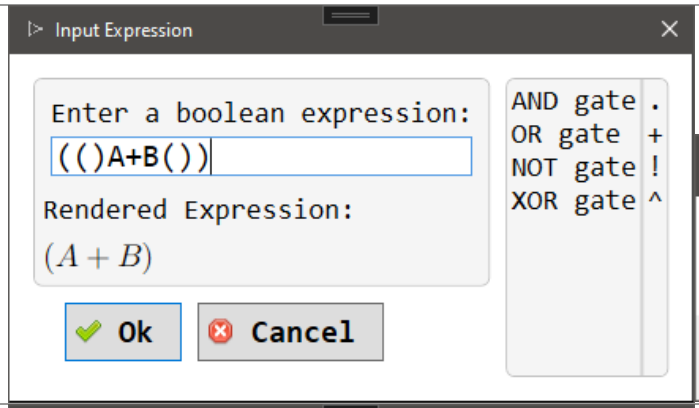


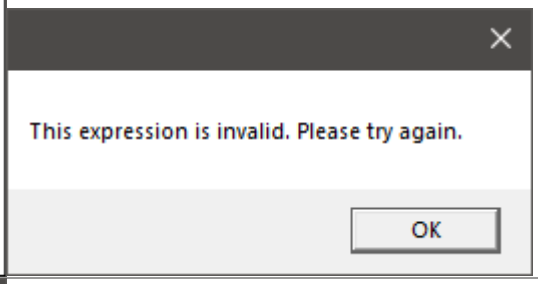
0.2.27



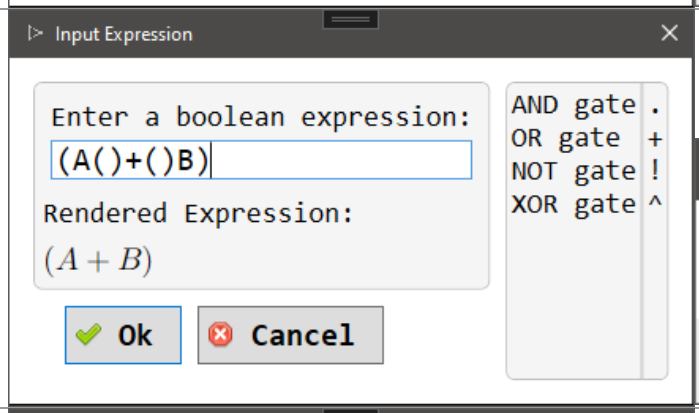


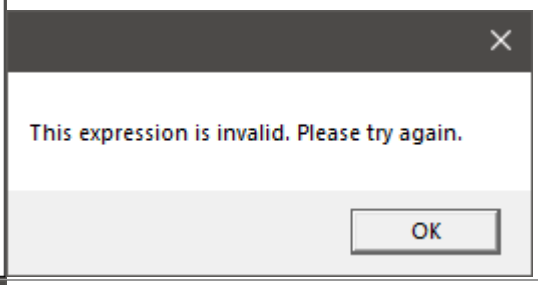
0.2.28



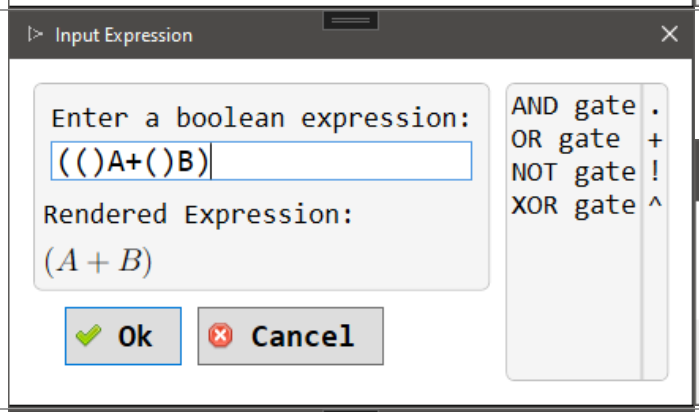


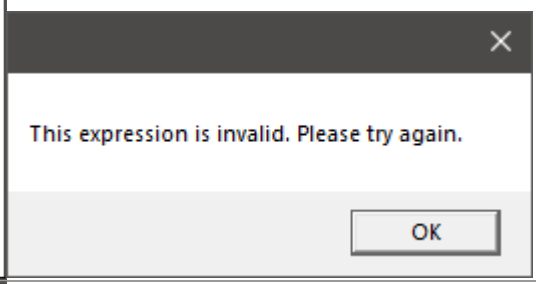
0.2.29



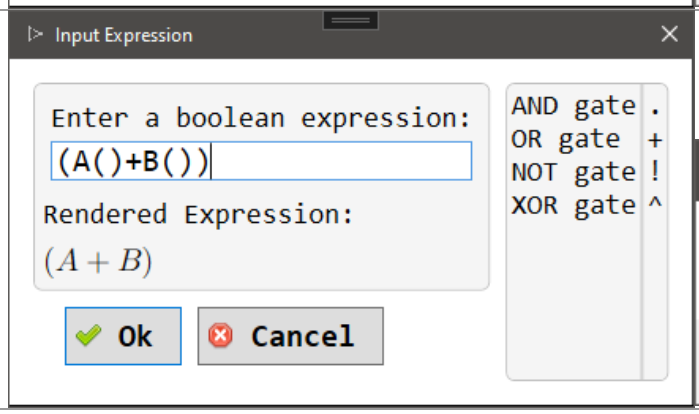


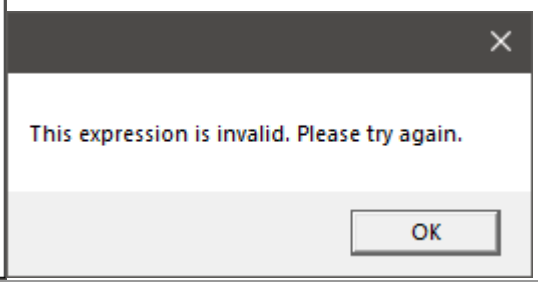
0.2.30



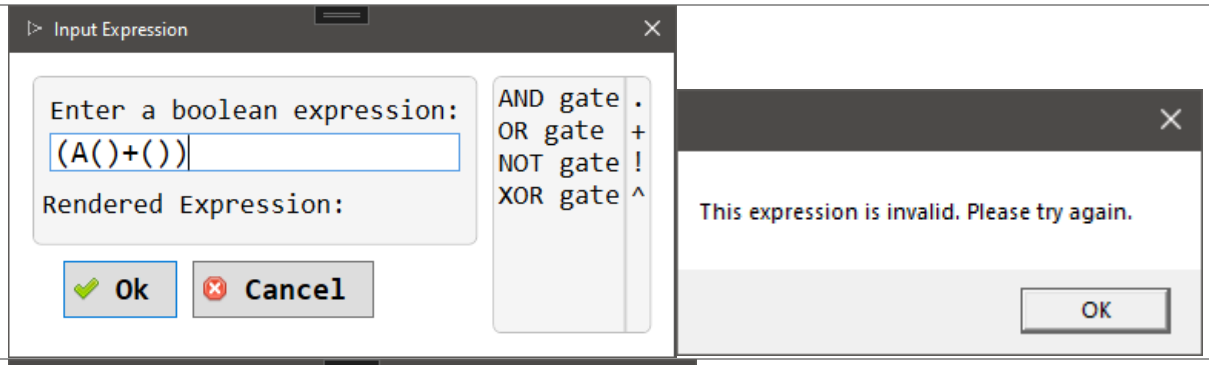


0.2.31

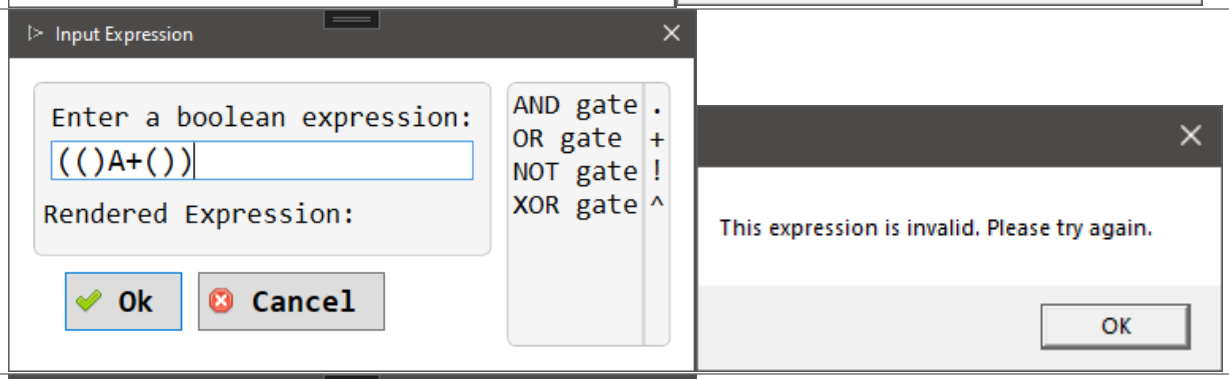




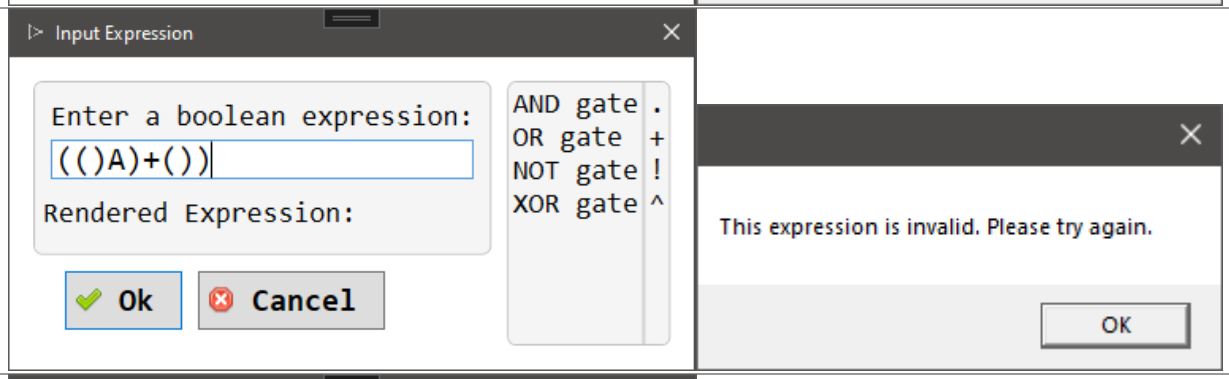
0.2.32



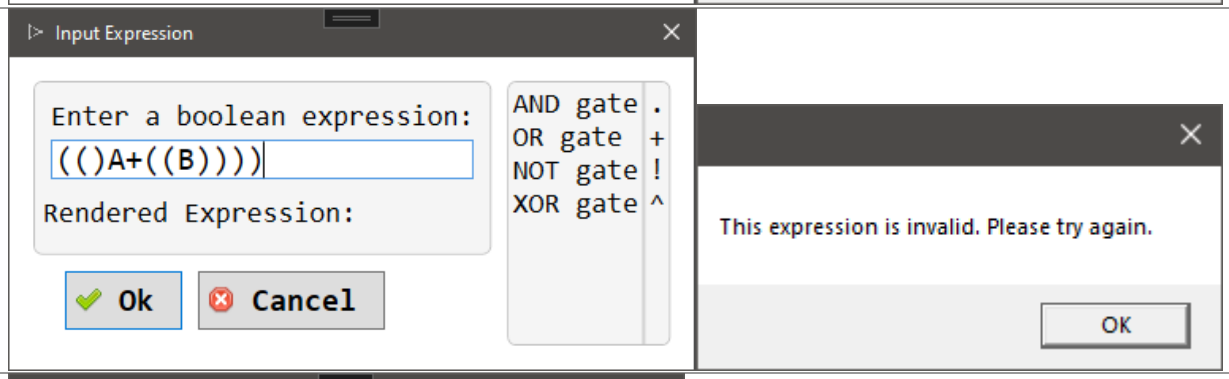
0.2.33



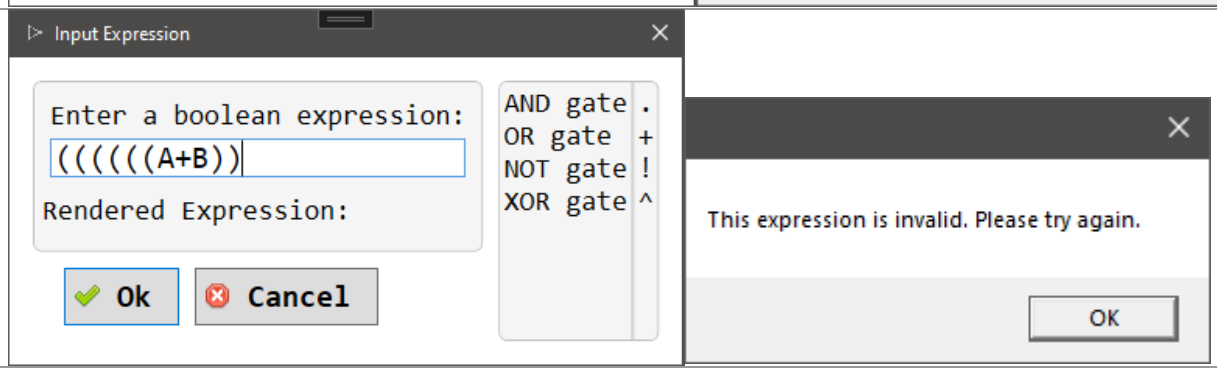
0.2.34



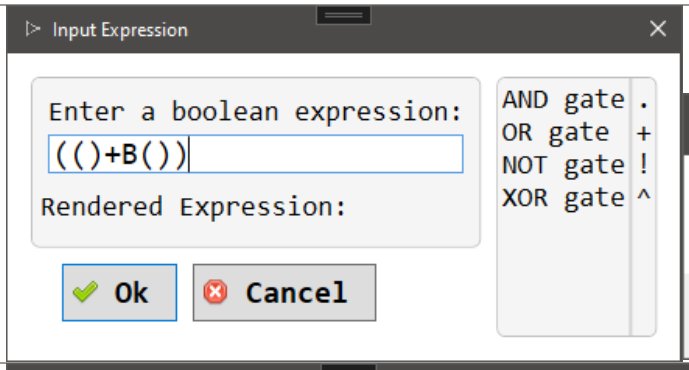
0.2.35

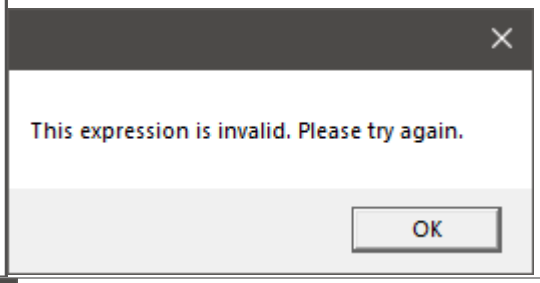


0.2.36

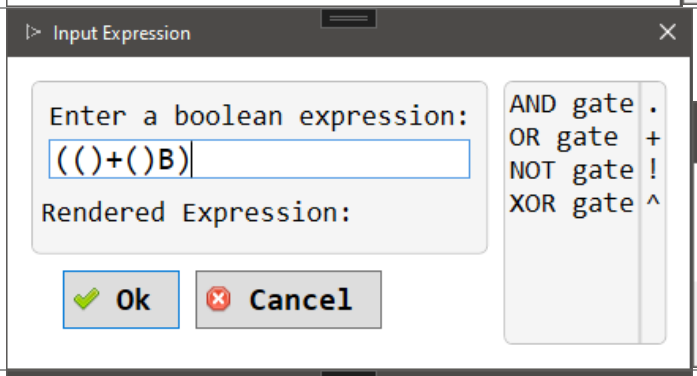


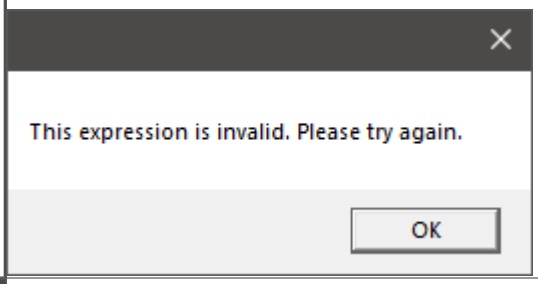
0.2.37



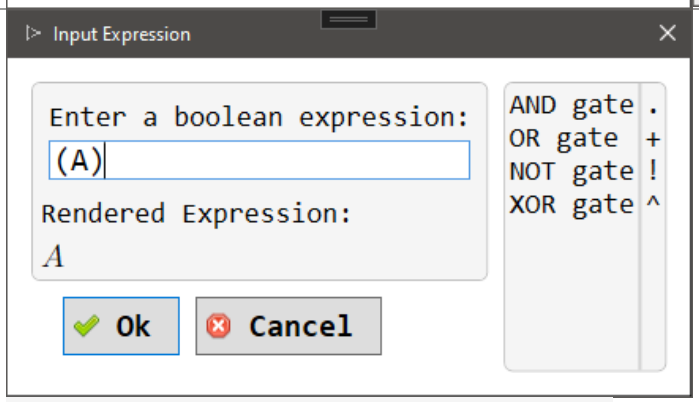


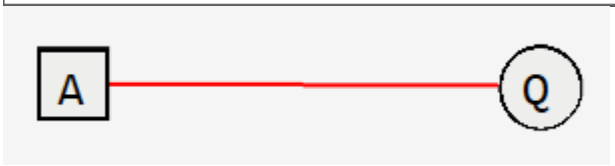
0.2.38



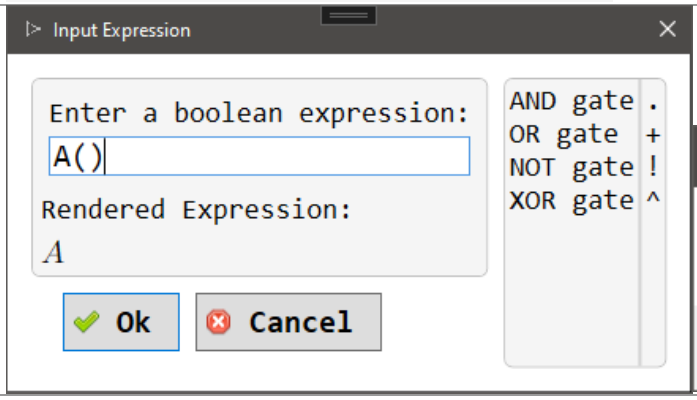


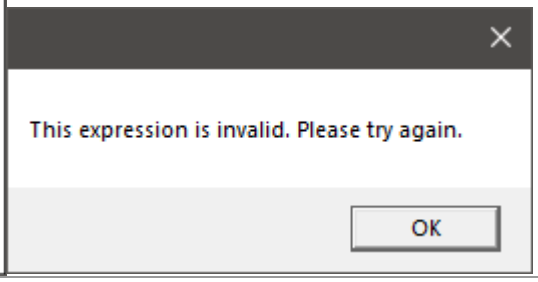
0.2.39



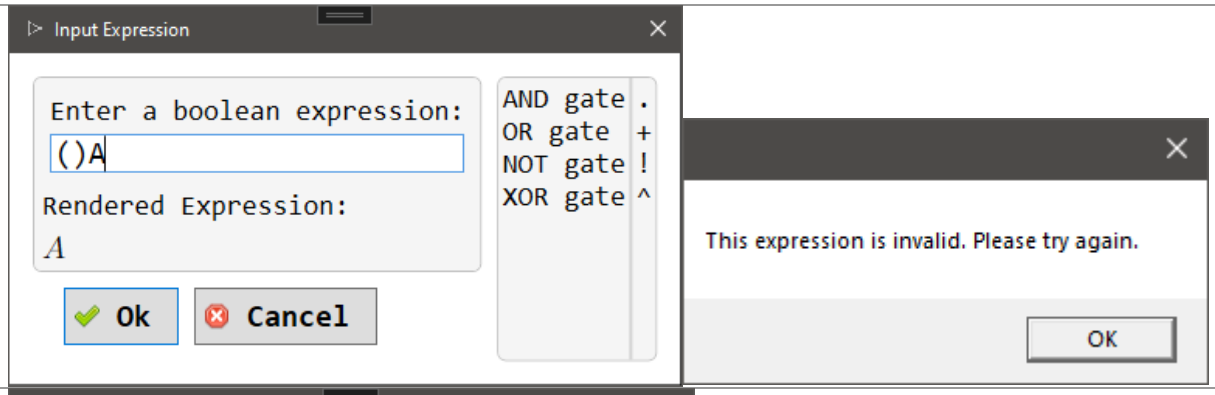


0.2.40

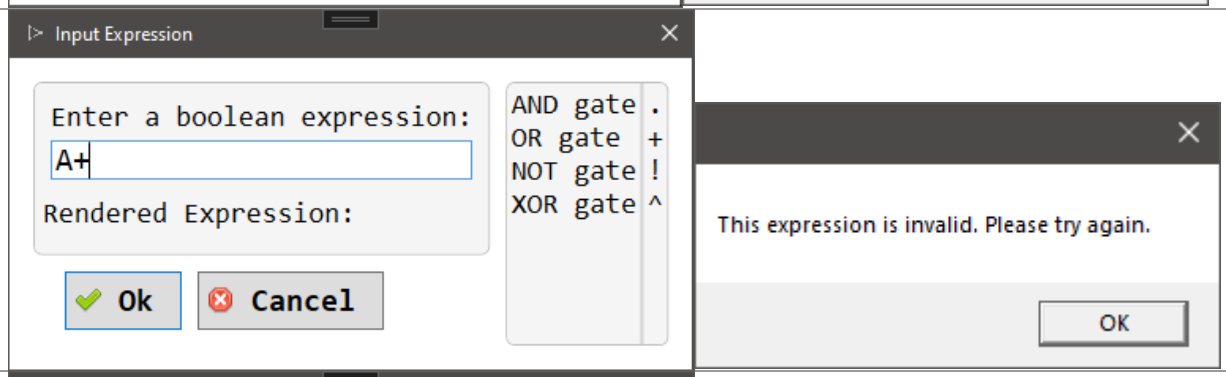




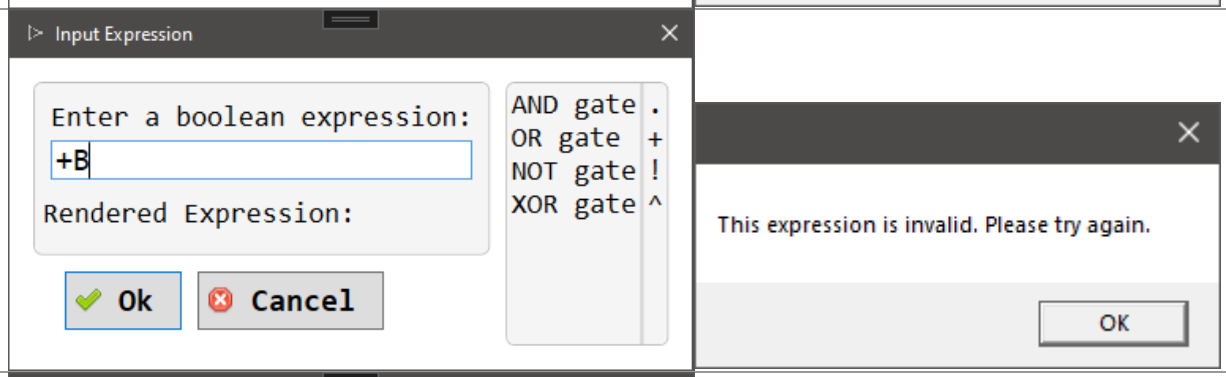
0.2.41



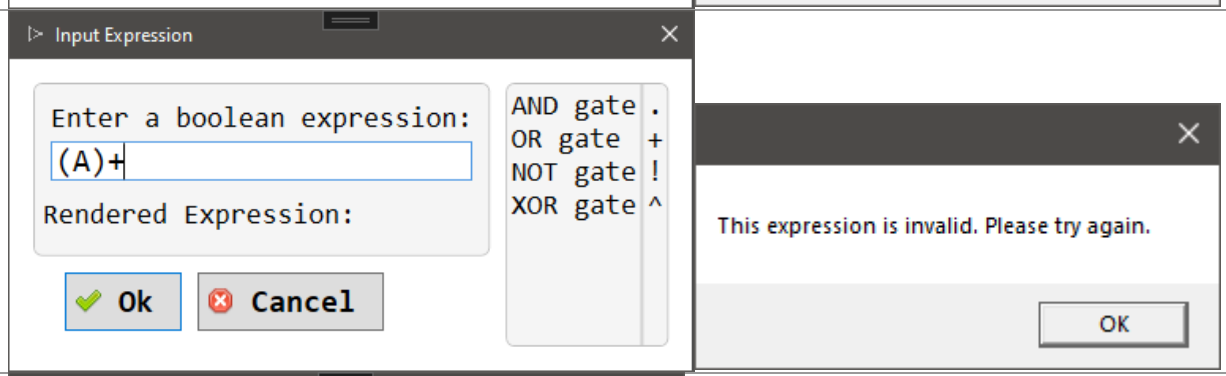
0.3.0



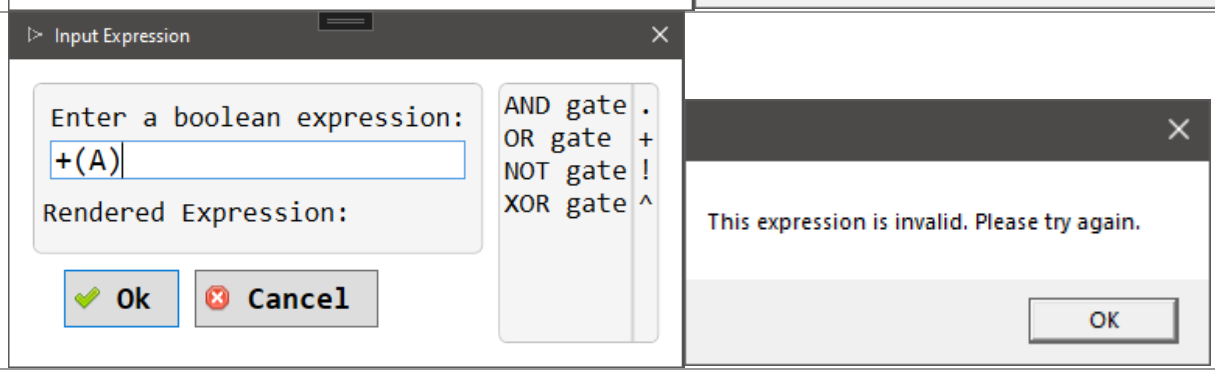
0.3.1



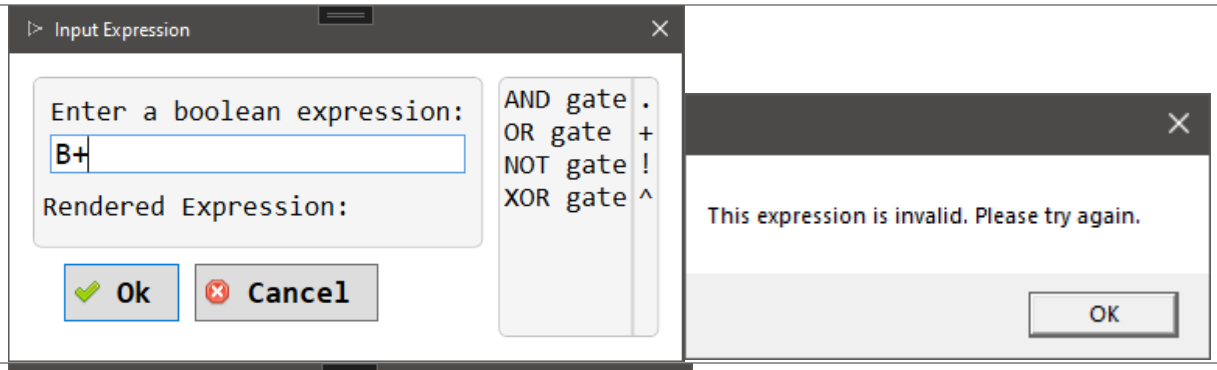
0.3.2



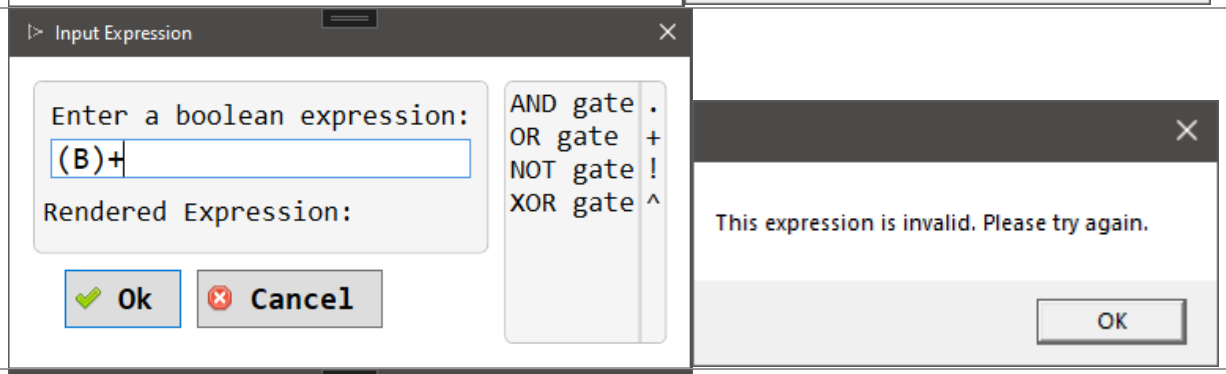
0.3.3



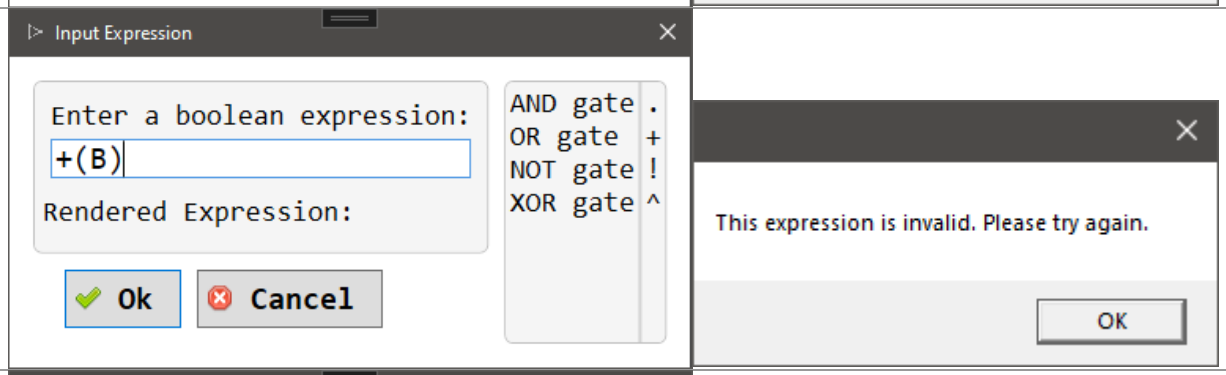
0.3.4



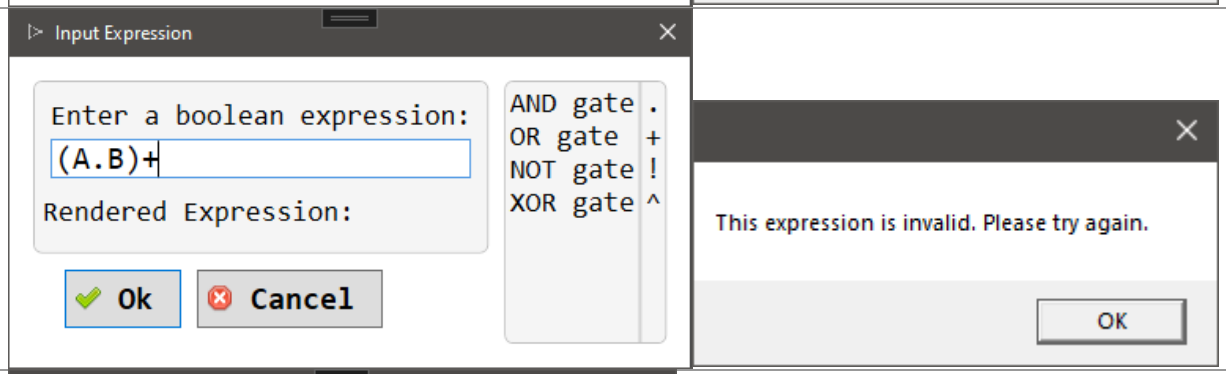
0.3.5



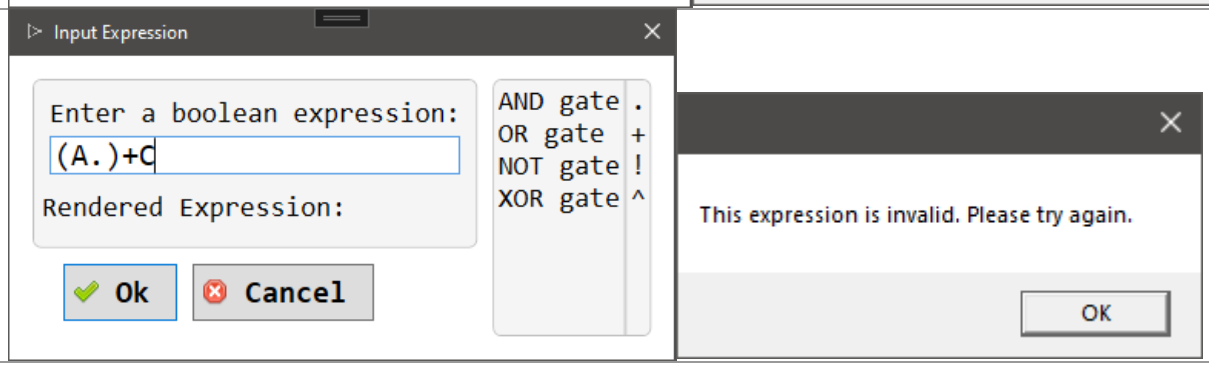
0.3.6



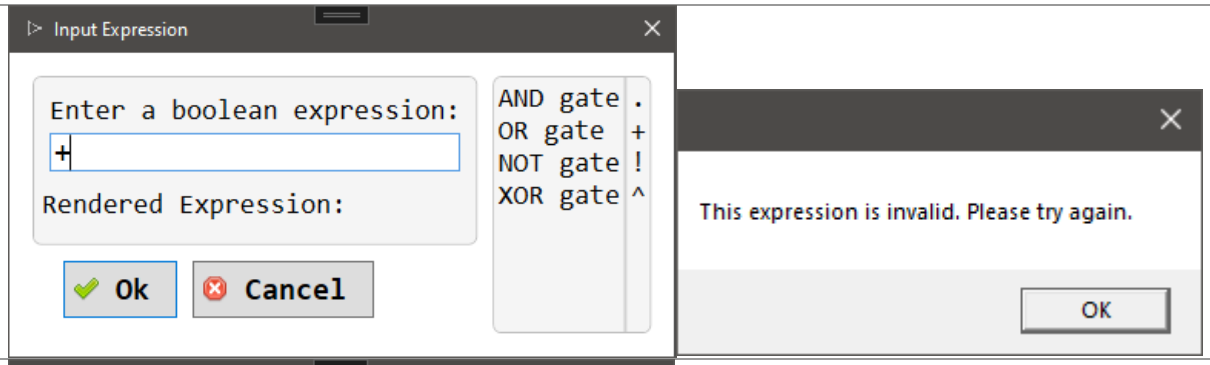
0.3.7



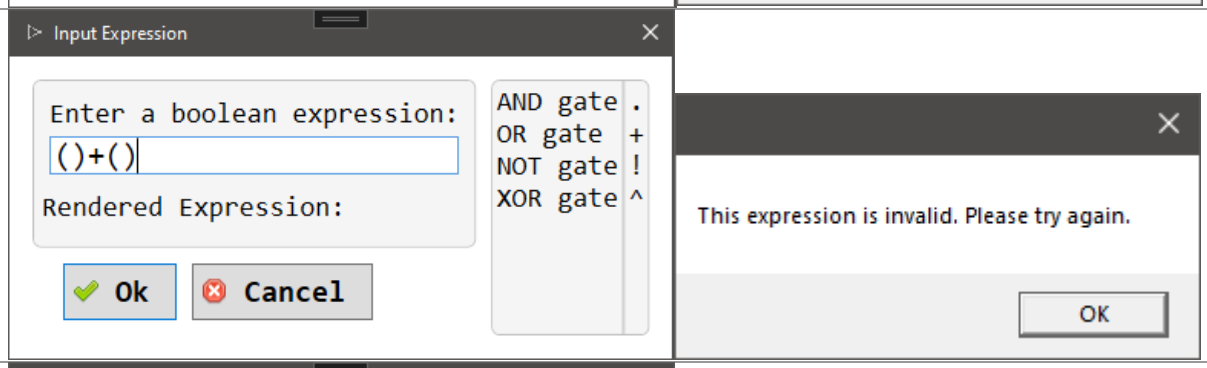
0.3.8



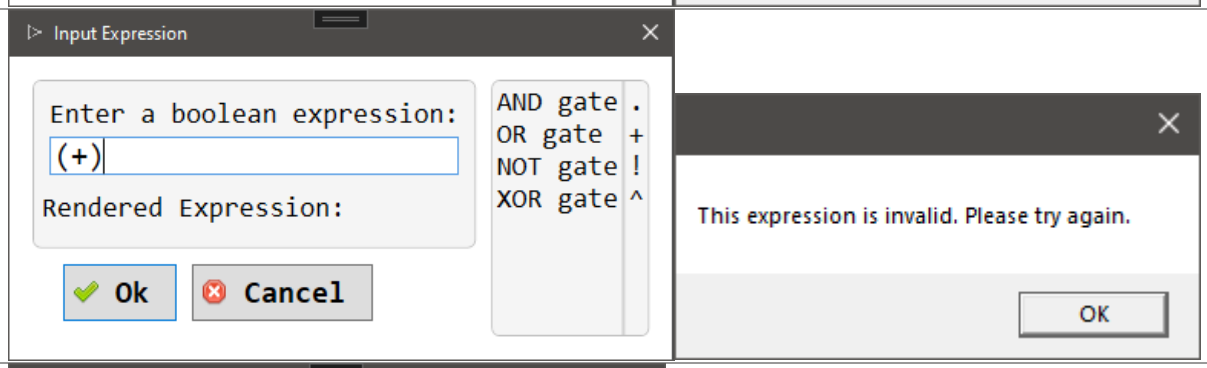
0.3.9



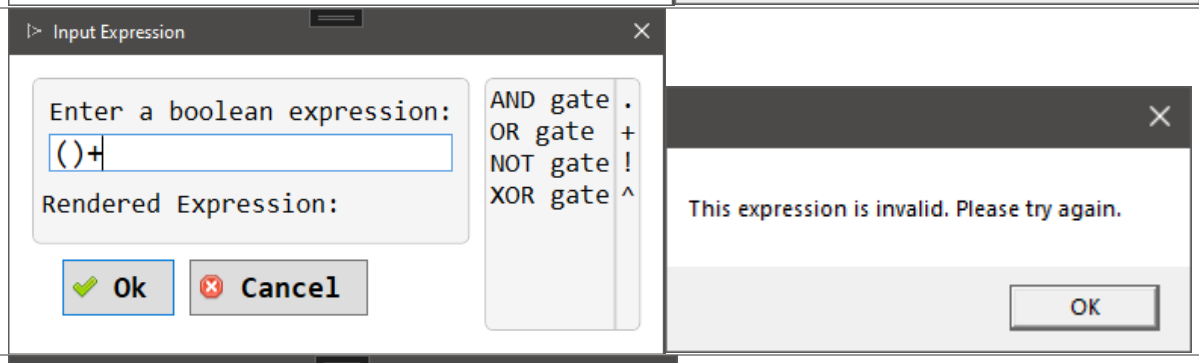
0.3.10



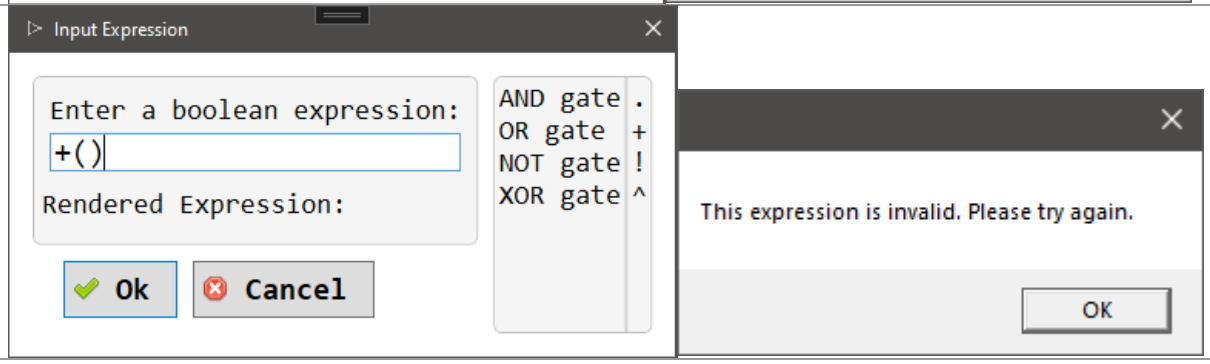
0.3.11



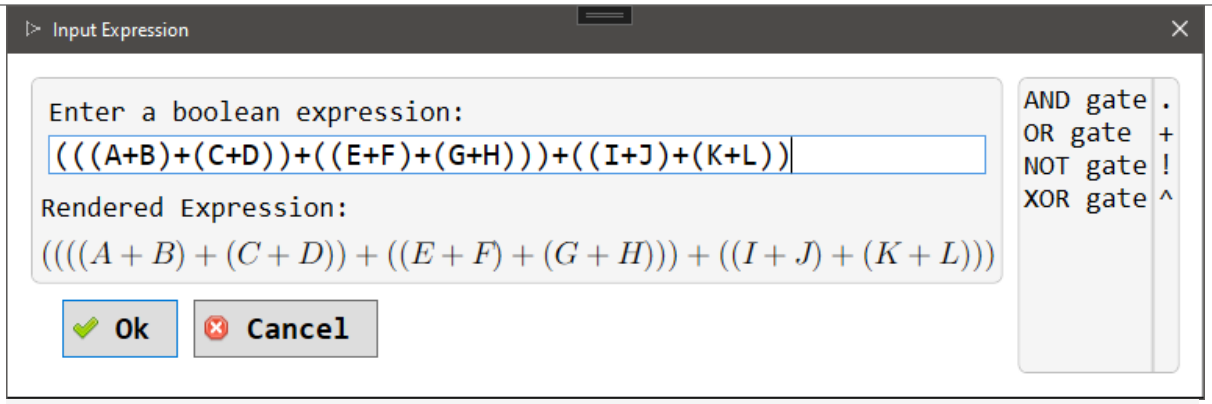
0.3.12

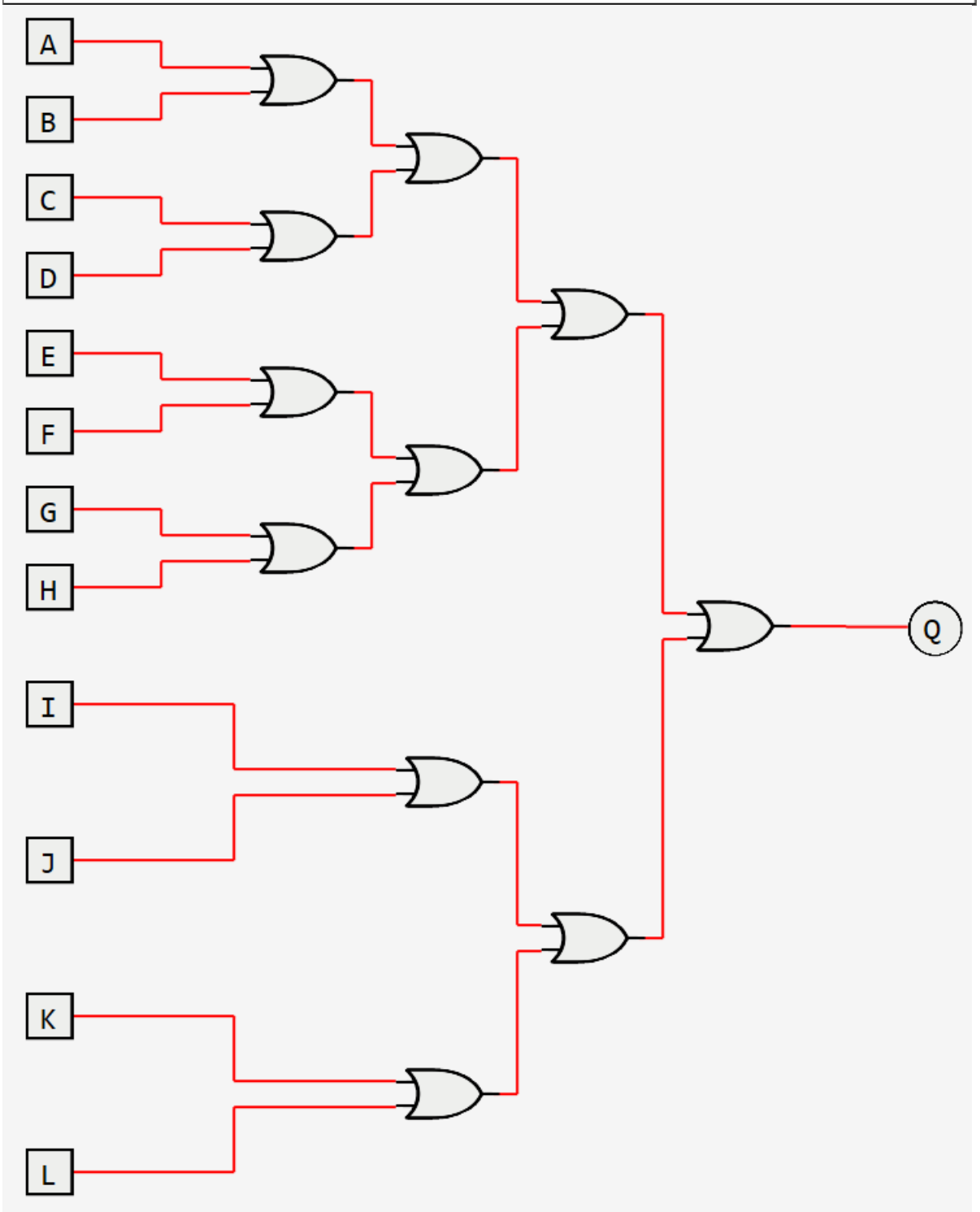


0.3.13

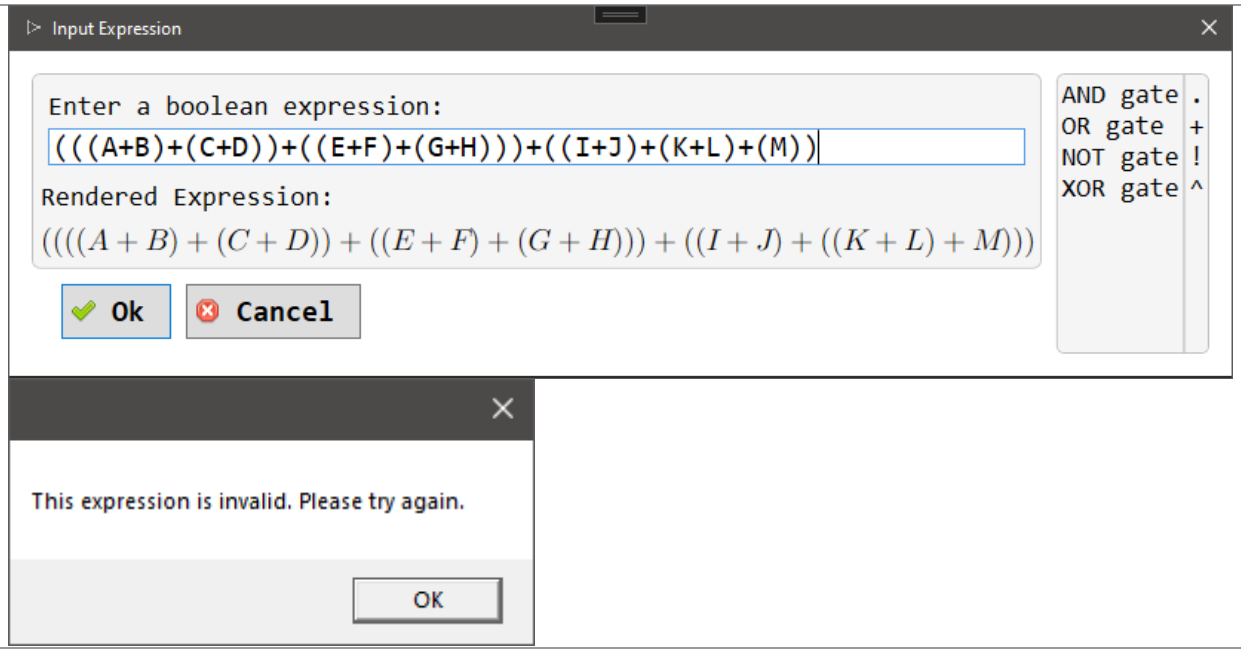


0.4.0

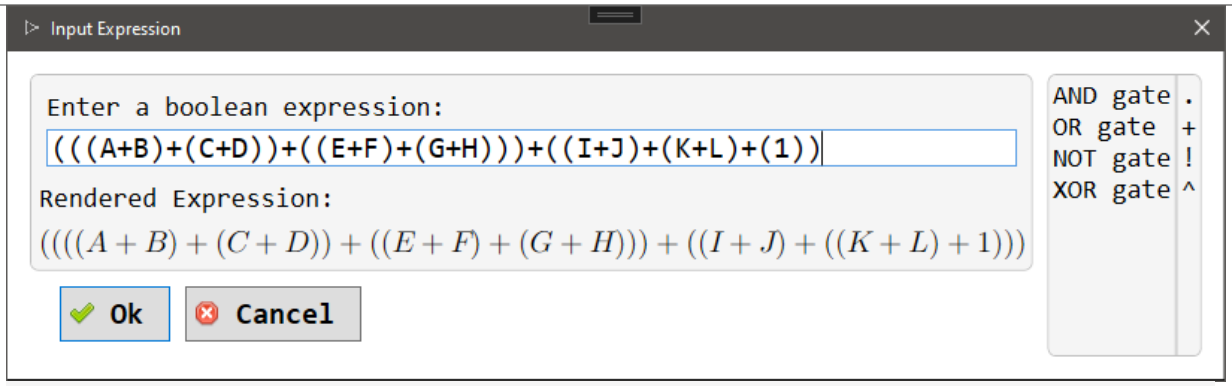

Input Expression
Enter a boolean expression:
(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L))
Rendered Expression:
$((((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L)))$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

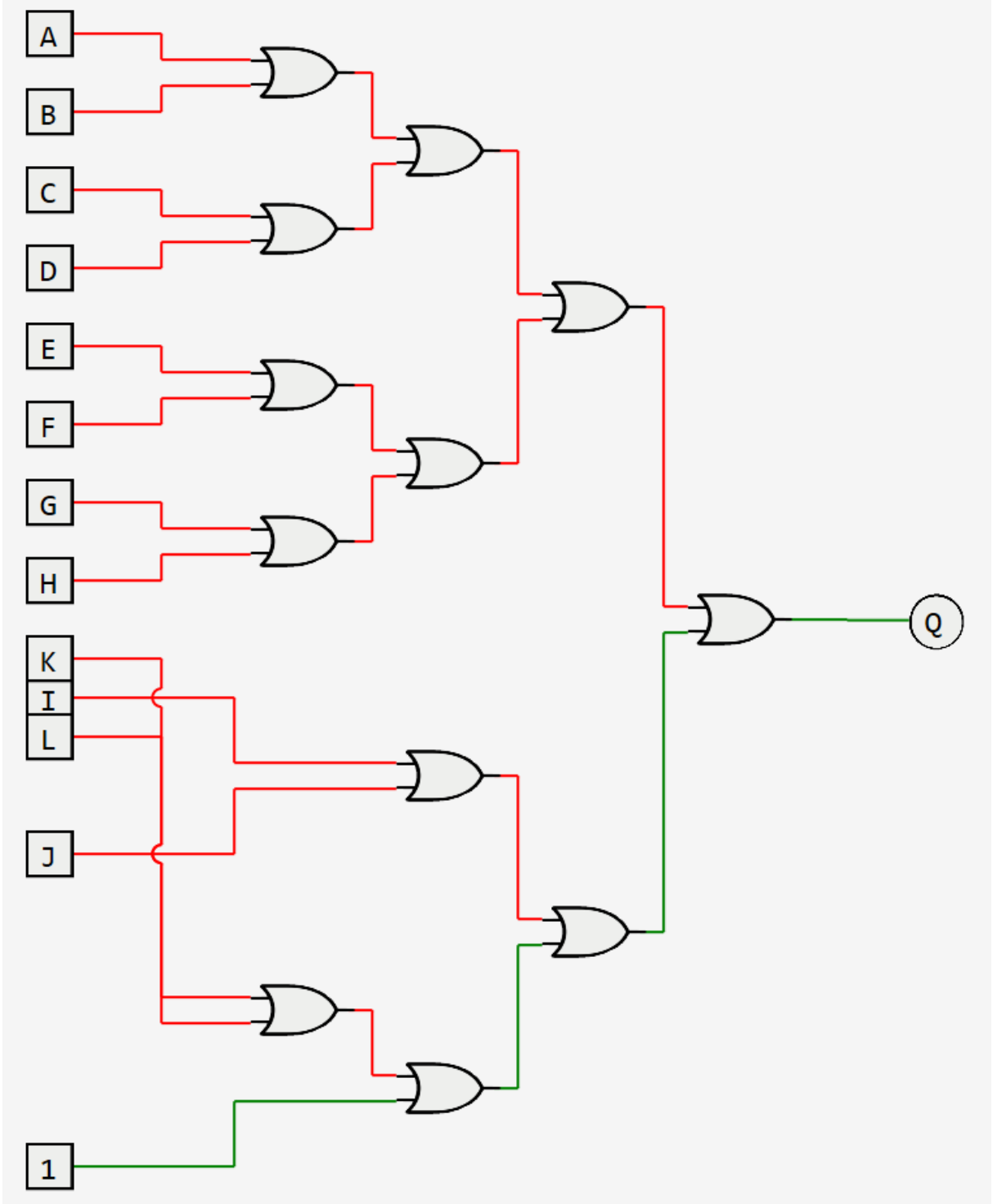

A
B
C
D
E
F
G
H
I
J
K
L
Q

0.4.1

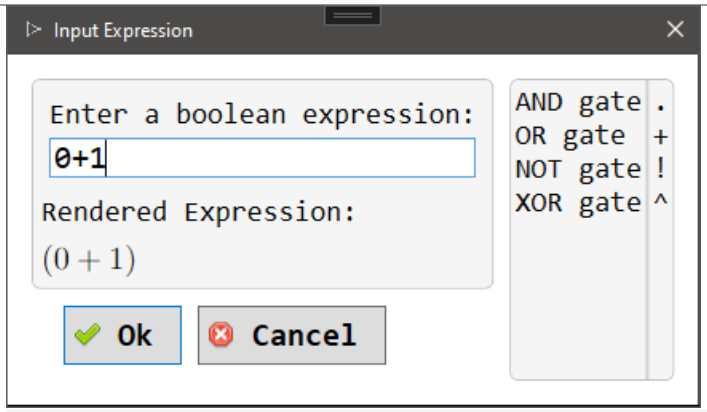

Input Expression
Enter a boolean expression:
(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L)+(M))
Rendered Expression:
$((((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+((K+L)+M)))$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^
This expression is invalid. Please try again.
OK

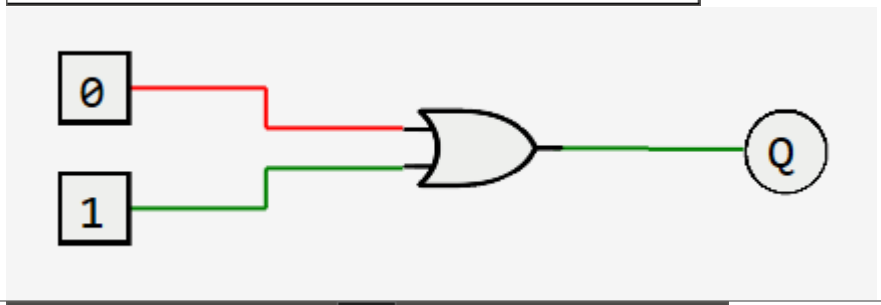
0.4.2


Input Expression
Enter a boolean expression:
(((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+(K+L)+(1))
Rendered Expression:
$((((A+B)+(C+D))+((E+F)+(G+H)))+((I+J)+((K+L)+1)))$
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

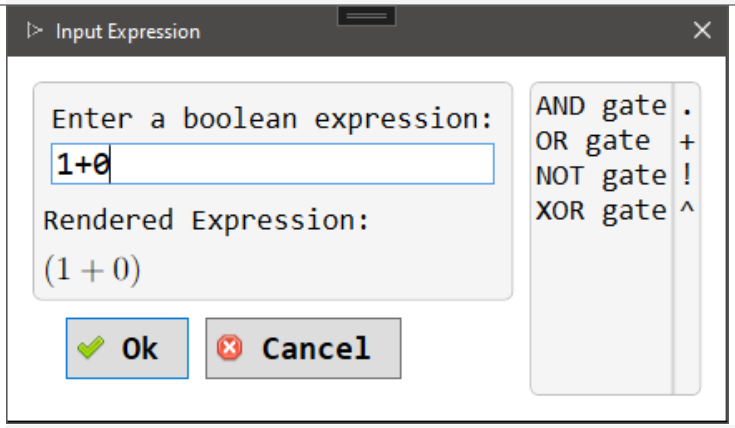

A
B
C
D
E
F
G
H
K
I
L
J
1
Q

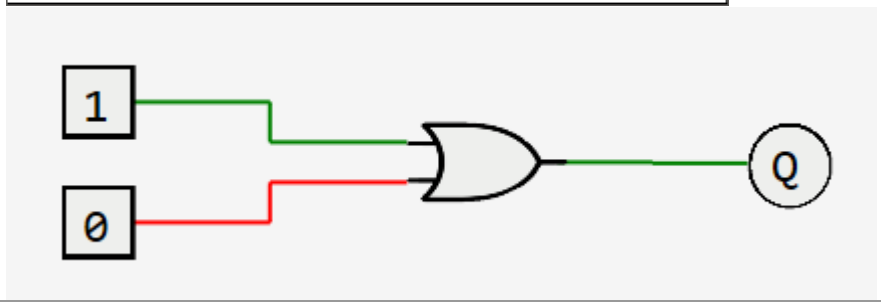
0.4.3


Input Expression
Enter a boolean expression:
0+1
Rendered Expression:
(0 + 1)
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

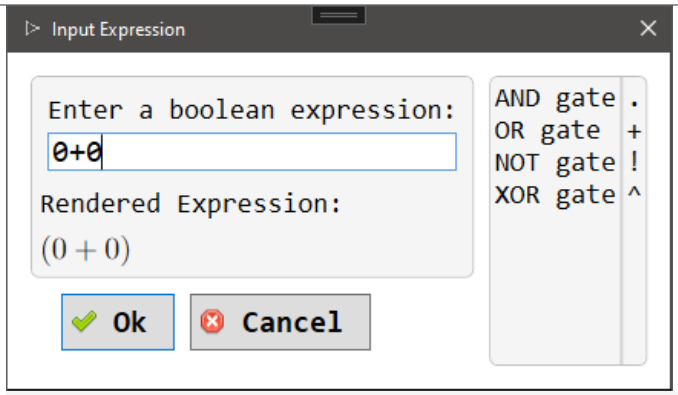

0
1
Q

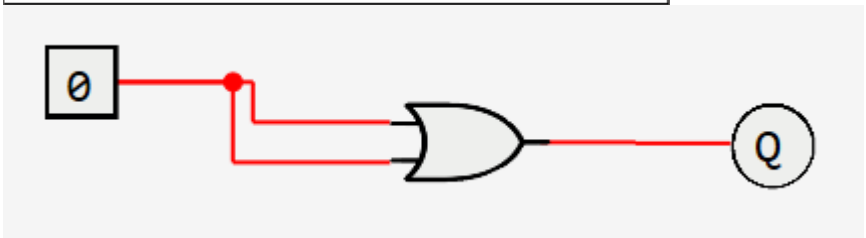
0.4.4


Input Expression
Enter a boolean expression:
1+0
Rendered Expression:
(1 + 0)
Ok
Cancel
AND gate .
OR gate +
NOT gate !
XOR gate ^

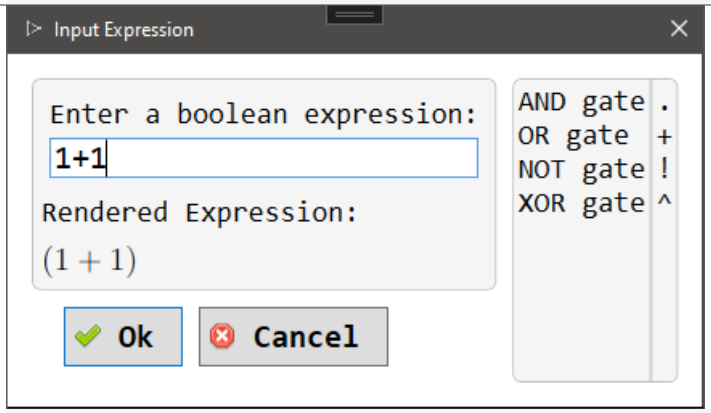

1
0
Q

0.4.5





0.4.6



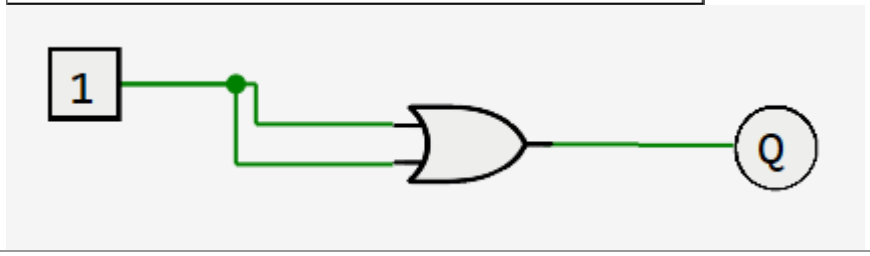


Diagram drawing

The following tests cover the process of drawing diagrams by entering Boolean expressions into the program.

Test ID	Description	Test data	Expected Outcome	Outcome
1.0.0	AND gate is loaded and drawn onto the canvas.	A.B	Diagram is drawn onto the canvas	Diagram is drawn onto the canvas.
1.0.1	OR gate is loaded and drawn onto the canvas.	A+B	Diagram is drawn onto the canvas	Diagram is drawn onto the canvas.
1.0.2	XOR gate is loaded and drawn onto the canvas.	A^B	Diagram is drawn onto the canvas	Diagram is drawn onto the canvas.
1.0.3	NOT gate is loaded and drawn onto the canvas.	!A	Diagram is drawn onto the canvas	Diagram is drawn onto the canvas.
1.0.4	Diagram is drawn onto the canvas.	B.A	Diagram is drawn onto the canvas	Diagram is drawn onto the canvas.
1.0.5	Adding brackets to expression. It should not change the diagram	(A+B)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.

1.0.6	Adding brackets to expression. It should not change the diagram.	(B+A)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.0.7	Basic expression including constants because they should also be drawn the same as the inputs.	A+0	Diagram is drawn onto the canvas. Two nodes are drawn.	Diagram is drawn onto the canvas with the constant displayed with its own input pin.
1.0.8	Diagram of only constants.	0+0	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas with a repeated wire and junction.
1.0.9	Diagram of only constants.	0+1	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas with no junction.
Section comment: The following tests (id's 1.1.0 to 1.1.16) focus on expressions with repeated inputs. This is to test the wire drawing by the program.				
1.1.0	Simple repeated input case.	(A.A)	Diagram is drawn onto the canvas with a junction where the two wires join.	Expression does not pass validation

Test comment: The validation does not accept the expression when validating it. From debugging, the IsSequential is returning false when checking this expression which is immediately discounting the expression, falsely.

Solution: The GetOnlyInputs stores the inputs as a list of characters. This then returned as an array of unique values using the Distinct() method. The CheckBracketPosition return values have also been changed. These changes can be seen below:

```
        private static char[] GetOnlyInputs(string expression)
        {
            var input = new List<char>();
            foreach (char c in expression)
            {
                // Ensuring only unique inputs are added to the array.
                if (char.IsLetter(c))
                {
                    input.Add(c);
                }
            }
            // Returning the unique inputs of expression.
            return input.Distinct().ToArray();
        }

        private static bool CheckBracketPosition(string expression)
        {
            // Regular expression to check for empty brackets within the expression.
            var bracketCheck = new Regex(@"\(\)");
            var match = bracketCheck.Match(expression);
            if (match.Success)
            {
                return true;
            }
            else
            {
                var gateCheck = new Regex(@"\([A-Z0-1]*[!.+^][A-Z0-1]\)");
                match = gateCheck.Match(expression);
                if (match.Success)
                {
```

```
                var operandCheck = new Regex(@"\([A-Z0-1][.+^][A-Z0-1]\)");
                match = operandCheck.Match(match.Value);
                if (match.Success)
                {
                    return false;
                }
                else
                {
                    return true;
                }
            }
            else
            {
                return true;
            }
        }
    }
```

`1.1.0	Simple repeated input case.	(A.A)	Diagram is drawn onto the canvas with a junction where the two wires join.	Diagram is drawn onto the canvas with a junction where the two wires join.
1.1.1	Expression that has a repeated input.	(A.B)+A	Diagram is drawn onto the canvas with an arc in the wire.	Diagram is drawn onto the canvas with an arc in the wire.
1.1.2	Expression that has a repeated input as a left child of the root node.	(A.A)+B	Diagram is drawn with 'A' wire having a junction.	Diagram is drawn with 'A' wire having a junction.
1.1.3	Expression that has a repeated input but as a right child of the root node.	A+(B.B)	Diagram is drawn with 'B' wire having a junction	Diagram is drawn with 'B' wire having a junction.
1.1.4	Expression with a repeated input.	(A.B)+B	Diagram is drawn onto the canvas with a junction on the 'B' wire.	Junction not drawn.

Test comment: The junction is not being drawn in the correct place when the 'B' wire is being drawn. This is because the program is always the adding the junction in the same place and not where the two wires intersect.

Solution: Calculate the point of the junction and add a public method for it. This has the added bonus of removing the 'repeated' field in the wire class. This create the following new and updated methods:

In Diagram.cs

```
private void DrawIntersections()
{
    // List that stores sets of points which create horizontal lines.
    // This is all of the horizontal segments of the wires.
    var horizontalLines = new List<Point>();
    // List that stores all of the vertical line segments of the wires.
    var verticalLines = new List<Point>();
    Point? intersection;
    Wire verticalWire;
    Wire horizontalWire;
    LogicGate gate;
    LogicGate gate1;
    // Add all of the horizontal and vertical points to the respective lists.
    for (var j = 0; j < wires.Length - 1; j++)
    {
        horizontalLines.AddRange(wires[j].GetPoints(true));
        verticalLines.AddRange(wires[j].GetPoints(false));
    }
```

```
            for (var i = 0; i < verticalLines.Count - 1; i += 2)
            {
                for (var c = 0; c < horizontalLines.Count - 1; c += 2)
                {
                    // Calculating the intersection, null => an intersection does not exist
                    // between these 2 lines.
                    intersection = FindIntersection(verticalLines[i], verticalLines[i + 1],
horizontalLines[c], horizontalLines[c + 1]);
                    // Ensuring the intersection exists and the intersection is not formed
                    // a horizontal and vertical line of the same wire before the
intersection
                    // is added.
                    verticalWire = FindWire(verticalLines[i]);
                    horizontalWire = FindWire(horizontalLines[c]);
                    gate = verticalWire.GetGate();
                    gate1 = horizontalWire.GetGate();
                    if (intersection != null && verticalWire != horizontalWire)
                    {
                        if (gate.GetGate().GetLabel() != gate1.GetGate().GetLabel())
                        {
                            verticalWire.AddBridge(intersection);
                        }
                        else
                        {
                            verticalWire.AddJunction(intersection);
                        }
                    }
                }
            }
        }
```

In Wire.cs

```
        public void AddJunction(Point? junctionLocation)
        {
            // Create the junction.
            junction = new Ellipse
            {
                Width = radiusOfJunction,
                Height = radiusOfJunction,
                Fill = Brushes.Black,
                Stroke = Brushes.Black
            };
            // Setting its position, which is always the joint between
            // the horizontal input wire and the vertical wire.
            Canvas.SetTop(junction, points[points.Count - 2].Y - radiusOfJunction / 2);
            Canvas.SetLeft(junction, points[points.Count - 2].X - radiusOfJunction / 2);
            // Setting the low ZIndex to stop overlapping and adding
            // the junction to the canvas.
            Panel.SetZIndex(junction, 1);
            c.Children.Add(junction);
        }
```

`1.1.4	Expression with a repeated input.	(A.B)+B	Diagram is drawn onto the canvas with a junction on the ‘B’ wire.	Diagram is drawn on the canvas with correct location for the junction.
1.1.5	Testing the repeated inputs with variables and constants.	A.1+A.1	A diagram with repeated wires going to the ‘A’ and ‘1’ pin	Expression does not pass validation

Test comment: The CheckBracketPosition method is returning the wrong value for one of the regular expression checks.

Solution: Change the return value.

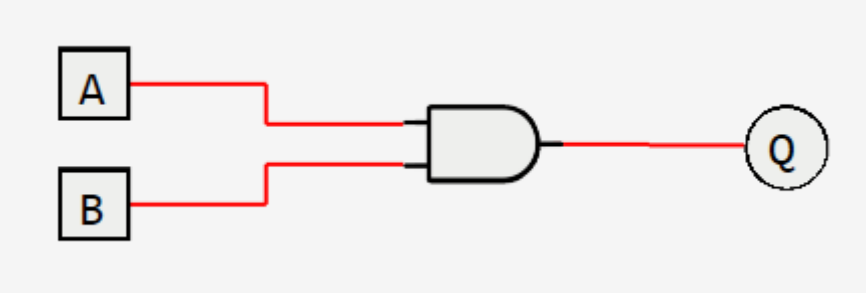
```
        private static bool CheckBracketPosition(string expression)
        {
            // Regular expression to check for empty brackets within the expression.
            var bracketCheck = new Regex(@"\(\)");
            var match = bracketCheck.Match(expression);
            if (match.Success)
            {
                return true;
            }
            else
            {
                var gateCheck = new Regex(@"\([A-Z0-1]*[!.+^][A-Z0-1]\)");
                match = gateCheck.Match(expression);
                if (match.Success)
                {
                    var operandCheck = new Regex(@"\([A-Z0-1][.+^][A-Z0-1]\)");
                    match = operandCheck.Match(match.Value);
                    if (match.Success)
                    {
                        return false;
                    }
                    else
                    {
                        return true;
                    }
                }
                else
                {
                    return false;
                }
            }
        }
```

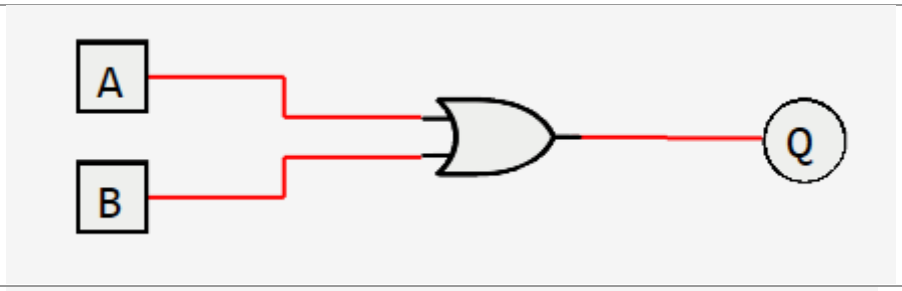
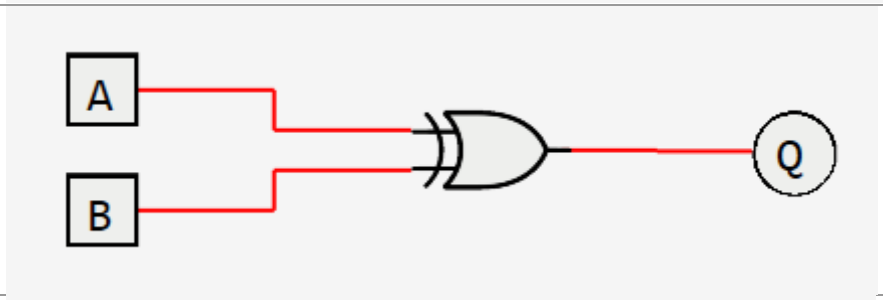
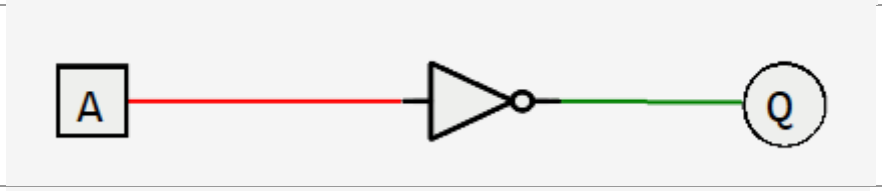
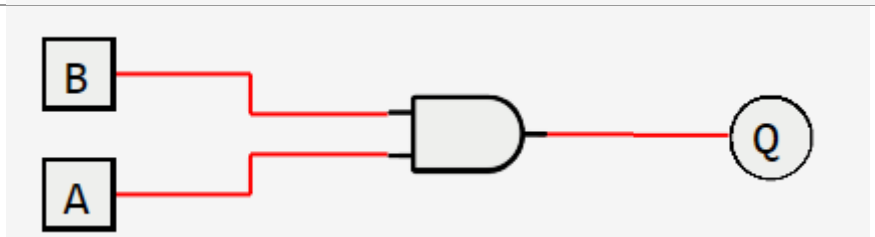
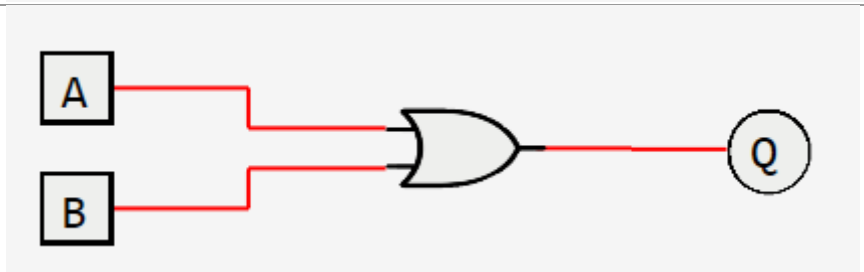
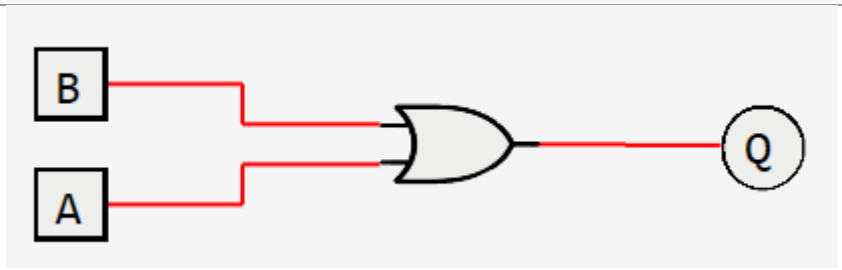
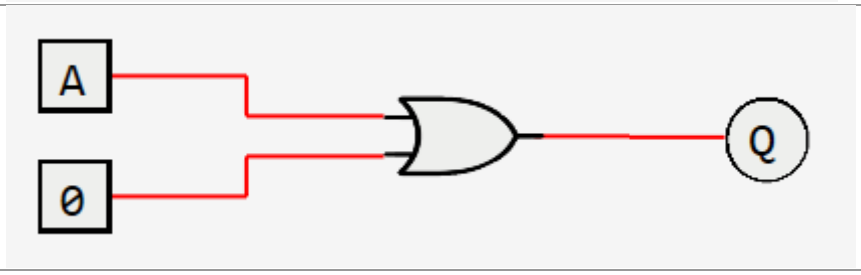
`1.1.5	Testing the repeated inputs with variables and constants	A.1+A.1	A diagram with repeated wires going to the ‘A’ and ‘1’ pin.	Diagram is drawn onto the canvas.
1.1.6	Repeated inputs where ‘B’ is lower down in the tree.	(A.B)+(A.!B)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.1.7	Nested NOT gates as they only have one child and have their children drawn in parallel.	!(!(A.B).!A)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.1.8	Expression that produces a balanced tree but a complex node/wire structure due to the NOT gates.	!(!(A+B).!(!A+C))	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.1.9	Repeated inputs however the left child is higher up in the tree.	!A+!(B+A)	Diagram is drawn onto the canvas	Diagram is drawn onto the canvas.
1.1.10	Using Nested NOT gates.	(A.B)^!(!C)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.1.11	Simple repeated input case.	(A.B)^(A+A)	Diagram is drawn onto the canvas. An arc should also be drawn.	Diagram is drawn onto the canvas.
1.1.12	Simple repeated input case.	(A.B)+(!C.A)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.

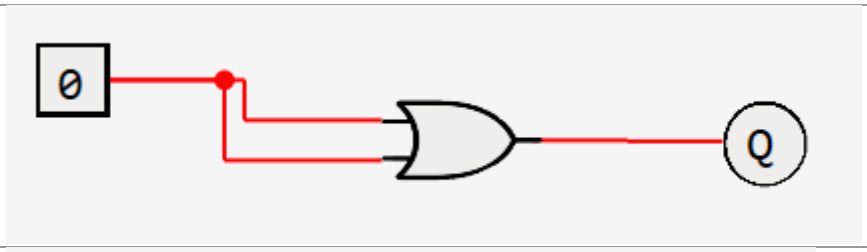
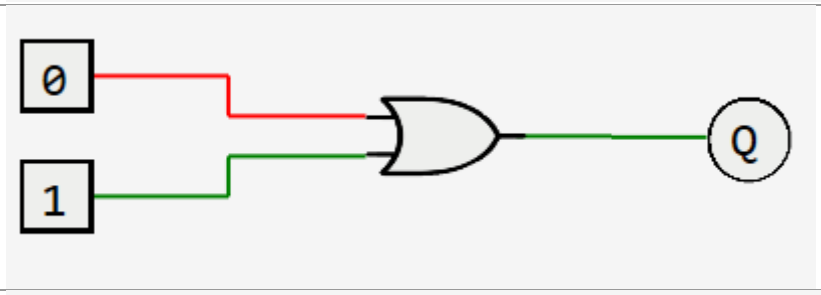
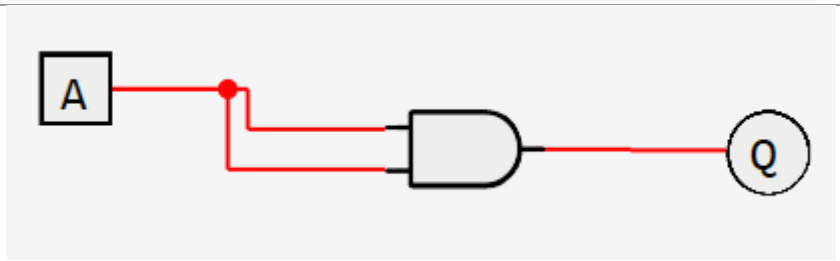
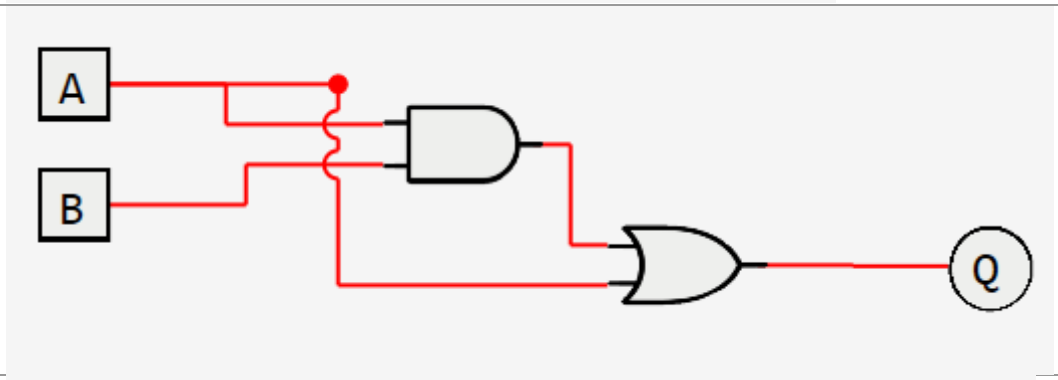
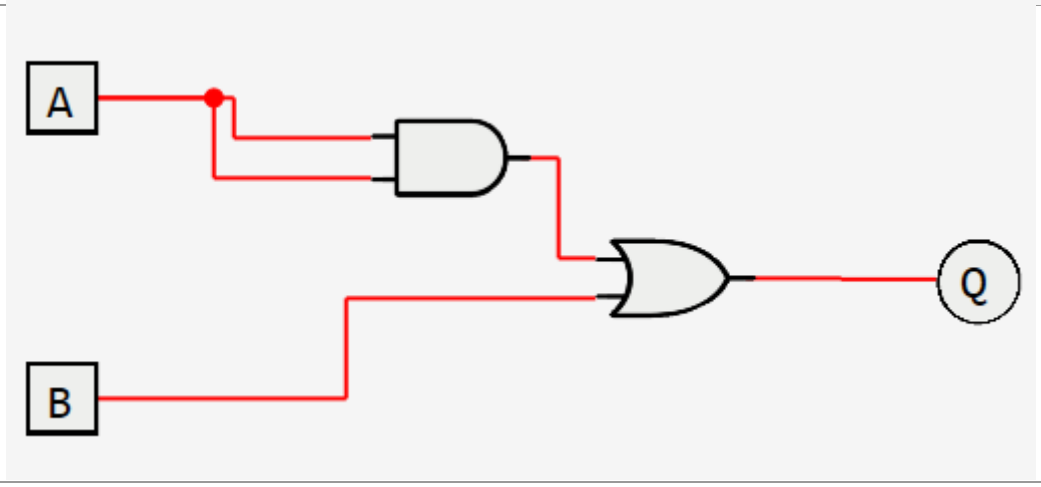
1.1.13	Simple repeated input case.	(A.B)^(A+C)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.1.14	Lop-sided tree due to repeated 'A' and also reversed input order for 'B' and 'C'	(A+A)^(C+B)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.1.15	'A' input nodes are furthest apart within the diagram.	(A.B)^(C+A)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.1.16	Lop-sided tree due to the repeated input and NOT gate.	A.(!A+B)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
Section comment: The following tests (id's 1.2.0 to 1.2.11) cover expressions that do not fit into the previous categories but should be included for completeness.				
1.2.0	Larger trees using brackets	(A.B)+C	Diagram is drawn onto the canvas	Diagram is drawn onto the canvas.
1.2.1	Small chain of AND gates to test the spacing of the nodes.	A.B.C	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.2	4-input expression that produces a balanced tree.	(A.B)^(C.D)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.3	4-input tree that is not balanced due to the NOT gate.	(A.B)+(!C^D)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.4	Same as the previous test however, the inputs have been re-ordered for node positioning.	(A.C)^(B+!D)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.5	Expression that is represented by a lop-sided tree. This example the right child has a greater max depth compared to the left child.	A.(!B+!C)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.6	Expressions with constants.	(A+(!B+0))	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.7	Expression with a single constant and a more complex input '!A'	!A+0	Diagram with two inputs is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.8	NOT inputs that are children of another NOT gate.	!(!A+!B)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.9	Maximum valid number of inputs to test the node spacing.	(((A+B)+(C+D))+ ((E+F)+(G+H)))+ ((I+J)+(K+L))	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.10	Boolean expression that results in an un-even tree.	(!A.!B)^(!A.B)	Diagram is drawn onto the canvas.	Diagram is drawn onto the canvas.
1.2.11	Null expression	" "	Error message shown	Error message shown

Evidence of tests

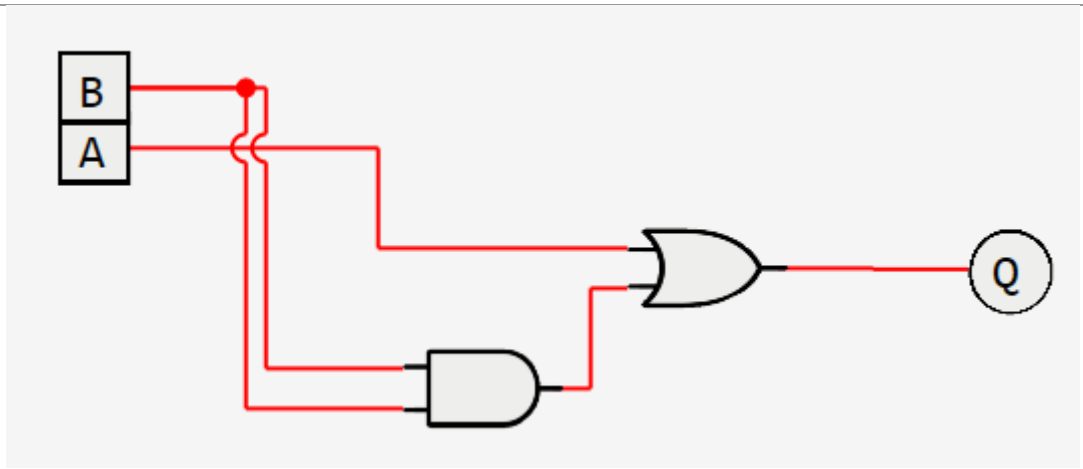
The following table contains all of the screenshots showing that each test in the table above has been completed.

Test ID	Evidence
1.0.0	

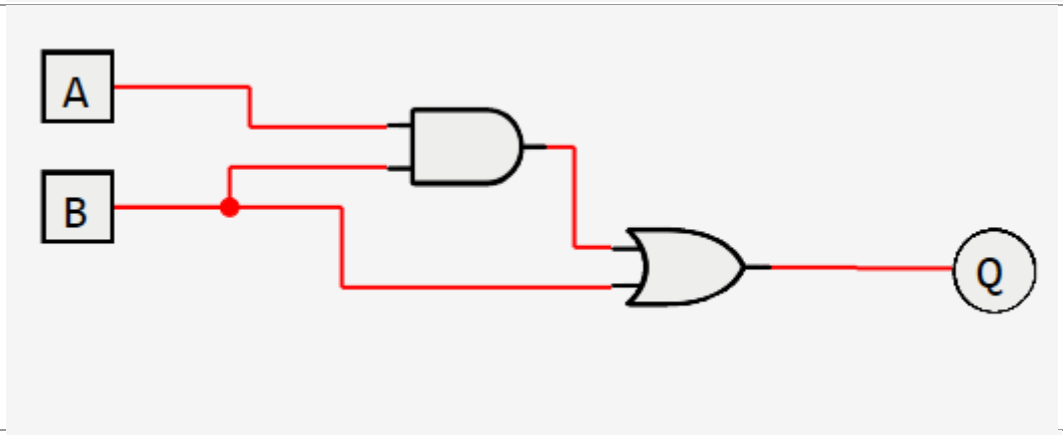
1.0.1	
1.0.2	
1.0.3	
1.0.4	
1.0.5	
1.0.6	
1.0.7	

1.0.8	
1.0.9	
1.1.0	
1.1.1	
1.1.2	

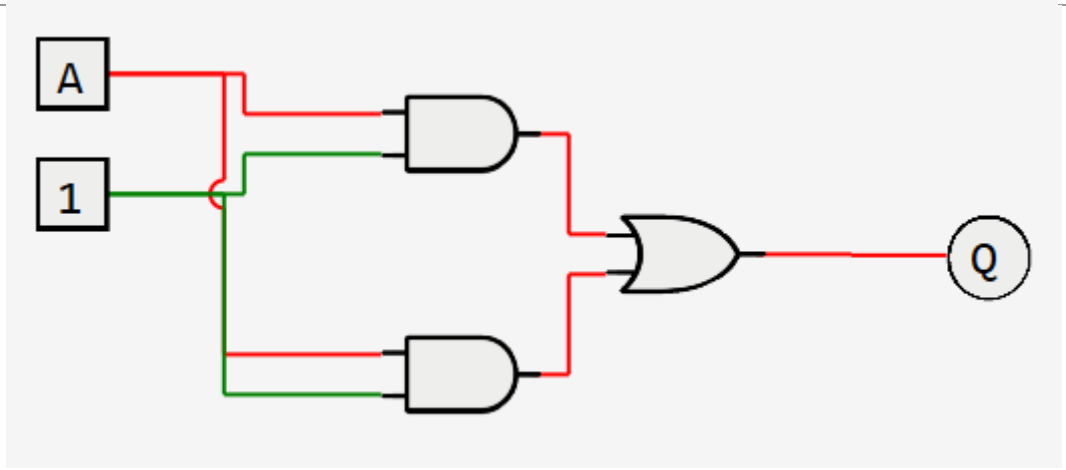
1.1.3



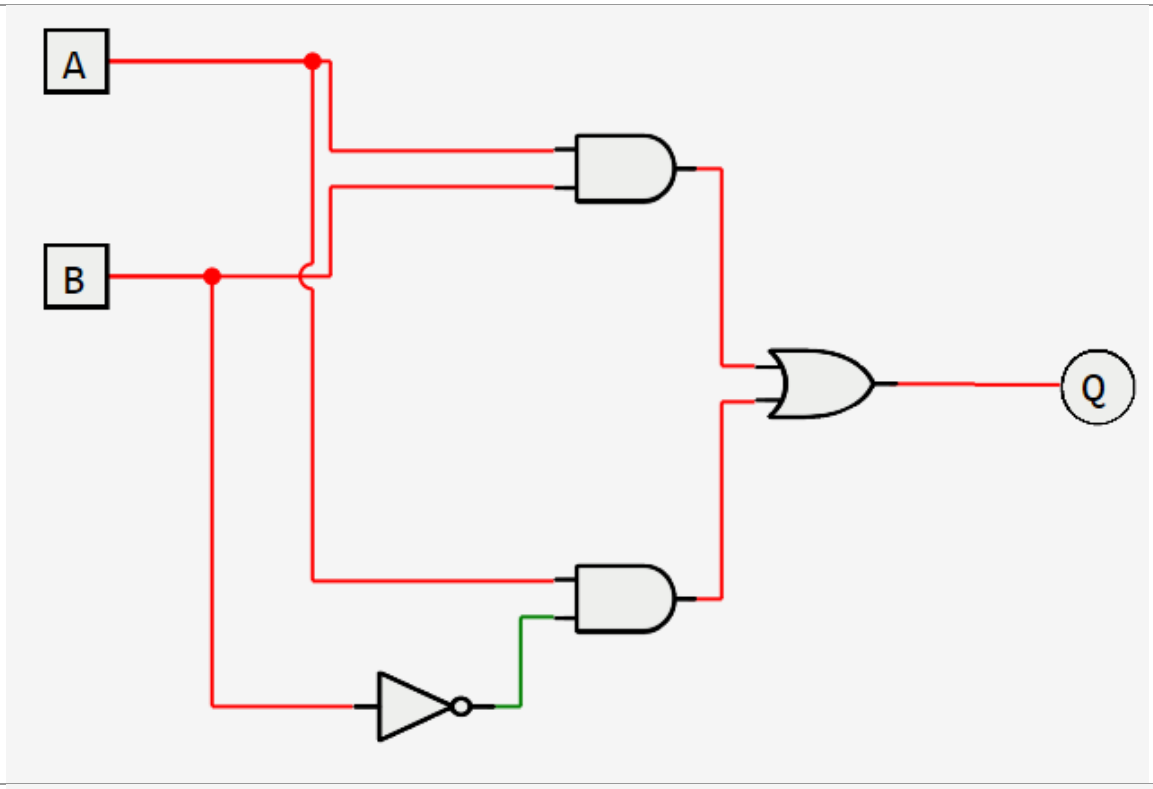
1.1.4



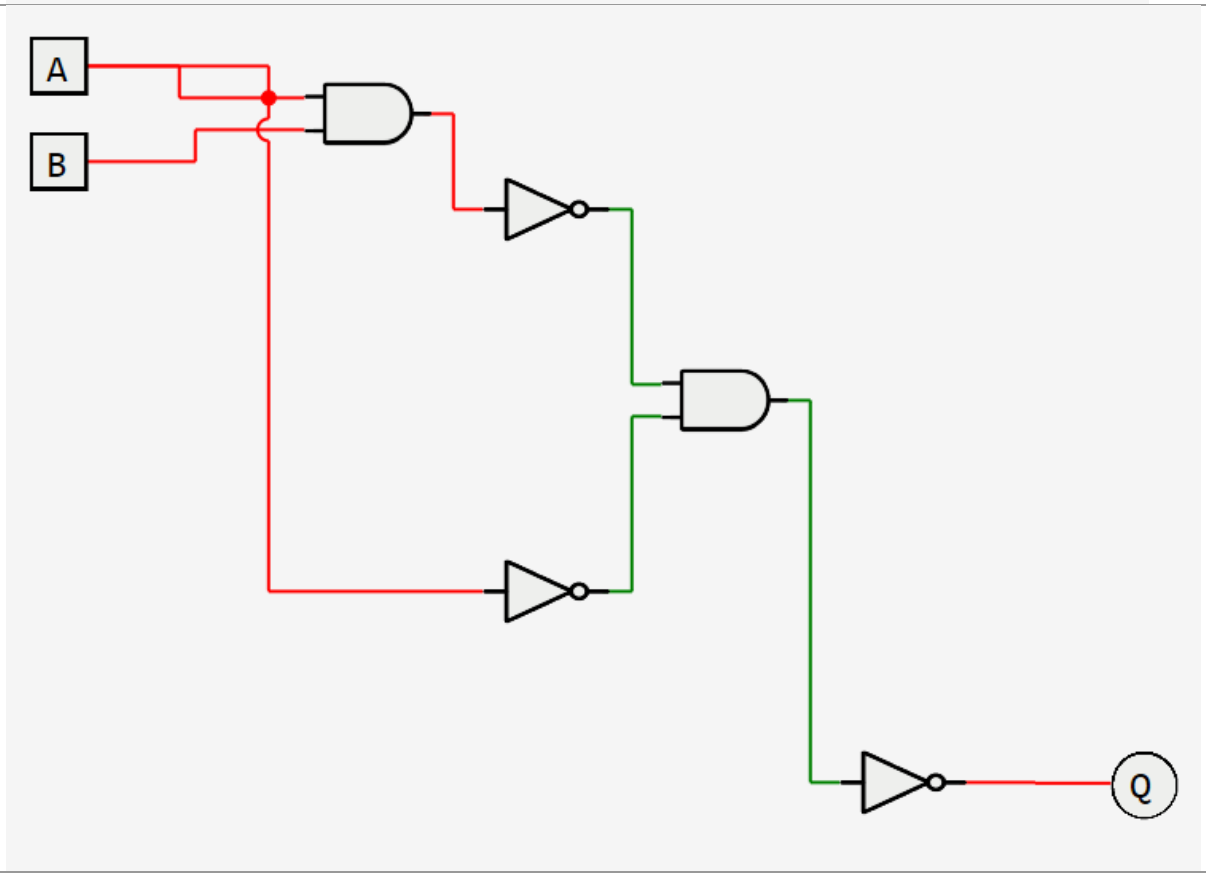
1.1.5



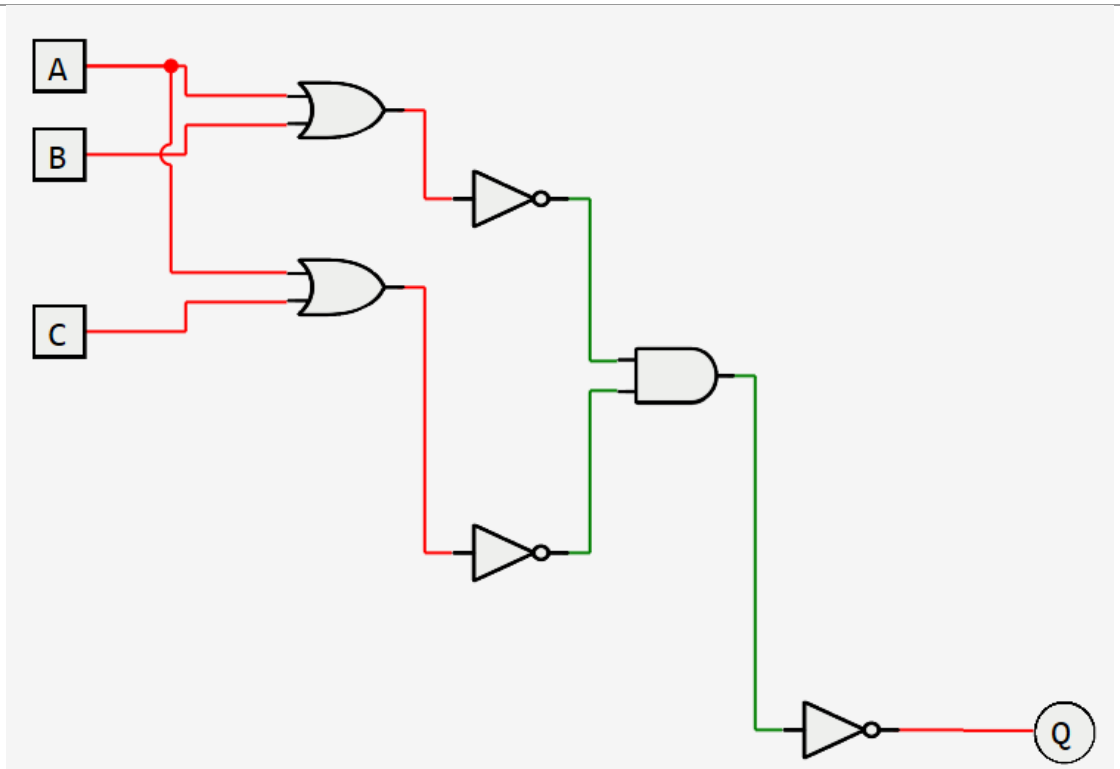
1.1.6



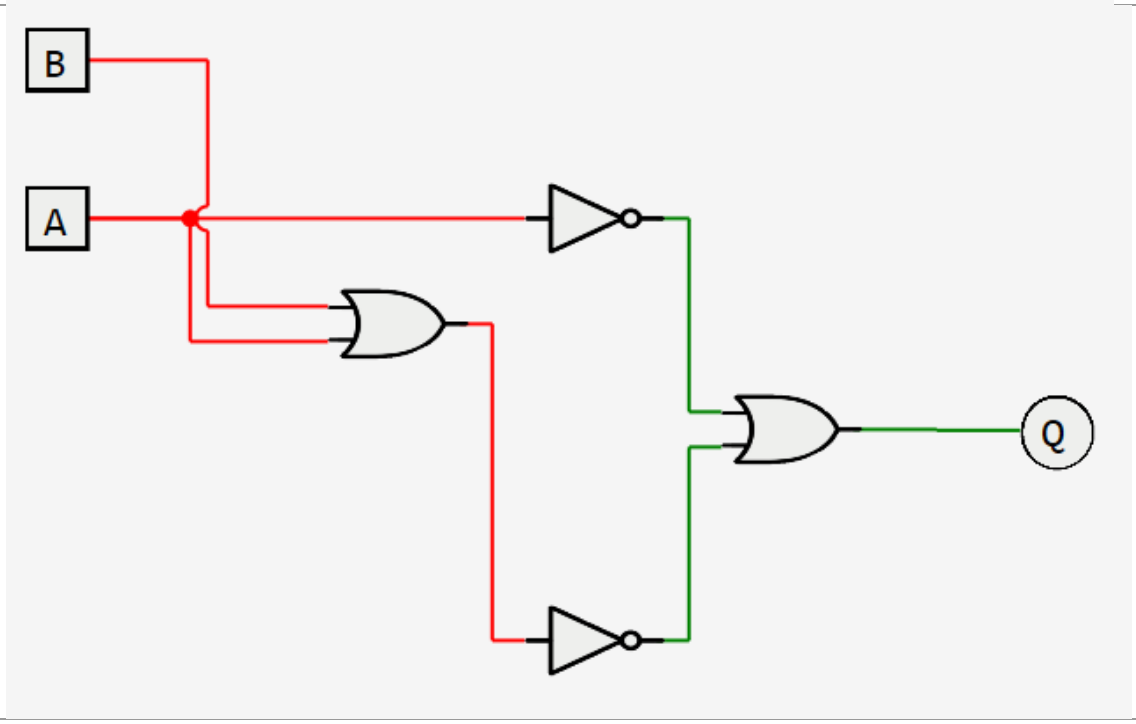
1.1.7



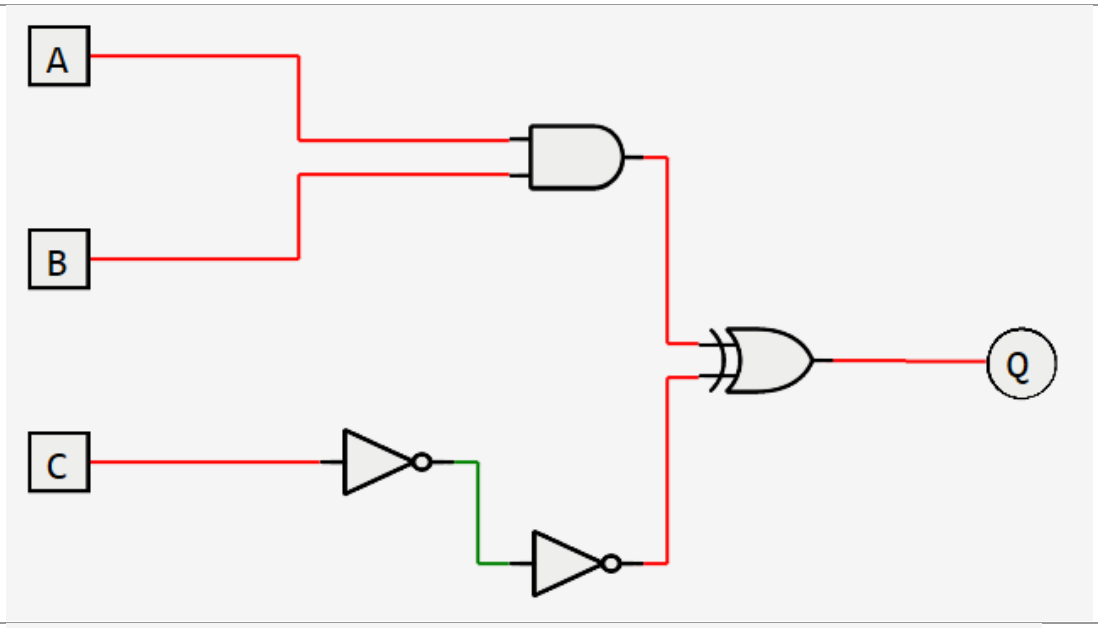
1.1.8



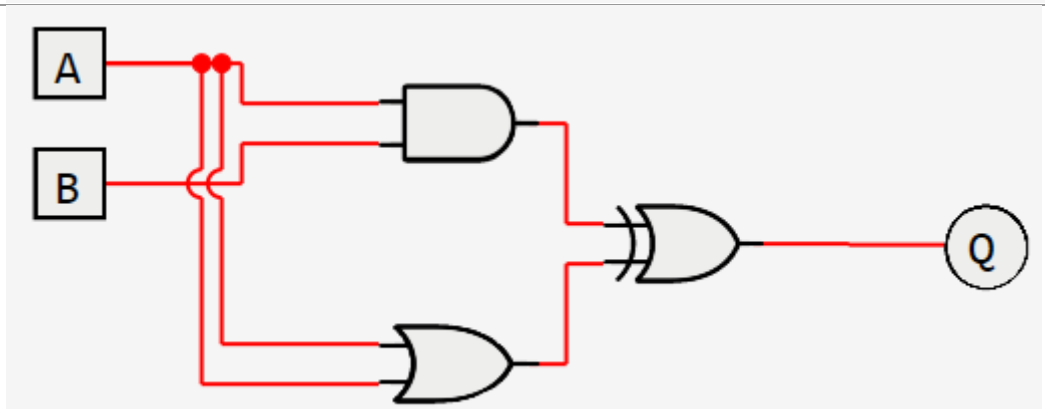
1.1.9



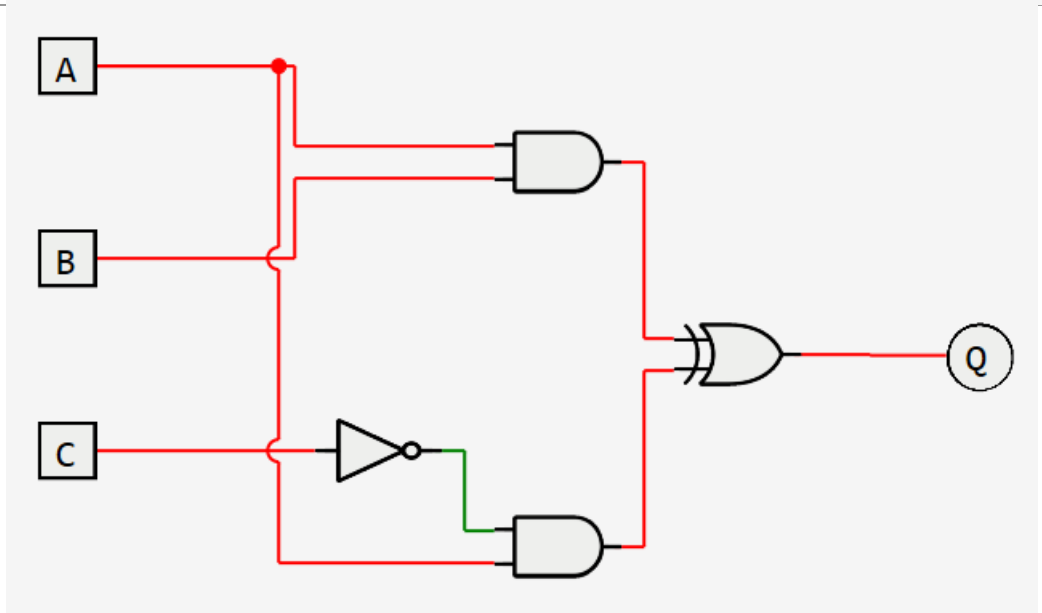
1.1.10



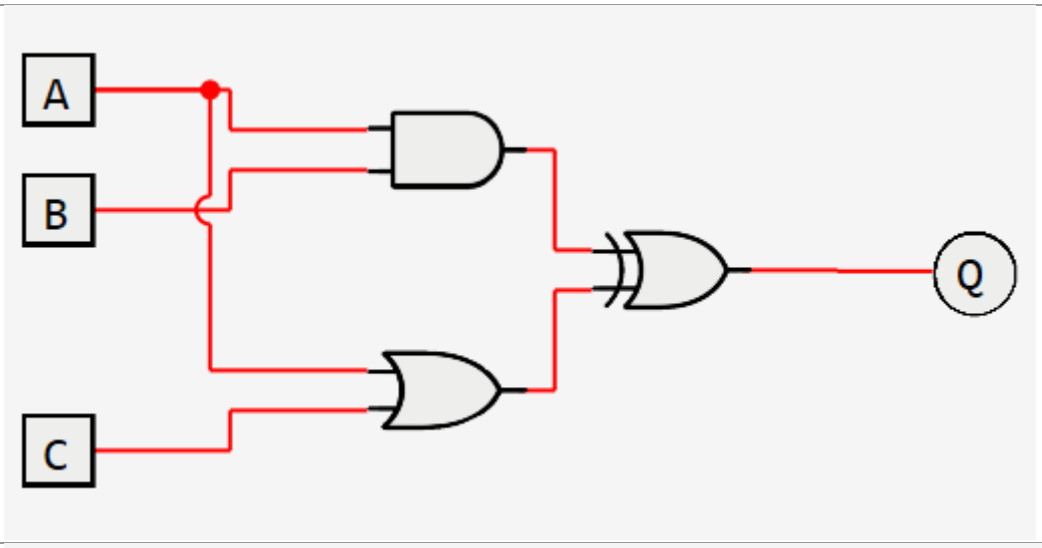
1.1.11



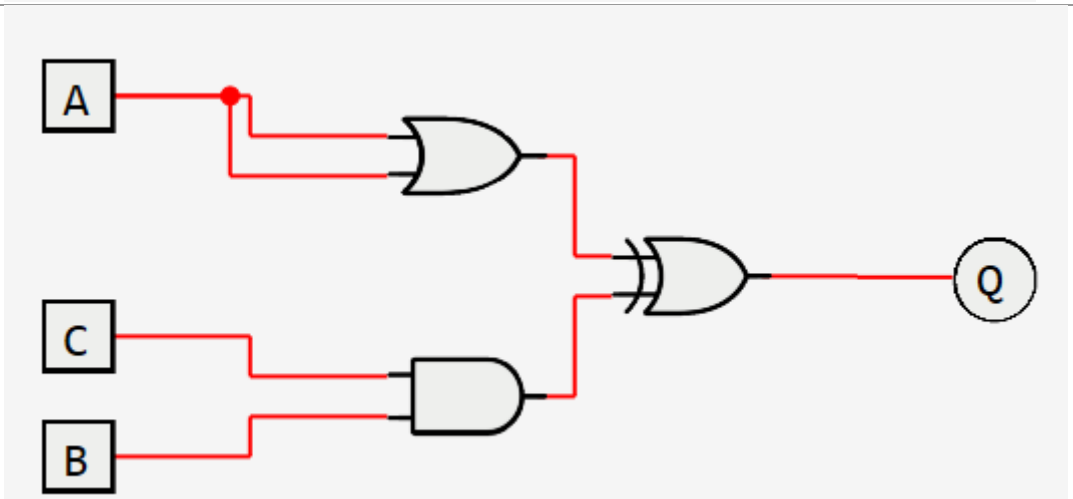
1.1.12



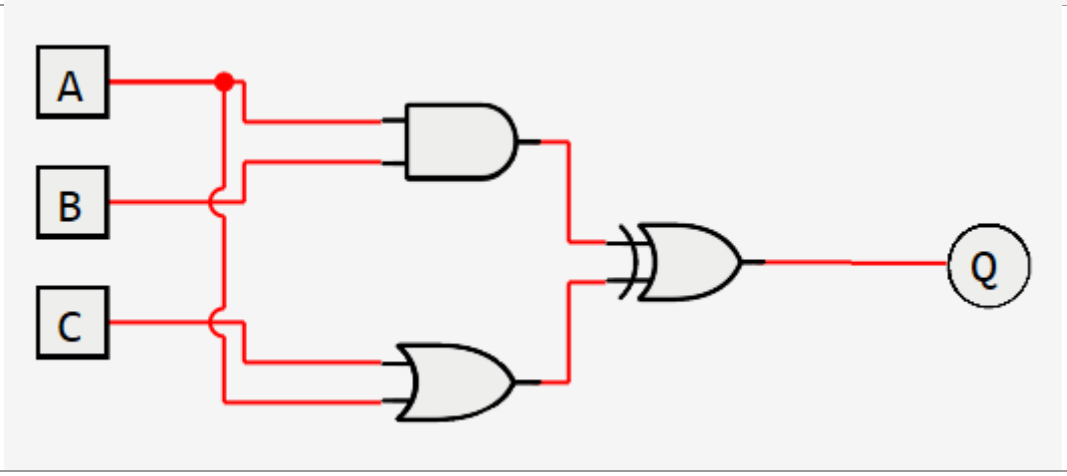
1.1.13



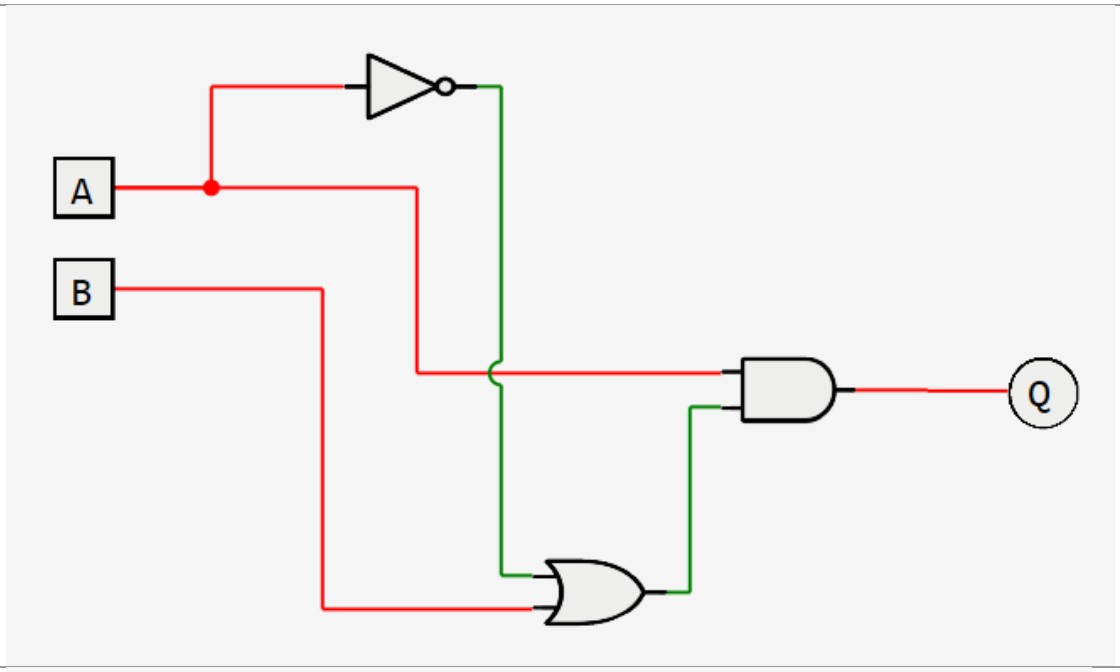
1.1.14



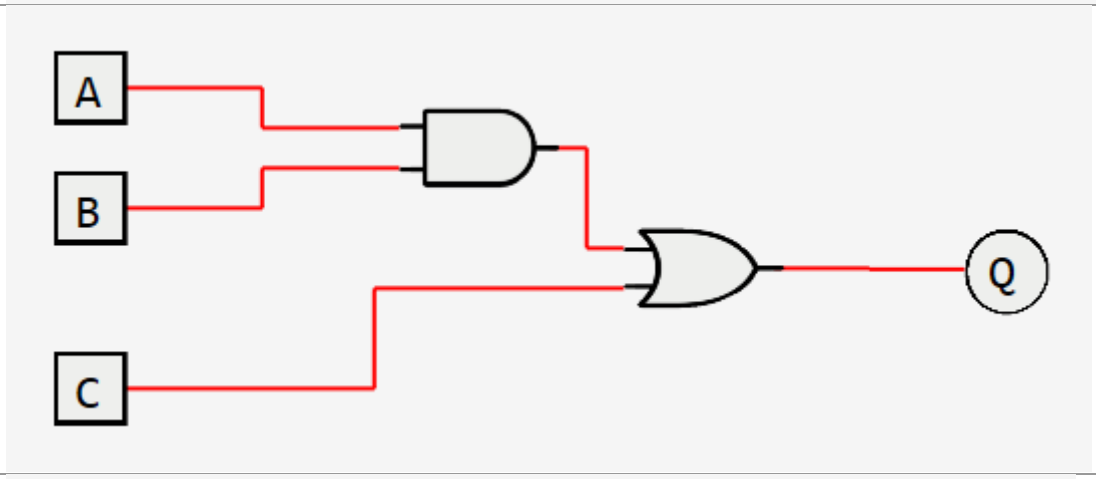
1.1.15



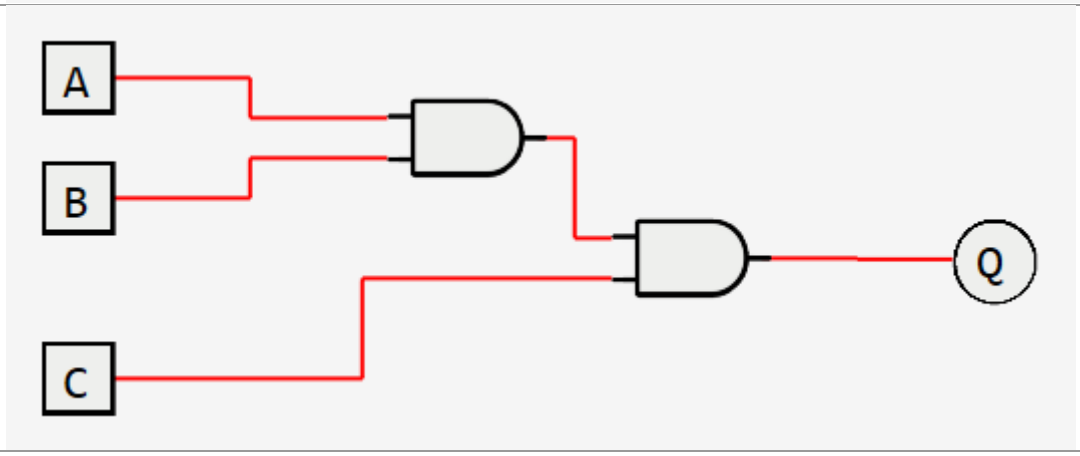
1.1.16



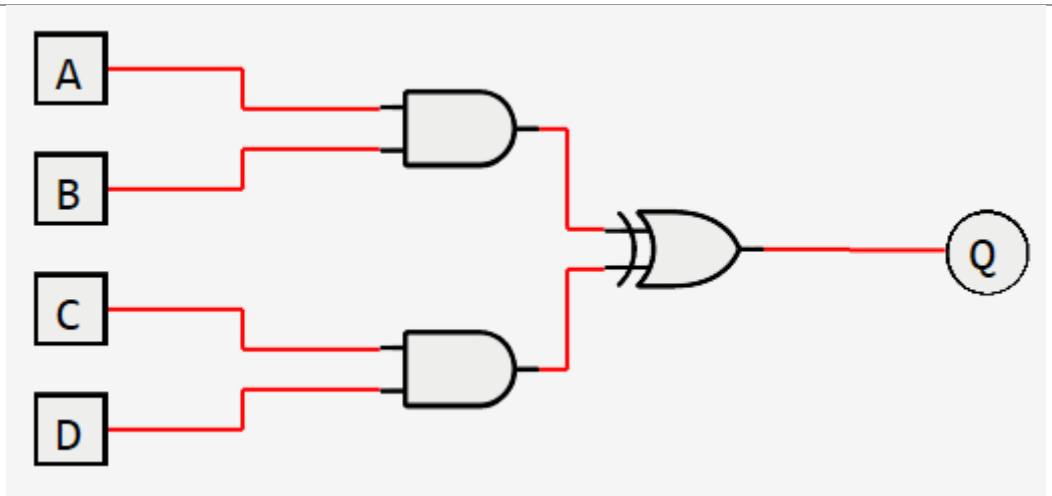
1.2.0



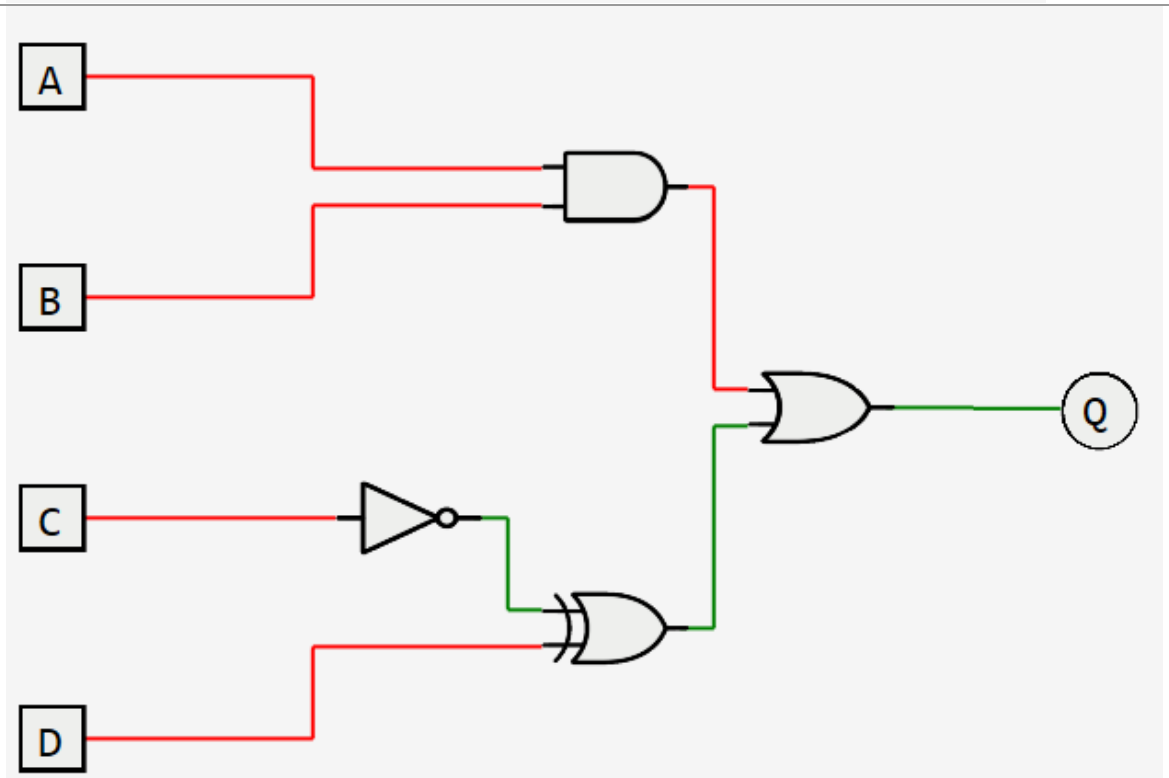
1.2.1



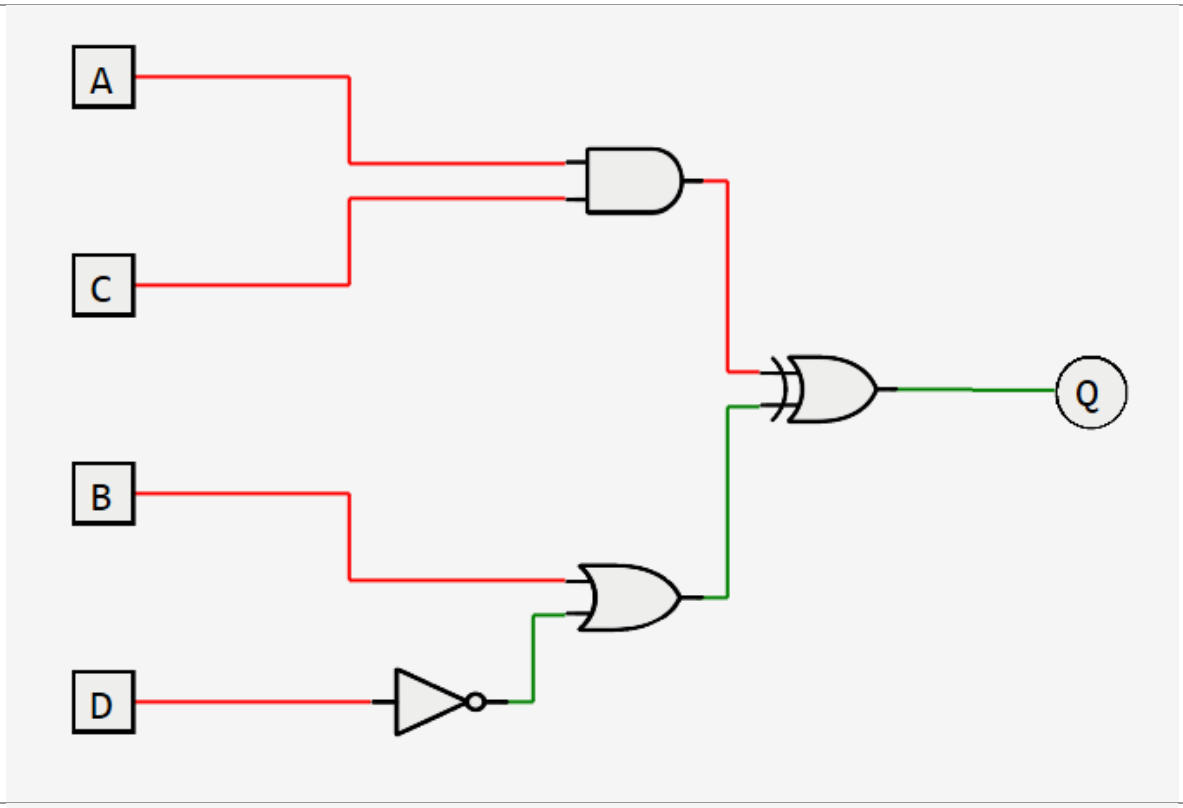
1.2.2



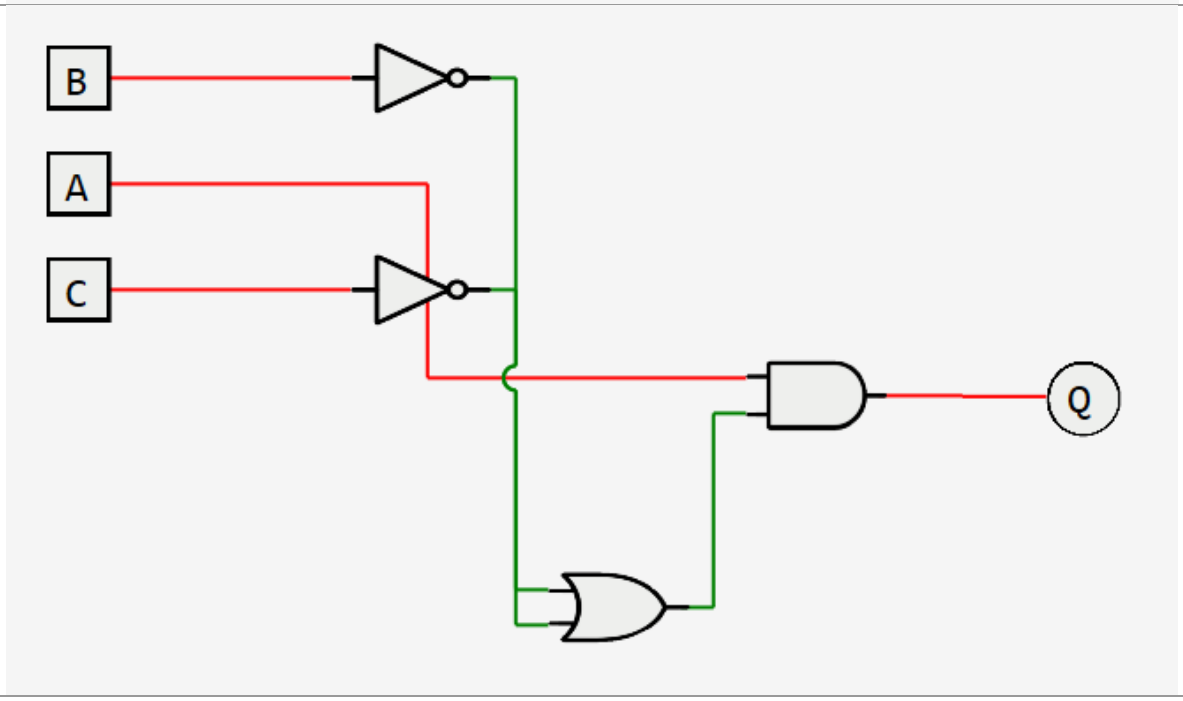
1.2.3



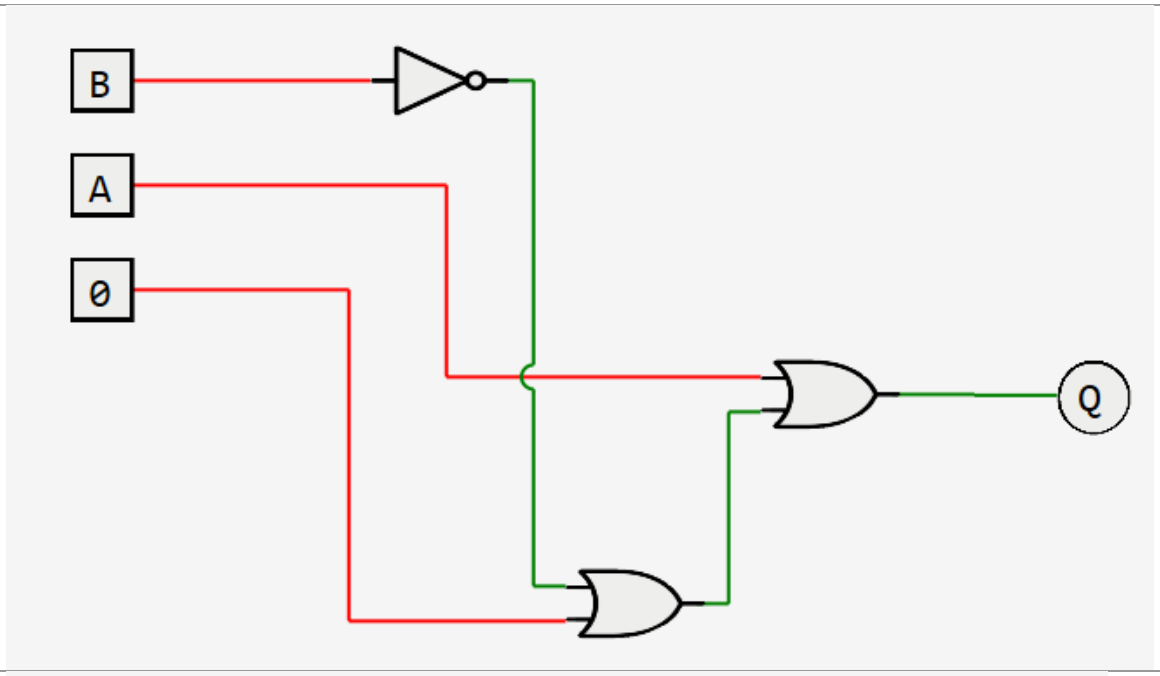
1.2.4



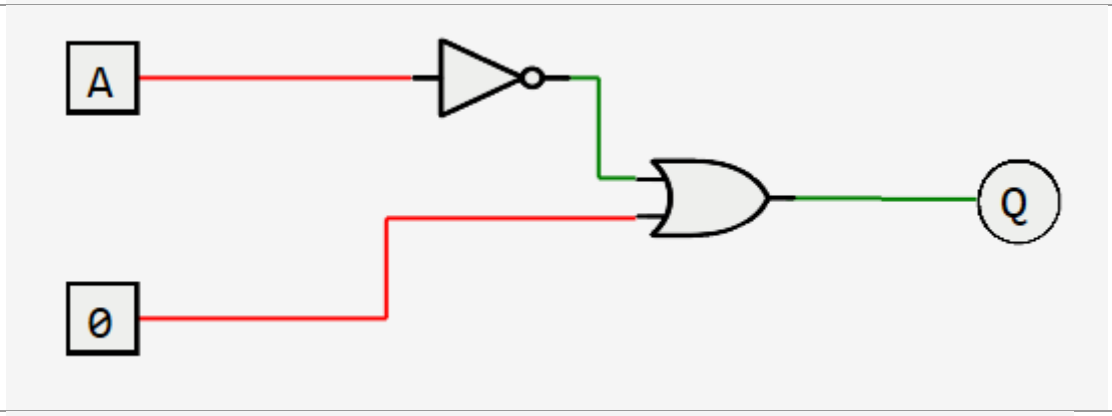
1.2.5



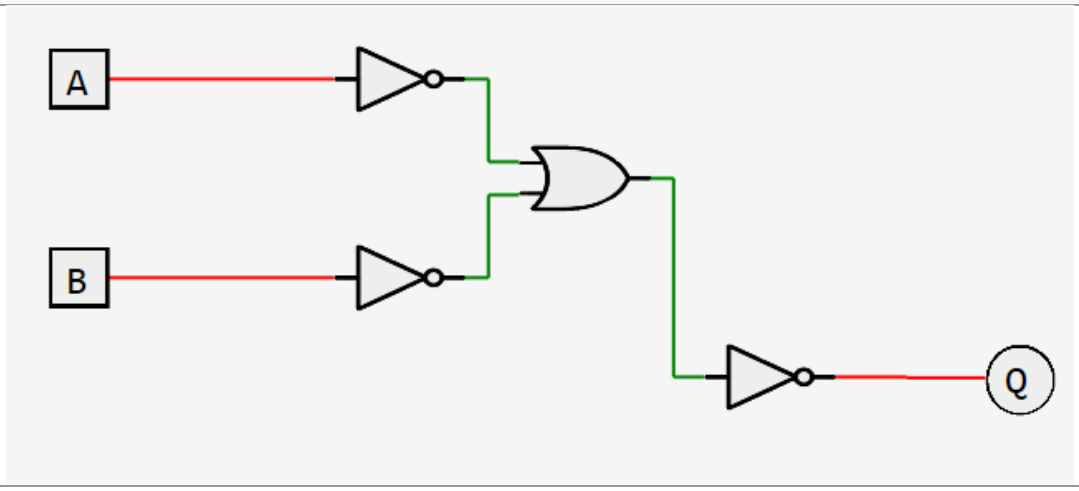
1.2.6



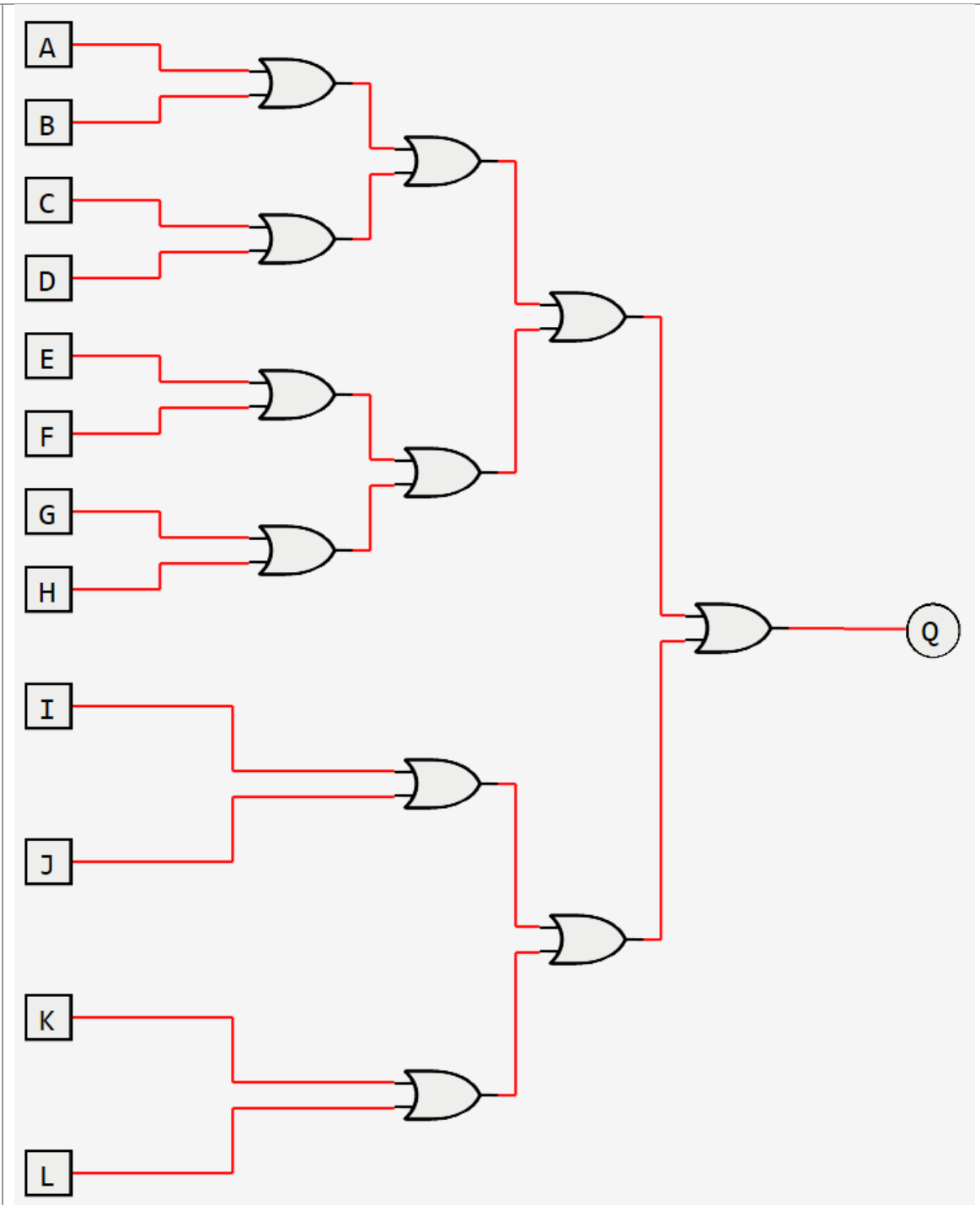
1.2.7



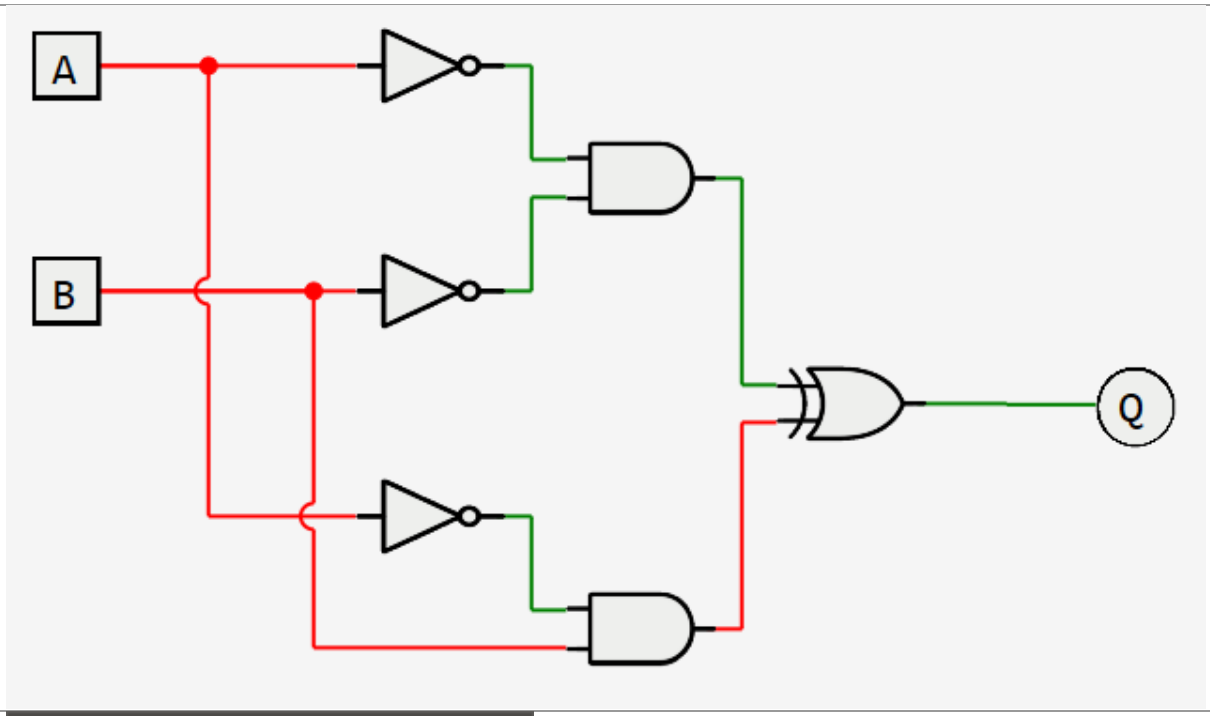
1.2.8



1.2.9



1.2.10



1.2.11

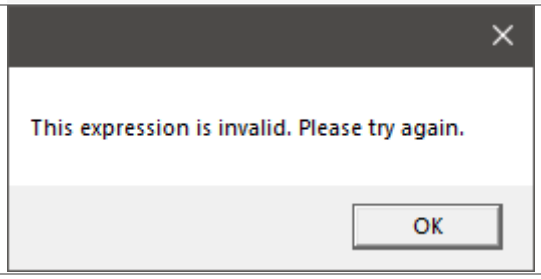


Diagram interactivity

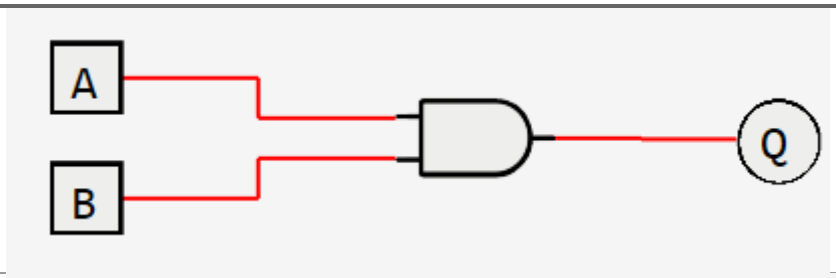
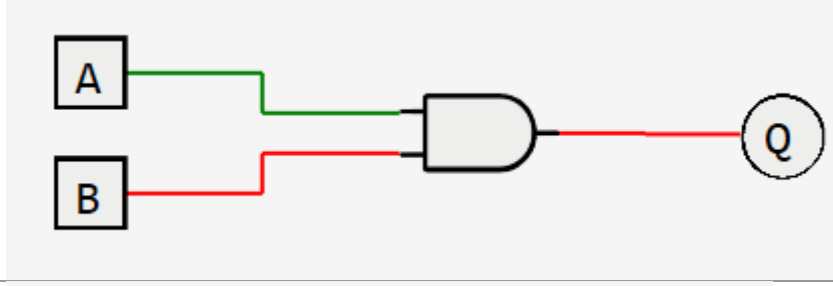
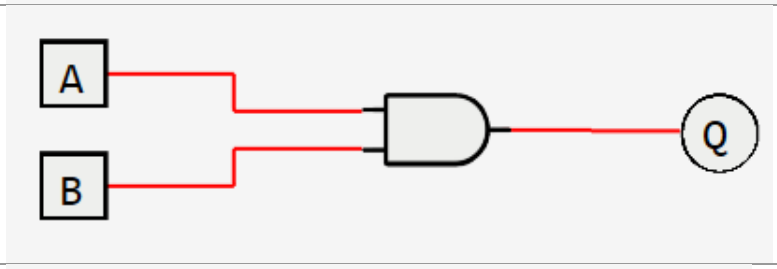
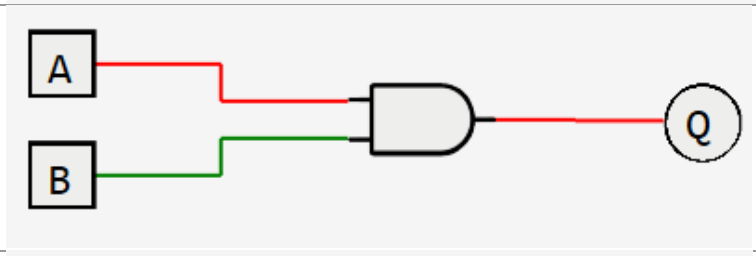
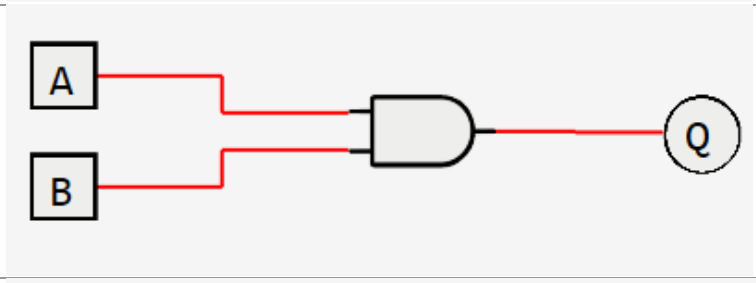
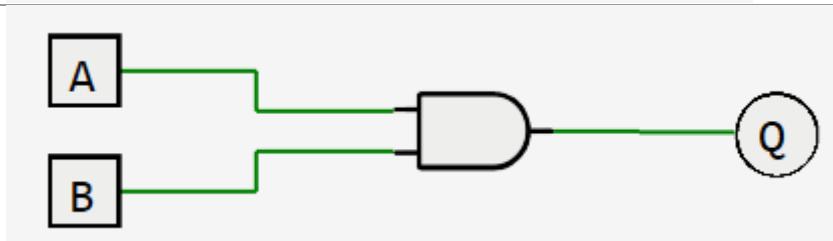
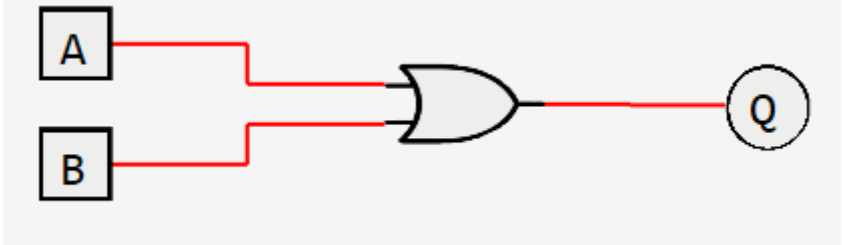
The following tests cover diagram interactivity and the behaviour of the logic gate diagrams when the input pins are clicked by the user, this allows them to dry run their diagrams and see how their inputs go through the diagram. It should be noted that the test data for this set of tests are the drawn logic diagrams. In the table below, the diagram representing the Boolean expression and the inputs that will be clicked are given. Constants have also not been included in these set of tests as their value never changes and so they are not interactive inputs of the logic gate diagram.

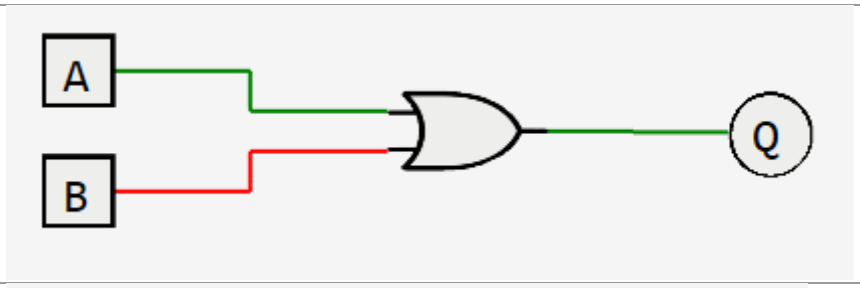
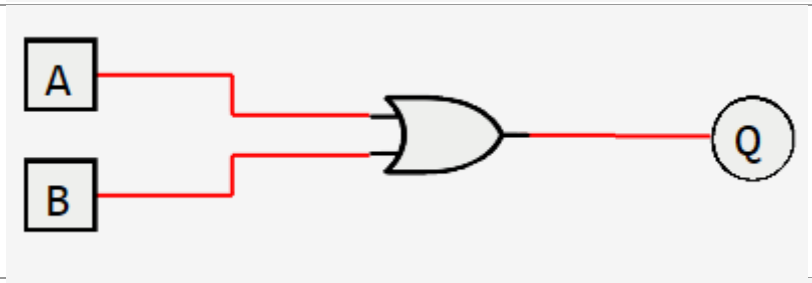
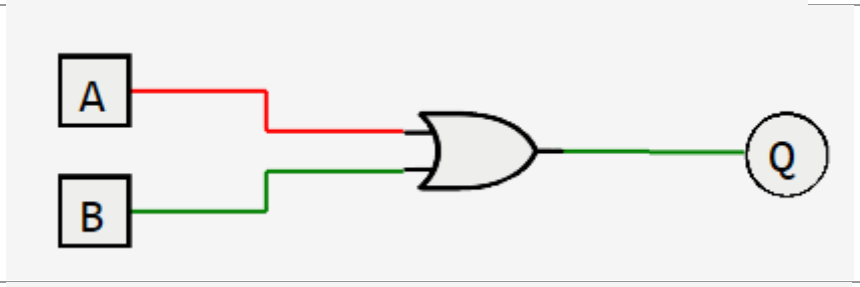
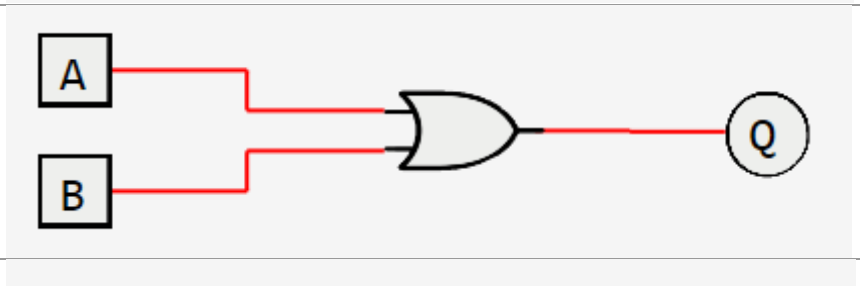
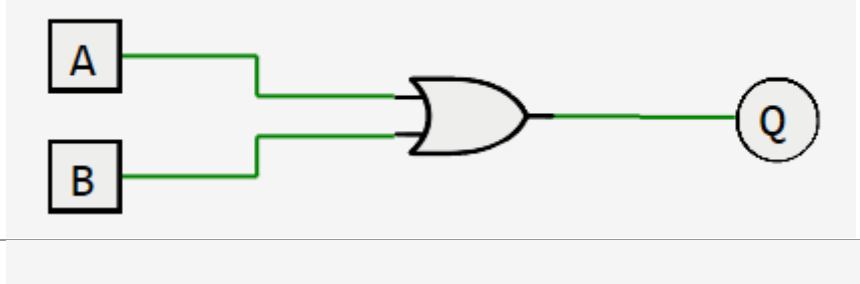
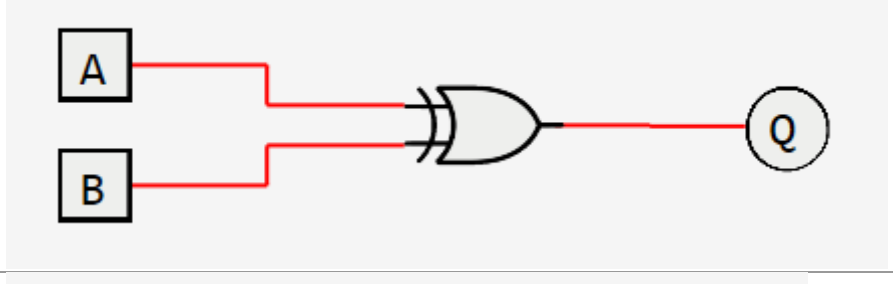
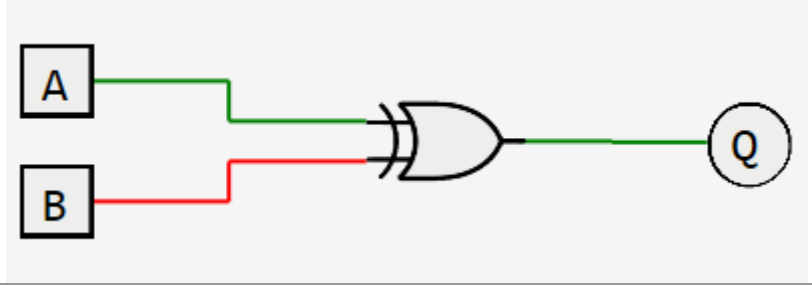
Test ID	Description	Test data	Expected outcome	Outcome
2.0.0	Simple AND gate	A.B – Click ‘A’	‘A’ wire is green.	‘A’ wire is green.
2.0.1	Simple AND gate	A.B – Click ‘B’	‘B’ wire is green.	‘B’ wire is green.
2.0.2	Simple AND gate	A.B – Click ‘A’ and ‘B’	‘A’ and ‘B’ and output of AND gate is green.	‘A’ and ‘B’ and output of AND gate is green.
2.0.3	Simple OR gate	A+B – Click ‘A’	‘A’ and output of OR gate is green.	‘A’ and output of OR gate is green.
2.0.4	Simple OR gate	A+B – Click ‘B’	‘B’ and output of OR gate is green.	‘B’ and output of OR gate is green.
2.0.5	Simple OR gate	A+B – Click ‘A’ and ‘B’	‘A’ and ‘B’ and output of OR gate is green.	‘A’ and ‘B’ and output of OR gate is green.
2.0.6	Simple XOR gate	A^B – Click ‘A’	‘A’ wire and output of XOR is green.	‘A’ wire and output of XOR is green.

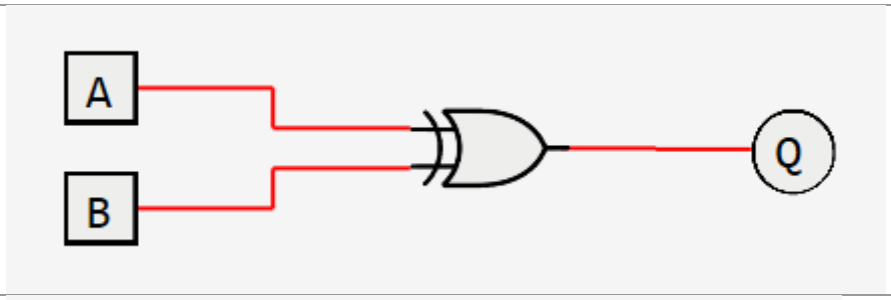
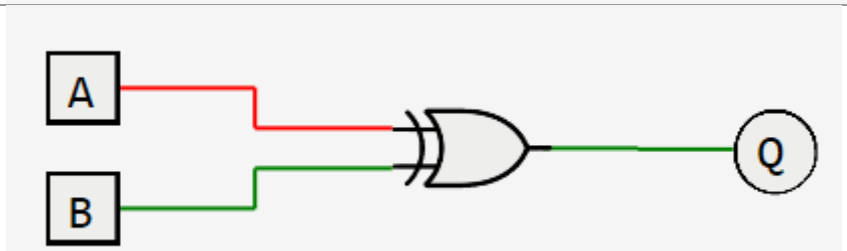
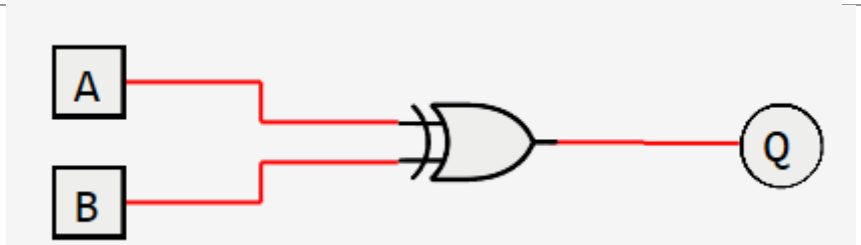
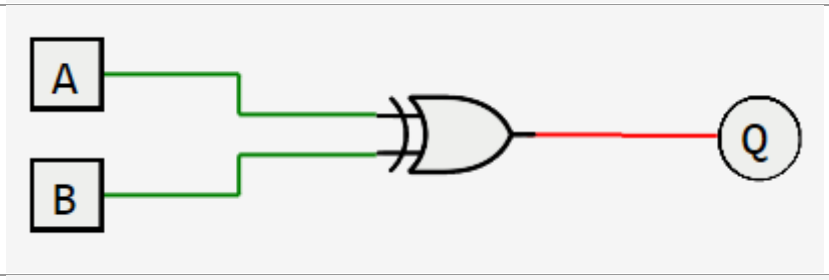
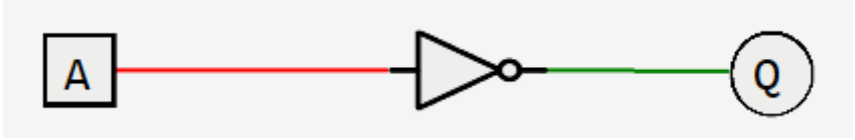
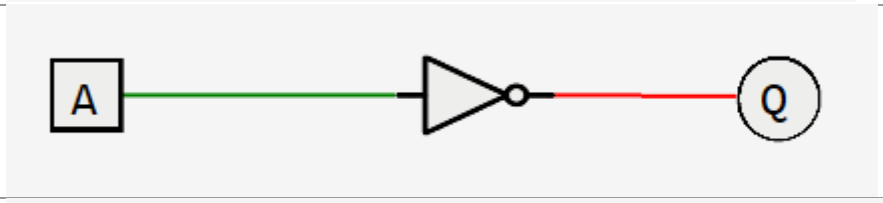
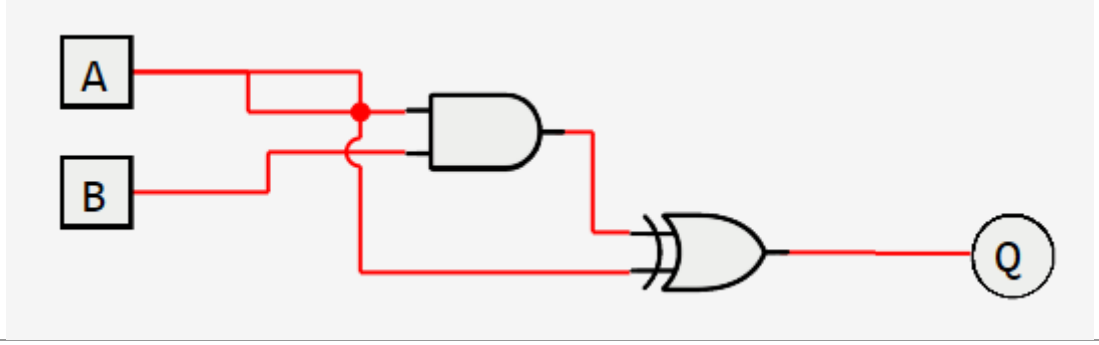
2.0.7	Simple XOR gate	A^B – Click 'B'	'B' wire and output of XOR is green.	'B' wire and output of XOR is green.
2.0.8	Simple XOR gate	A^B – Click 'A' and 'B'	'A' and 'B' is green but not the output.	'A' and 'B' is green but not the output.
2.0.9	Simple NOT gate	!A – Click 'A'	'A' is green but not the output.	'A' is green but not the output.
2.0.10	Expression with repeated input.	(A.B)^A – Click 'A'	All wires going to 'A' and the output are green.	All wires going to 'A' and the output are green.
2.0.11	Expression with repeated input	(A.B)^A – Click 'B'	Only the 'B' wire is green.	Only the 'B' wire is green.
2.0.12	Expression with repeated input.	(A.B)^A – Click 'A' and 'B'	All wires but the output are green.	All wires but the output are green.
2.0.13	Expression with two repeated inputs.	(A.B)+(A.!B) – Click 'A'	Only 'A' wires are green.	Only 'A' wires are green.
2.0.14	Expression with two repeated inputs.	(A.B)+(A.!B) – Click 'B'	Only the 'B' wires are green.	Only the 'B' wires are green.
2.0.15	Expression with two repeated inputs.	(A.B)+(A.!B) – Click 'A' and 'B'	All wires but the output from the NOT and AND gate are green.	All wires but the output from the NOT and AND gate are green.
2.0.16	4-input diagram	(A.B)^(C.D) – Click 'B'	Only the 'B' wire is green.	Only the 'B' wire is green.
2.0.17	4-input diagram	(A.B)^(C.D) – Click 'C' and 'D'	All wires but the 'A' and 'B' wire and their output are green.	All wires but the 'A' and 'B' wire and their output are green.
2.0.18	4-input diagram.	(A.B)^(C.D) – Click 'A' and 'B' and 'C' and 'D'	All wires but the output are green.	All wires but the output wire are green.
2.0.19	Nested NOT gate example.	!(!A+!B) – Click 'A'	Output is red.	Output is red.
2.0.20	Nested NOT gate example.	!(!A+!B) – Click 'B'	Output is red.	Output is red.
2.0.21	Nested NOT gate example.	!(!A+!B) – click 'A' and 'B'	Output is green.	Output is green.
2.0.22	Repeated '!A' example	(!A.!B)^(!A+B) – Click 'A'	Output is red.	Output is red.
2.0.23	Repeated '!A' example	(!A.!B)^(!A+B) – Click 'B'	Output is green.	Output is green.
2.0.24	Repeated '!A' example	(!A.!B)^(!A+B) – Click 'A' and 'B'	Output is green.	Output is green.
2.0.25	Repeated input example.	(A.(!A+B)) – Click 'A'	Output is red.	Output is red.
2.0.26	Repeated input example.	(A.B)^(!C.A) – Click 'C'	Output is red.	Output is red.
2.0.27	Repeated input example.	(A.B)^(!C.A) – Click 'A'	Output is green.	Output is green
2.0.28	Repeated input example.	(A.B)^(!C.A) – Click 'A' and 'C'	Output is red.	Output is red.
2.0.29	Simple AND gate	(A.B) - Click 'A'	All wires are red	All wires are red
2.0.30	Repeated input example	(!A.!B)^(!A+B) – Click 'B'	Output is red.	Output is red.

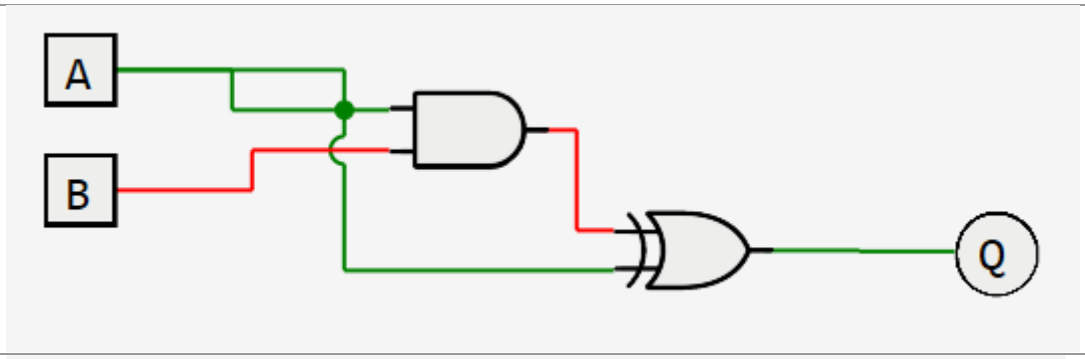
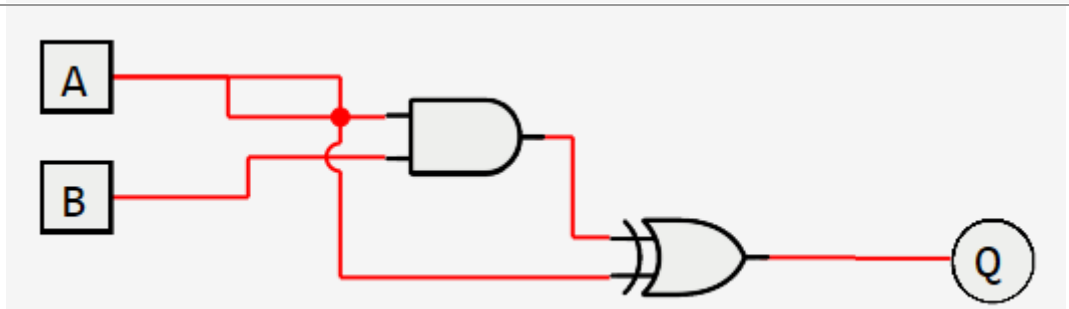
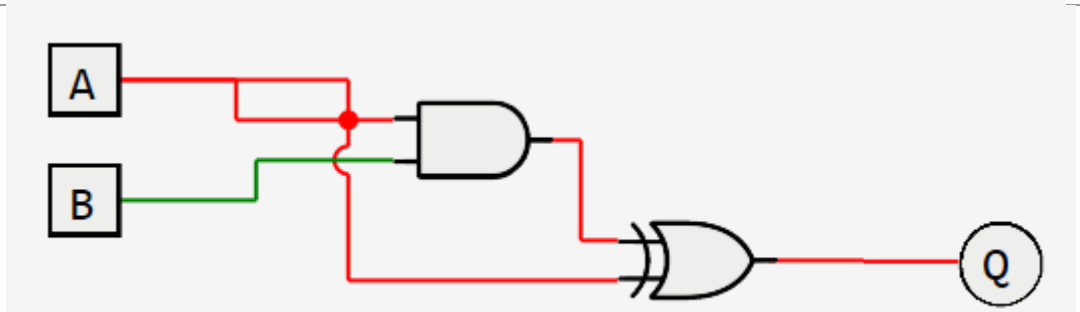
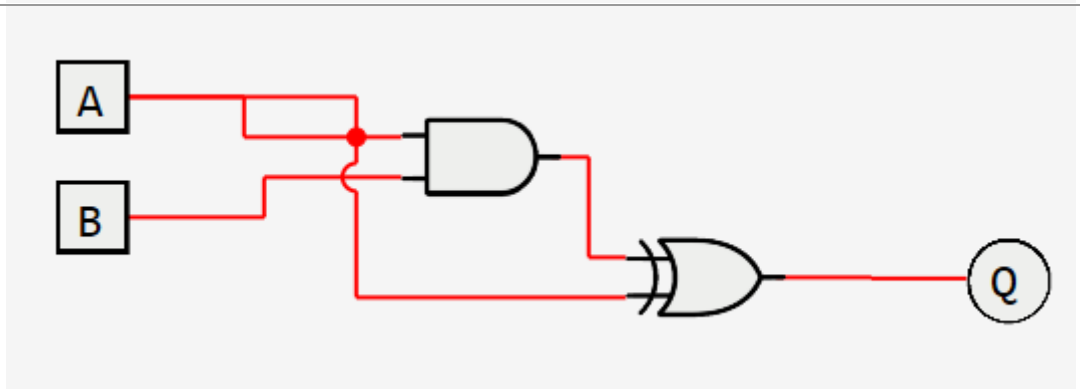
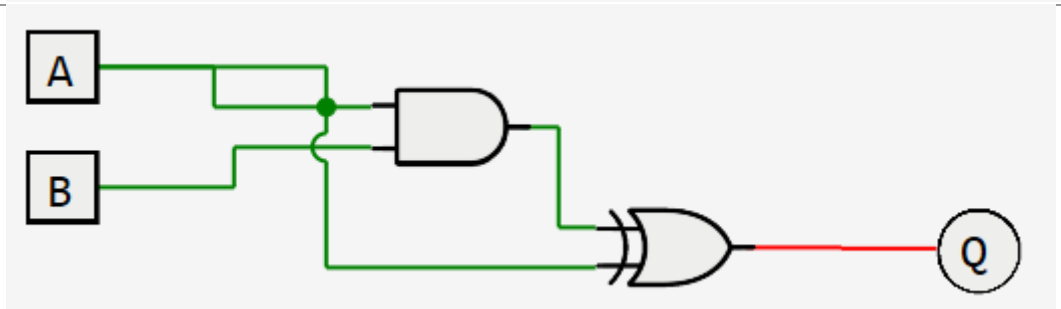
Evidence of tests

In the table below, there will be two images for each test, marked 2.0.0a and 2.0.0b. These correspond to the before and after images of each of the tests once the action defined in the test has taken place.

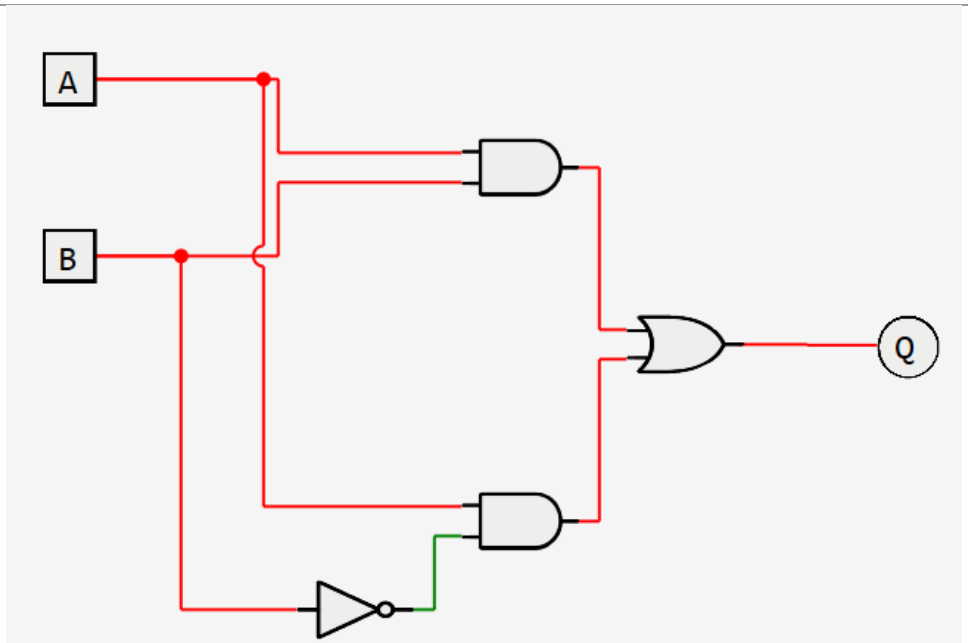
Test ID	Evidence
2.0.0a	
2.0.0b	
2.0.1a	
2.0.1b	
2.0.2a	
2.0.2b	
2.0.3a	

2.0.3b	
2.0.4a	
2.0.4b	
2.0.5a	
2.0.5b	
2.0.6a	
2.0.6b	

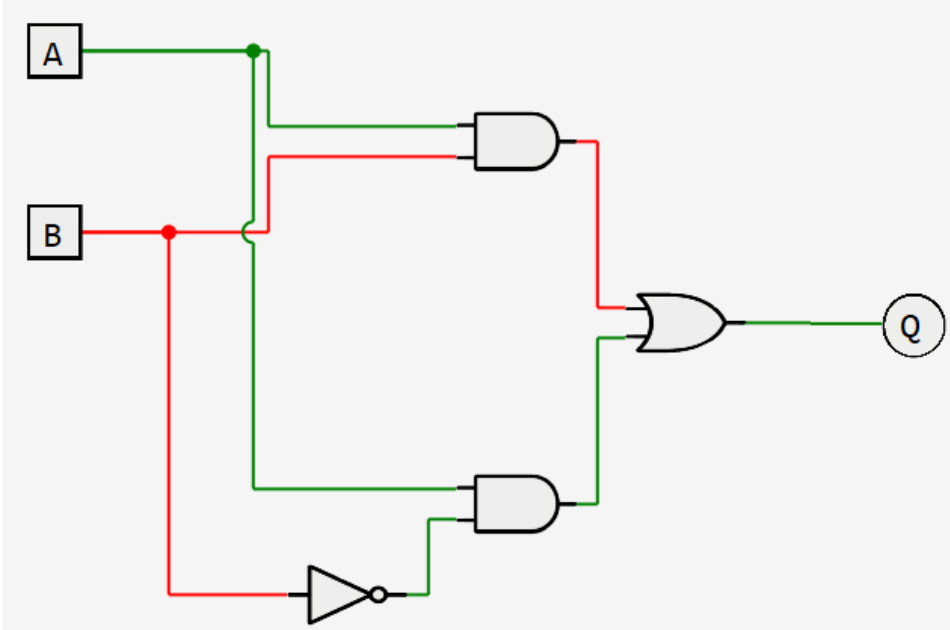
2.0.7a	
2.0.7b	
2.0.8a	
2.0.8b	
2.0.9a	
2.0.9b	
2.0.10a	

2.0.10b	
2.0.11a	
2.0.11b	
2.0.12a	
2.0.12b	

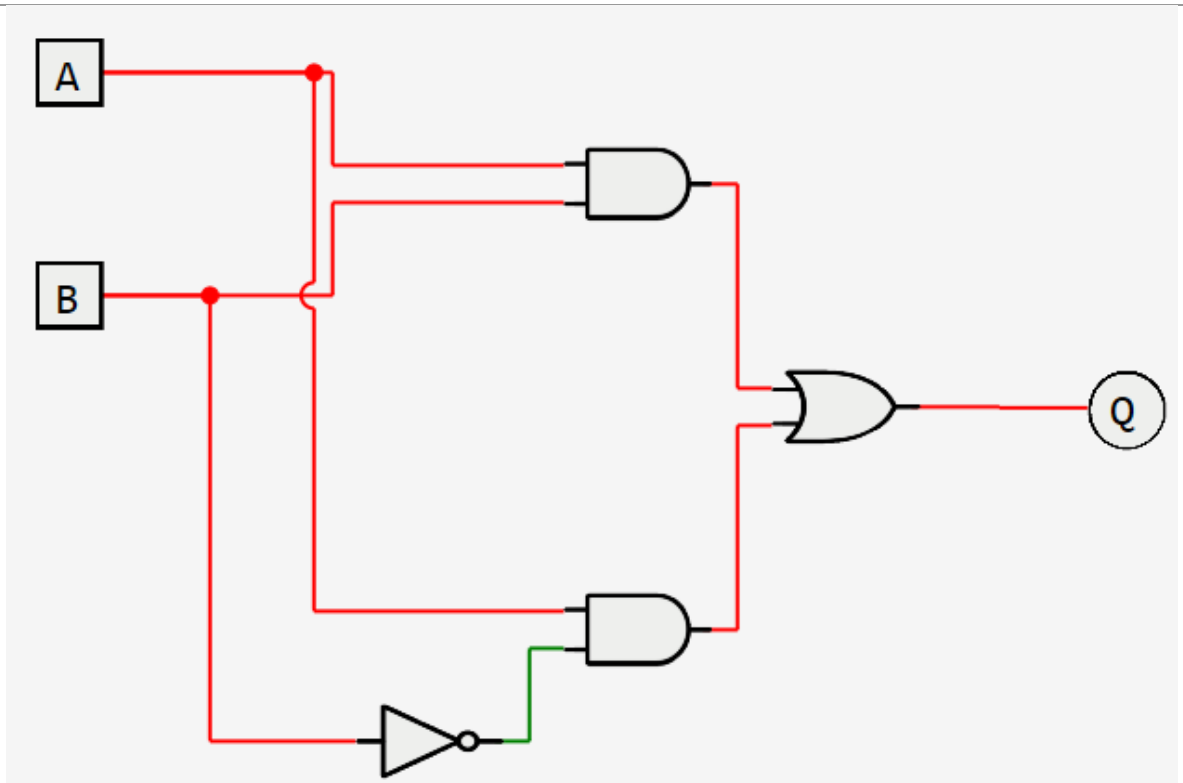
2.0.13a



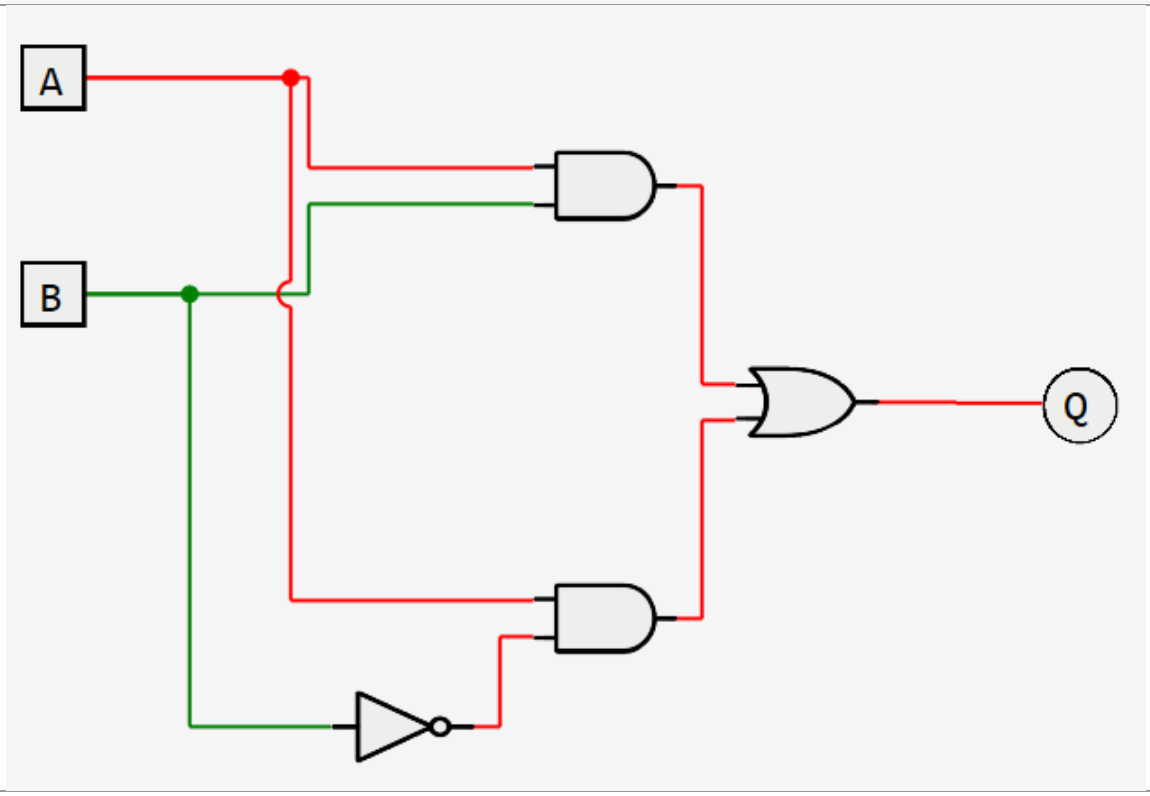
2.0.13b



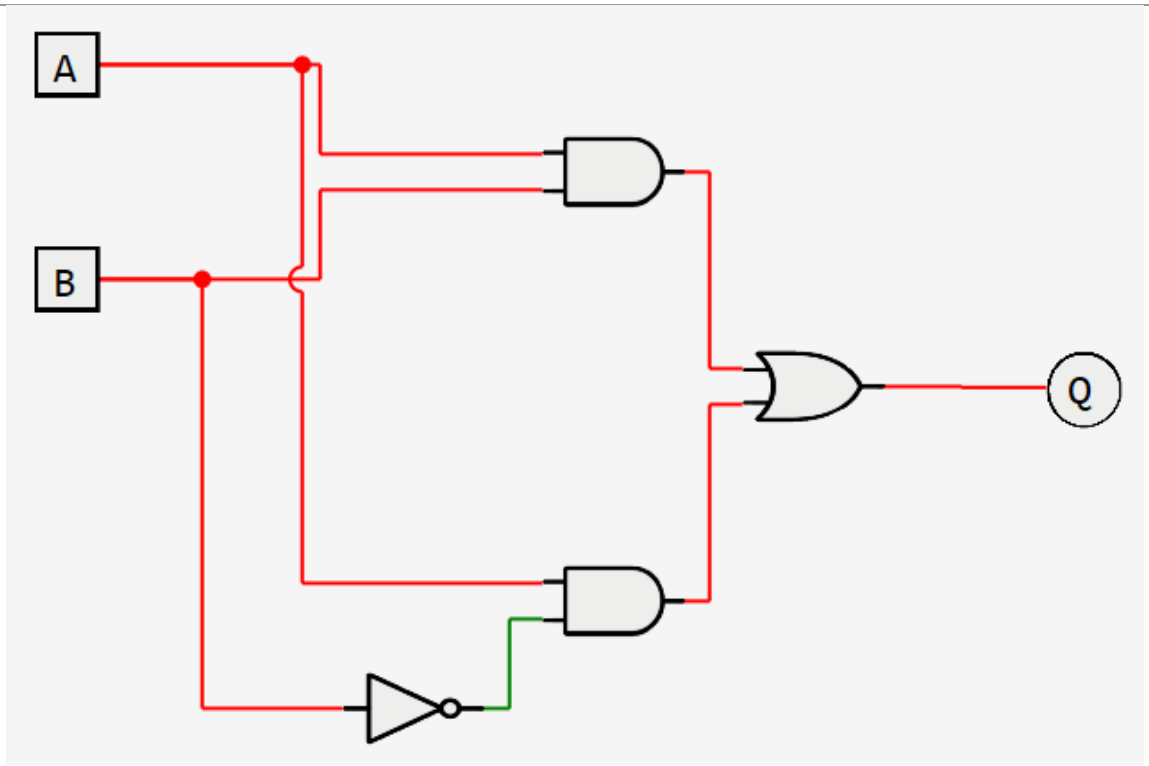
2.0.14a



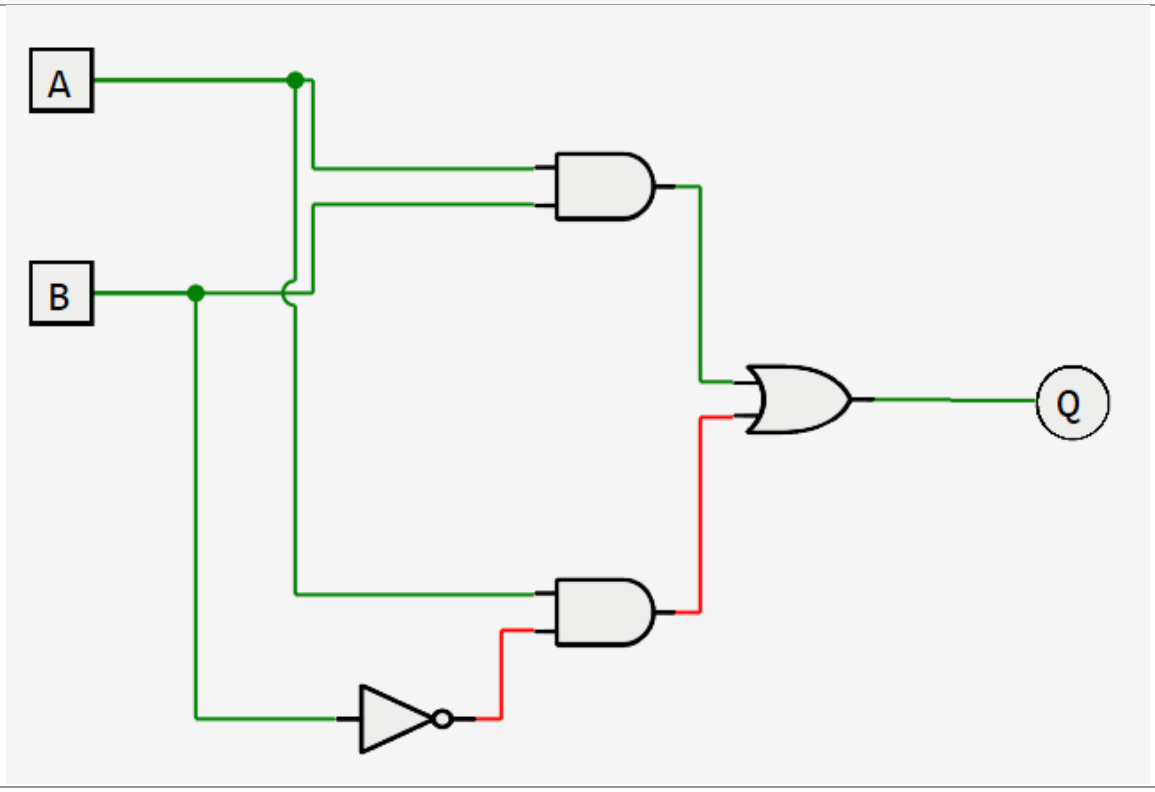
2.0.14b

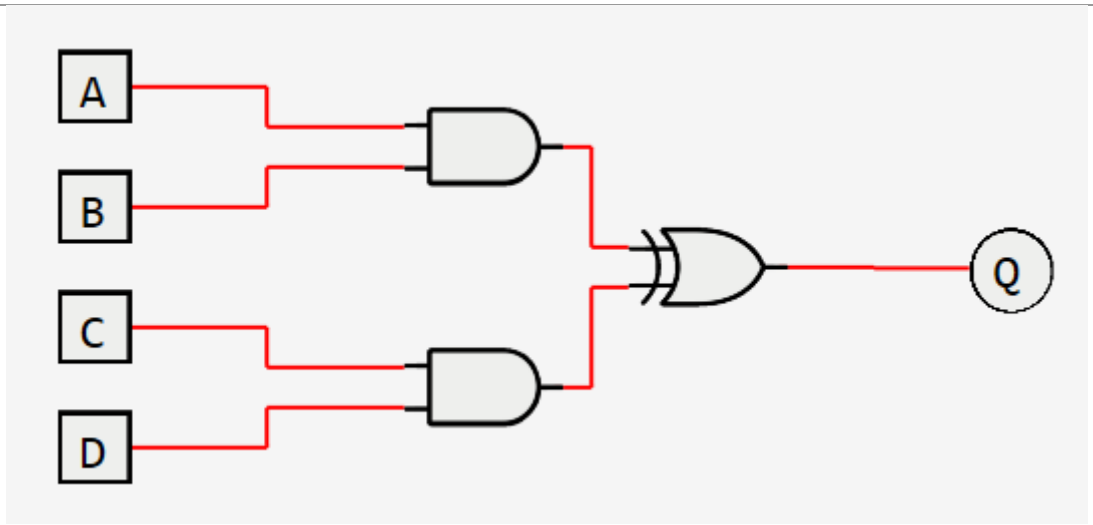
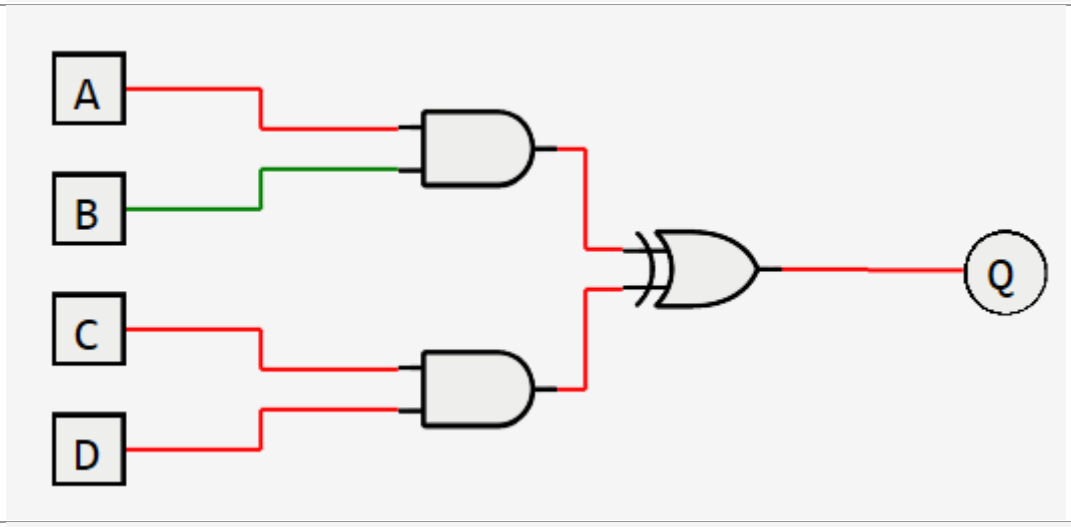
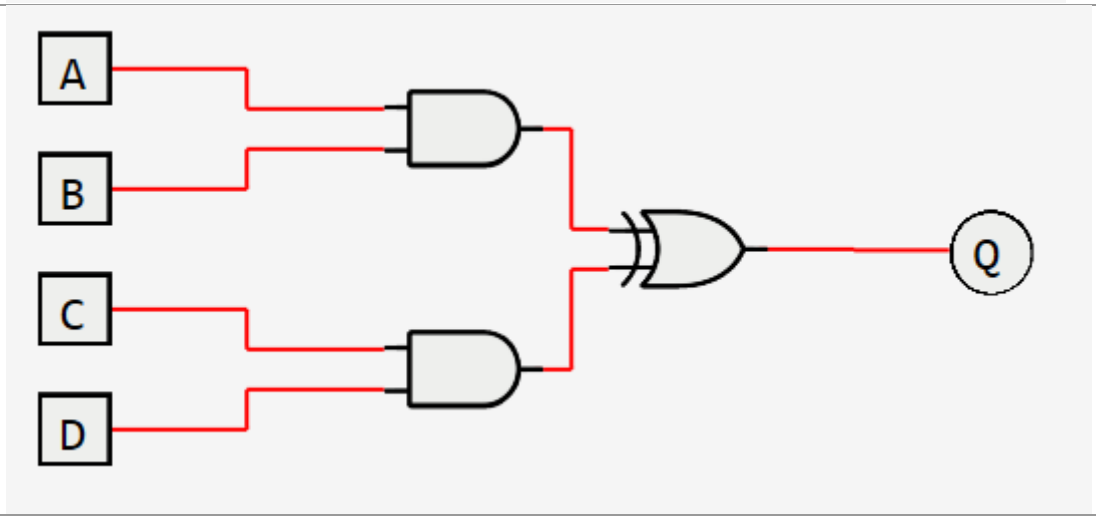


2.0.15a

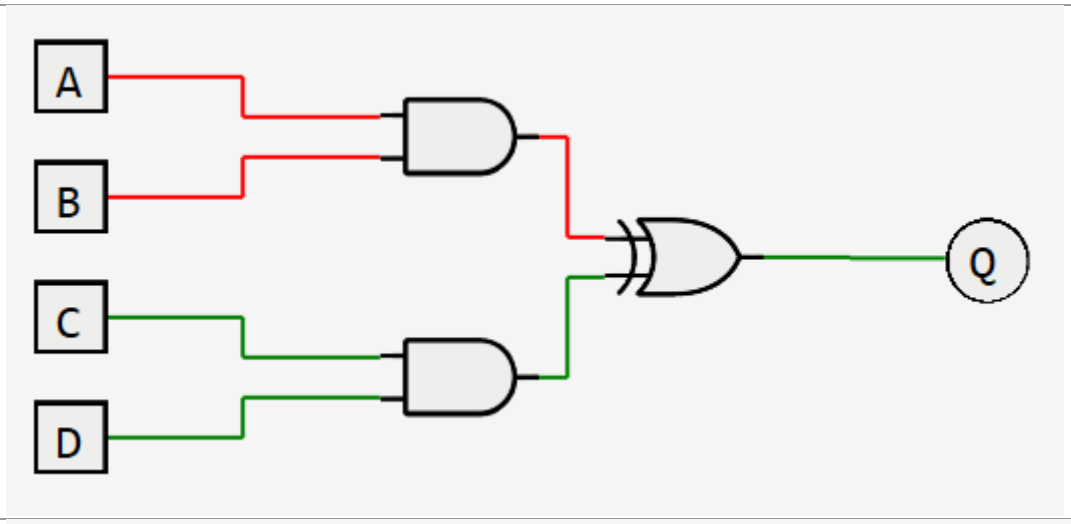


2.0.15b

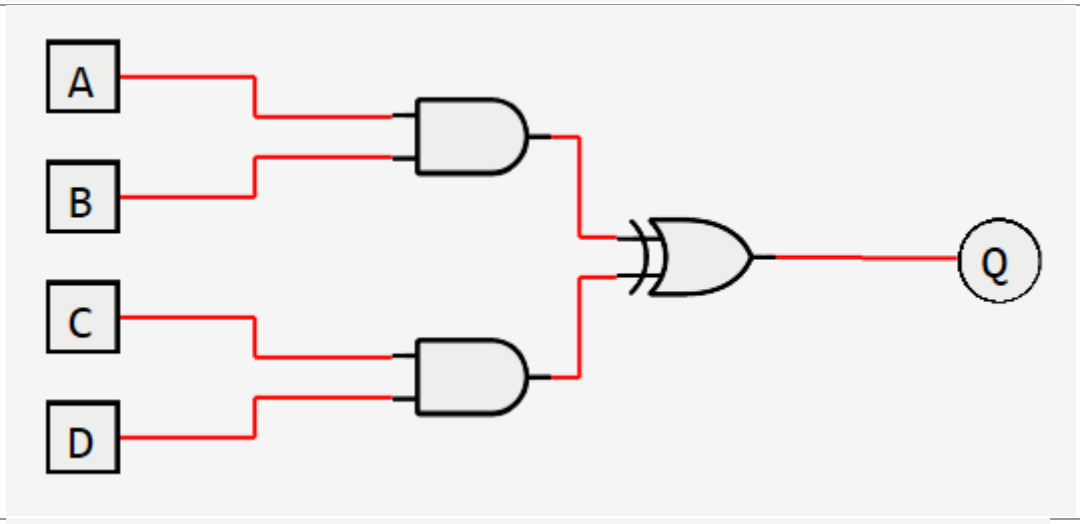


2.0.16a	
2.0.16b	
2.0.17a	

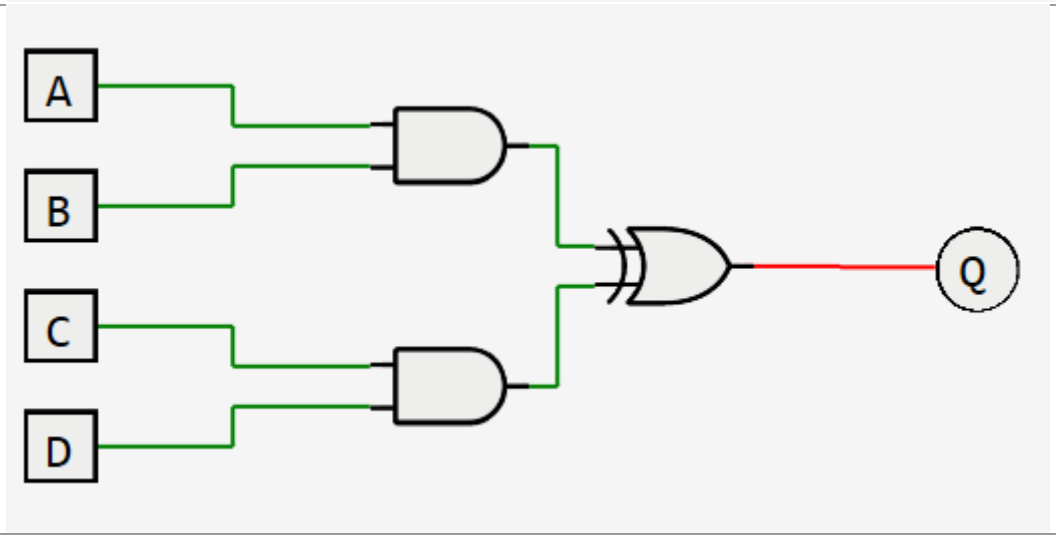
2.0.17b

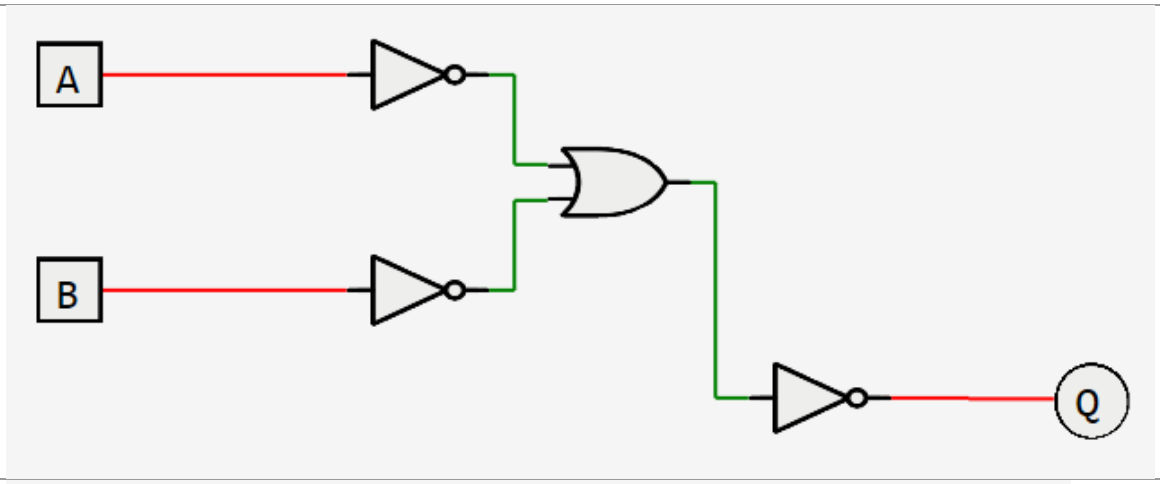
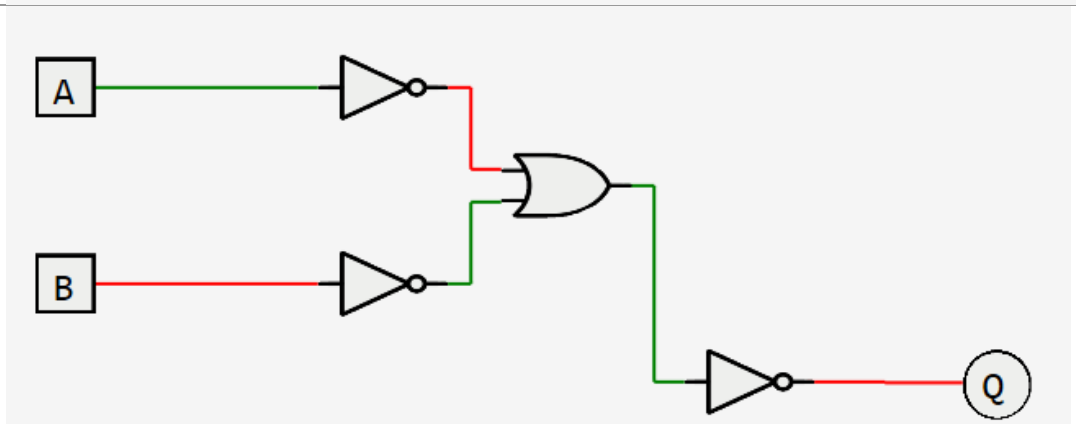
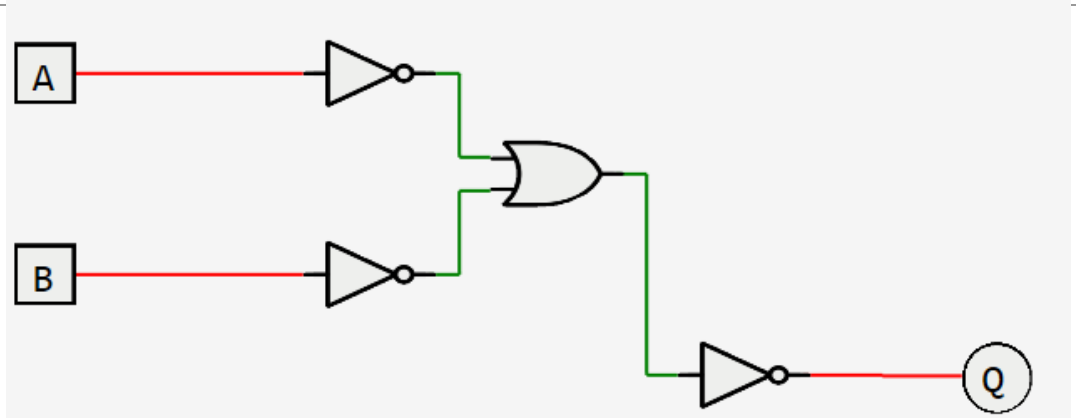
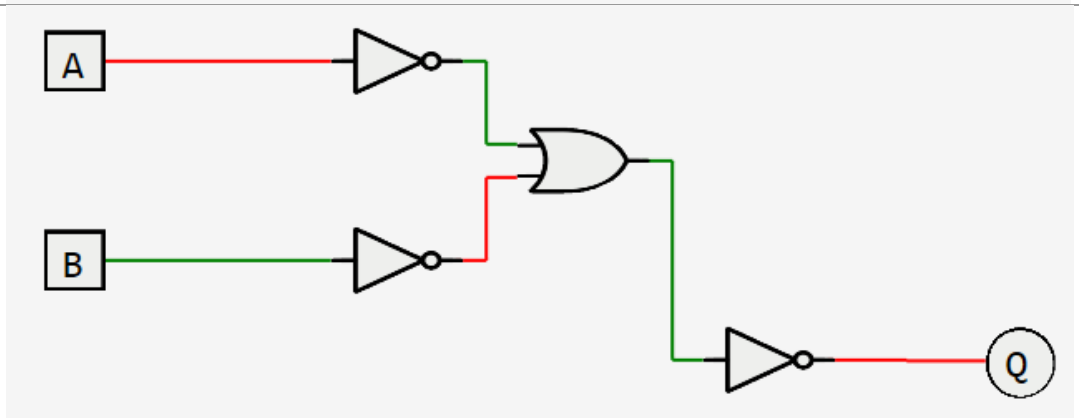


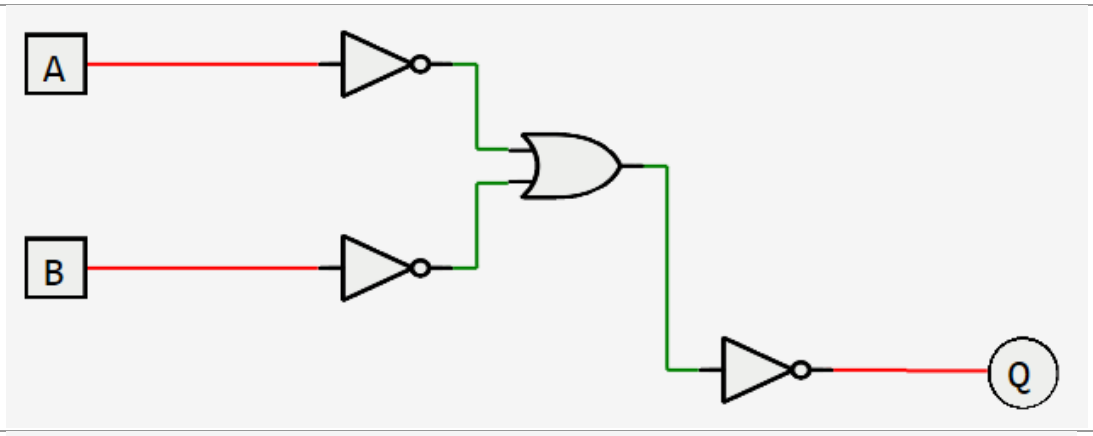
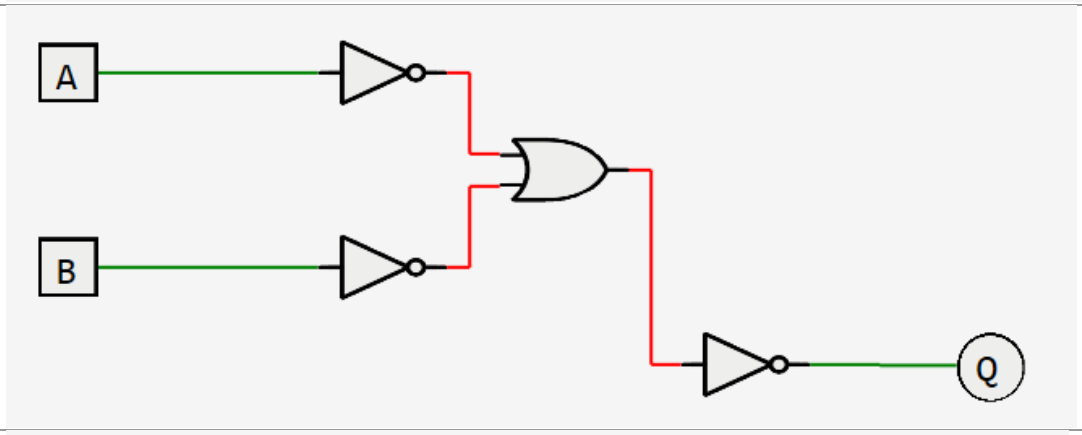
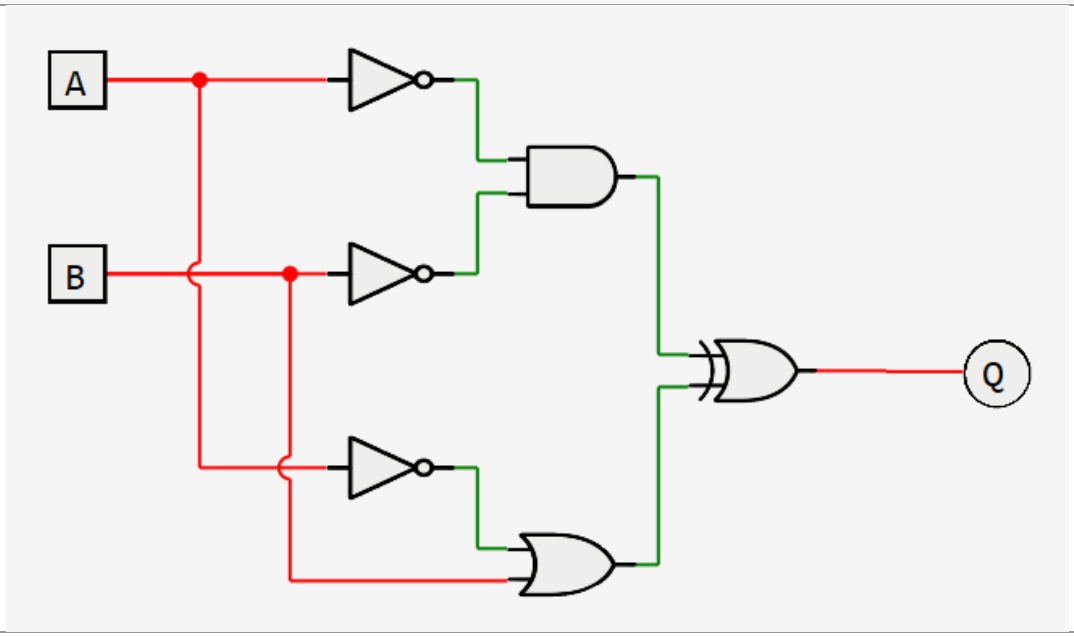
2.0.18a



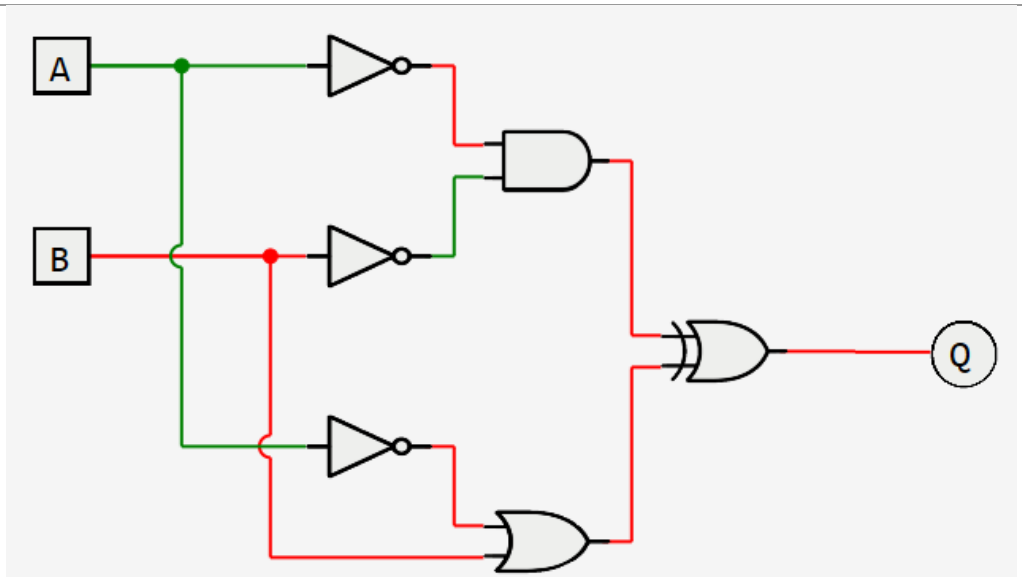
2.0.18b



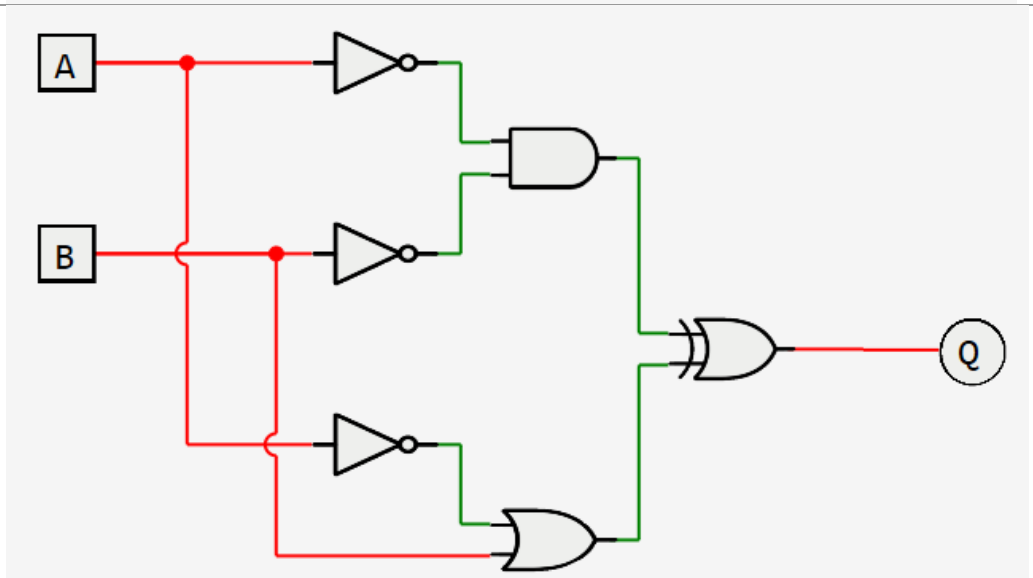
2.0.19a	
2.0.19b	
2.0.20a	
2.0.20b	

2.0.21a	
2.0.21b	
2.0.22a	

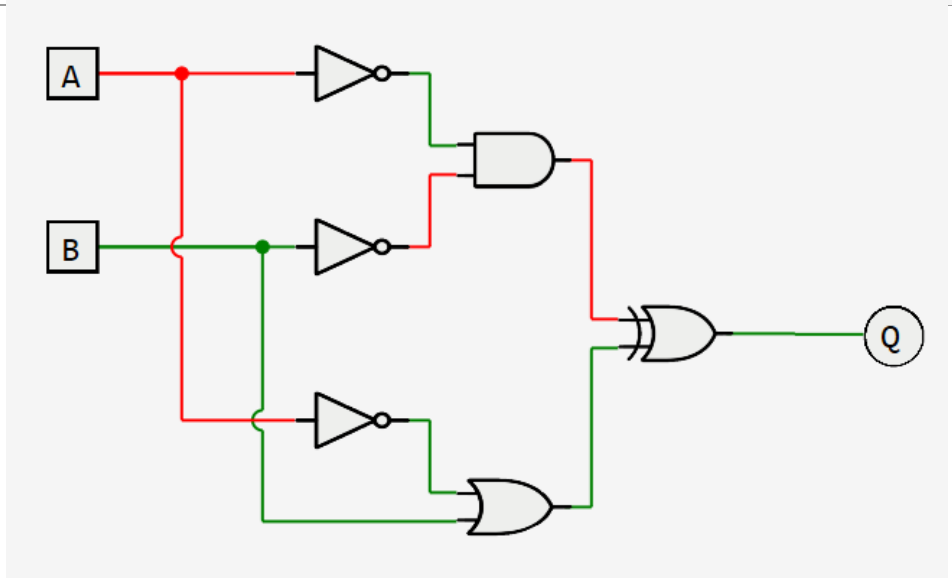
2.0.22b



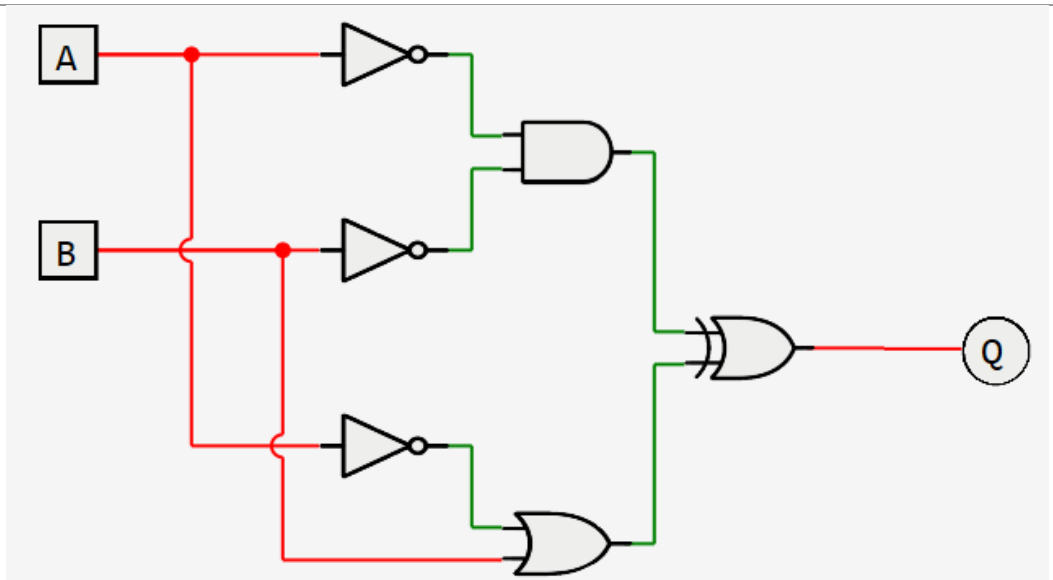
2.0.23a



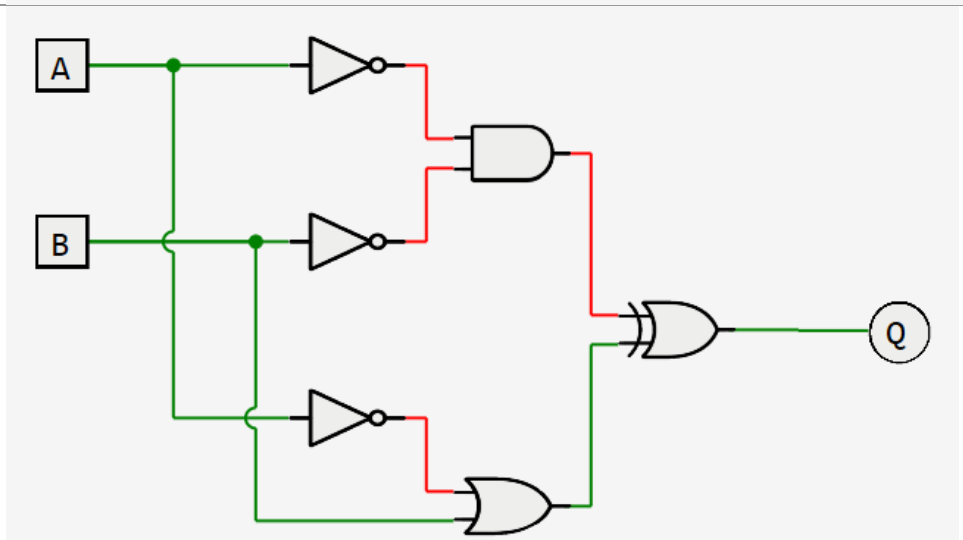
2.0.23b



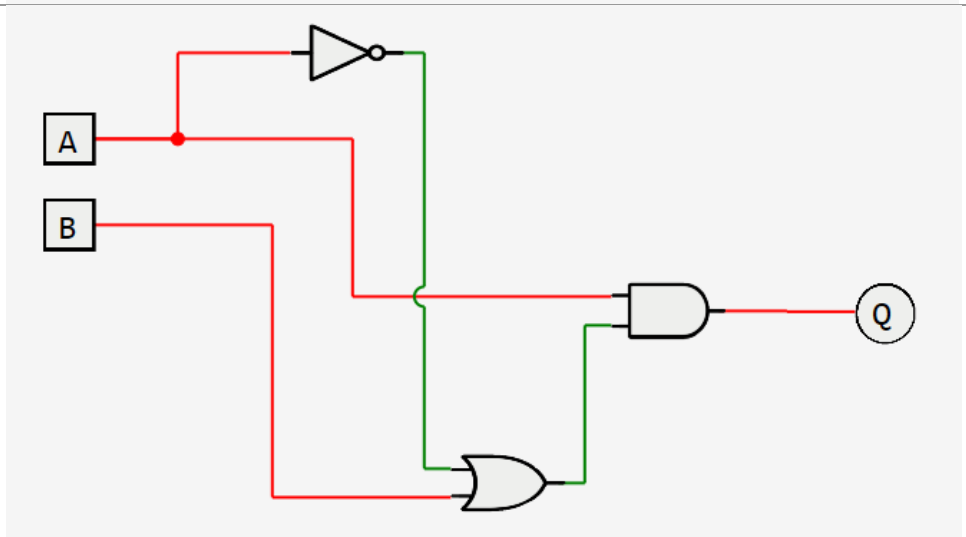
2.0.24a

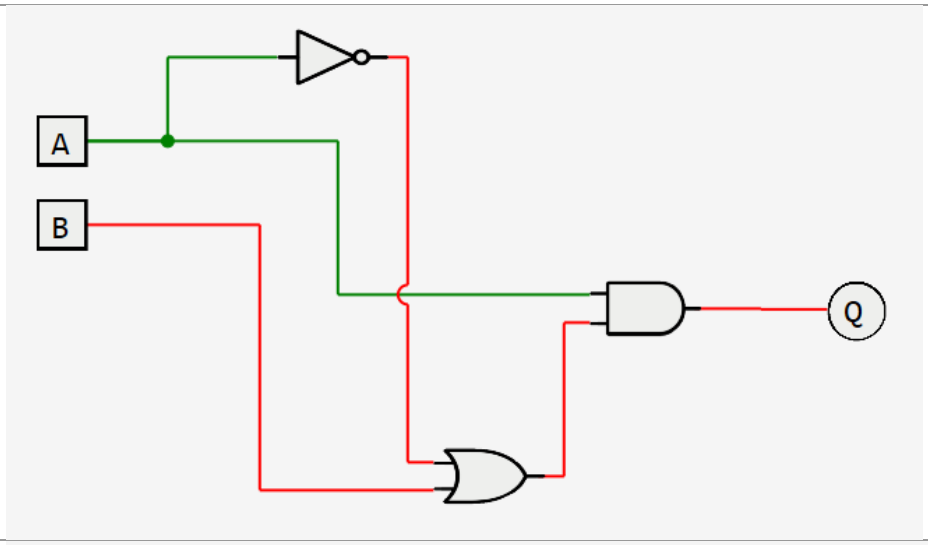
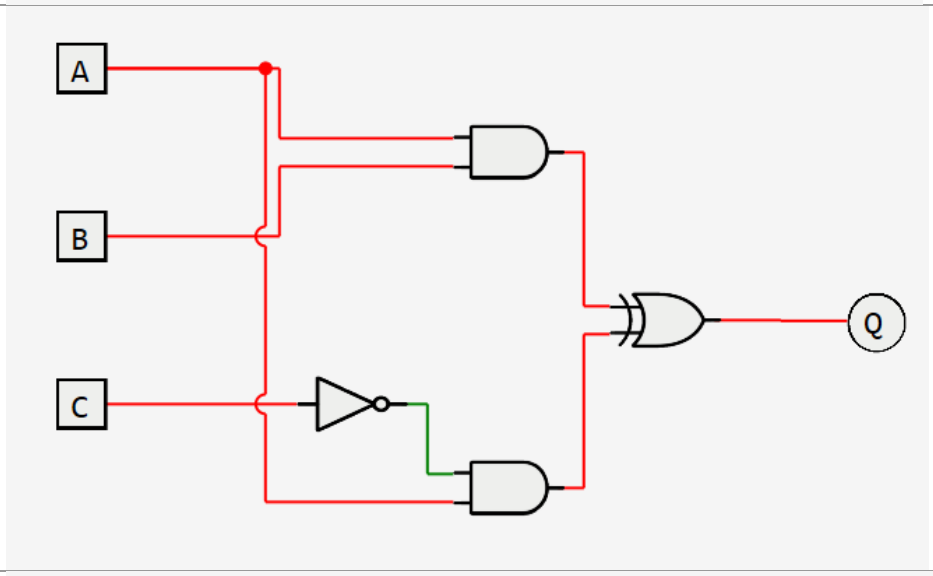
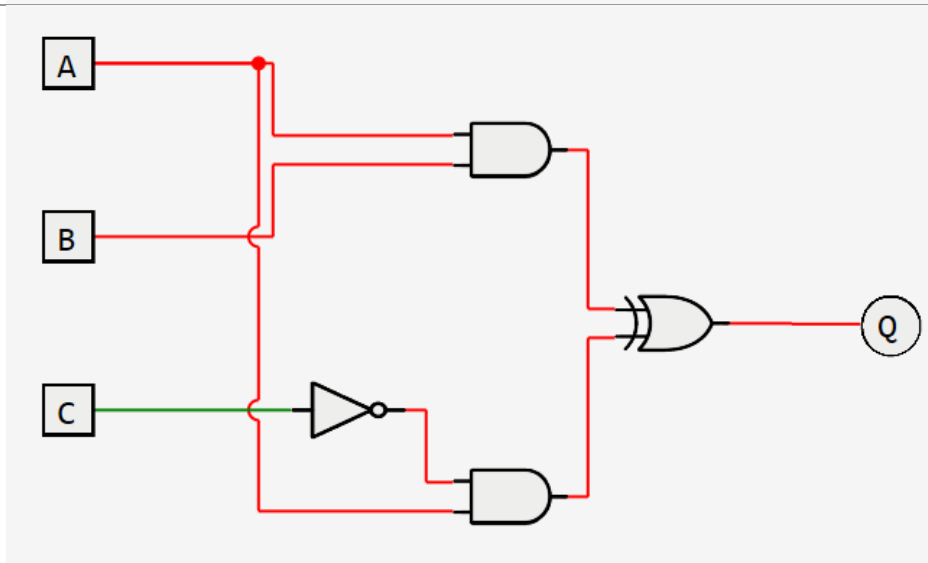


2.0.24b

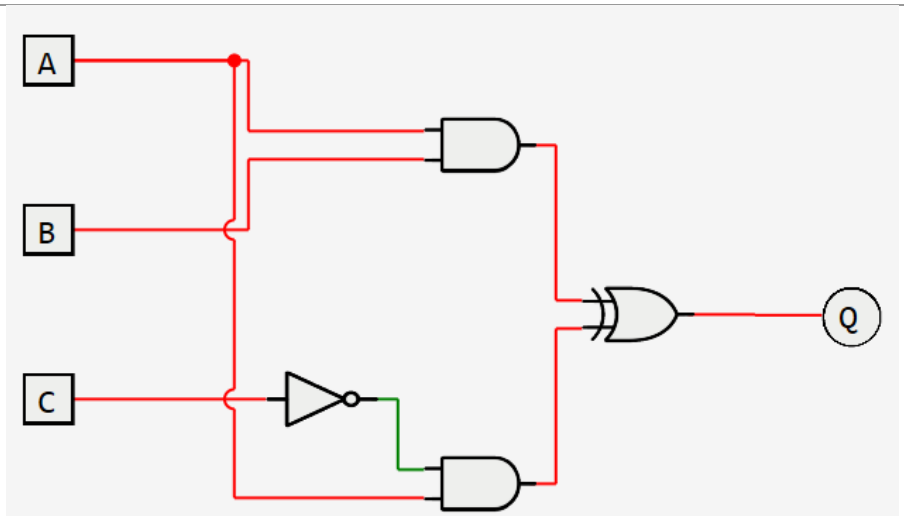


2.0.25a

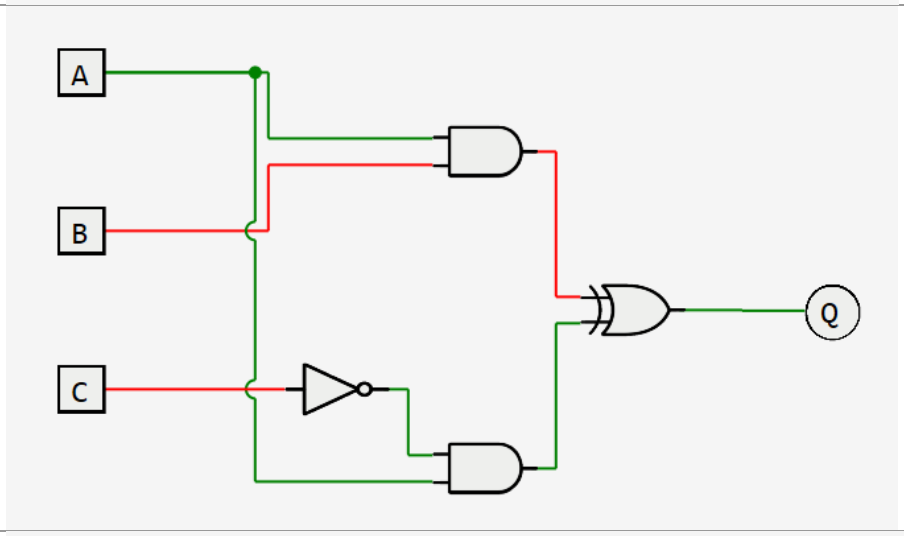


2.0.25b	
2.0.26a	
2.0.26b	

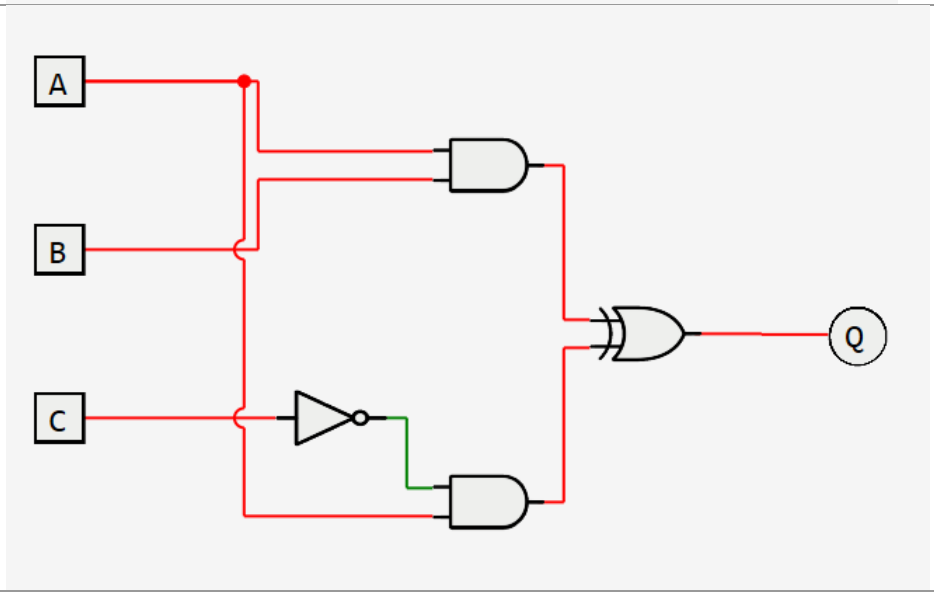
2.0.27a

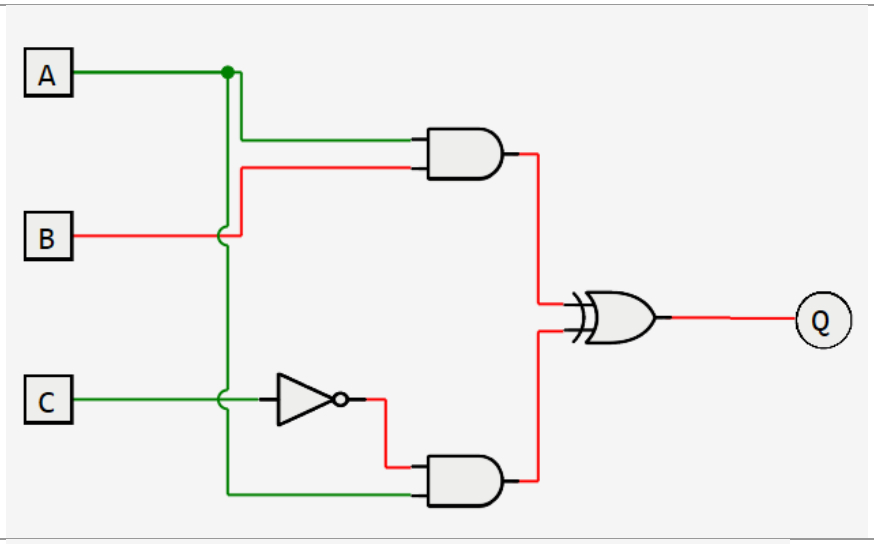
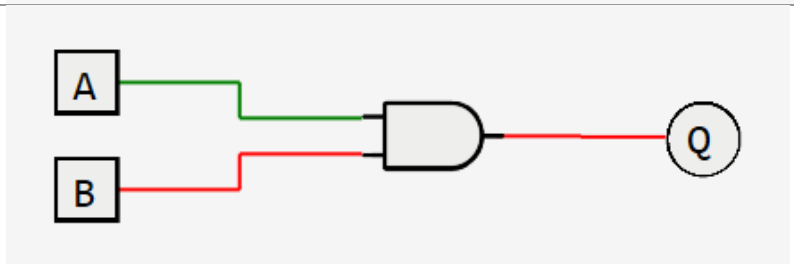
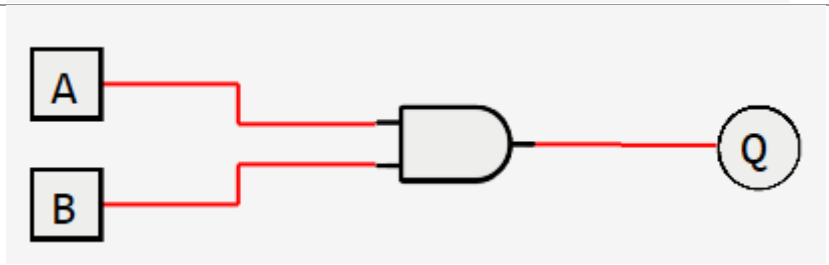
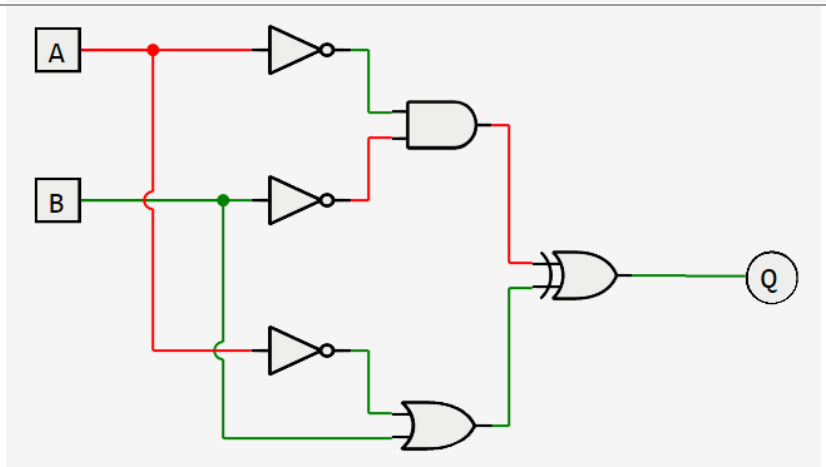


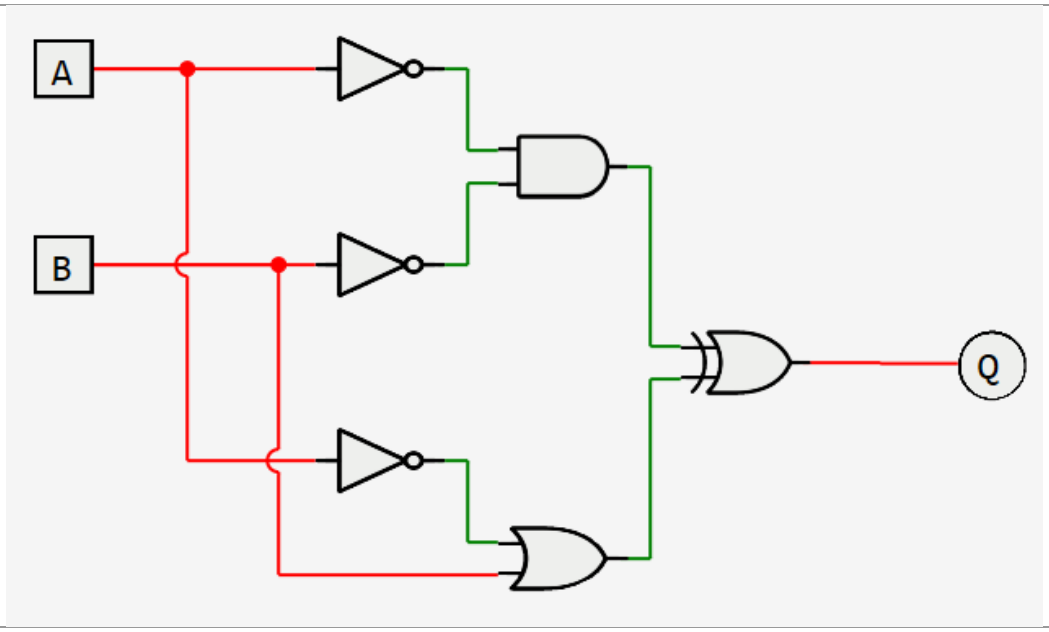
2.0.27b



2.0.28a



2.0.28b	
2.0.29a	
2.0.29b	
2.0.30a	

2.0.30b	

Truth table generation

The following tests cover truth table generation within the program. In this section, to main types of test will be conducted, one from a drawn diagram and another generated from an expression.

Test ID	Description	Test data	Expected Outcome	Outcome
3.0.0	Basic AND gate	(A.B)	Truth table generated	Truth table generated.
3.0.1	Basic OR gate	(A+B)	Truth table generated	Truth table generated
3.0.2	Basic XOR gate	(A^B)	Truth table generated	Truth table generated
3.0.3	Basic NOT gate	(!A)	Truth table generated	Truth table generated.
3.0.4	Multi staged table	((A.B)^C)	Truth table generated	Truth table generated.
3.0.5	Repeated input table	((A.B)+A)	Truth table generated	Truth table generated.
3.0.6	Duplicate subexpressions	(A.B)+((A.B)^C)	Truth table generated	Truth table generated
3.0.7	Inversing De Morgan's law	!(!(A+B))	Truth table generated	Truth table generated
3.0.8	More complex repeated inputs.	(!A.!B)^(A+A)	Truth table generated	Truth table generated
3.0.9	Boolean identity example	(!A.!B)^(!A+A)	Truth table generated	Truth table generated
3.0.10	Boolean identity example	(A+1)	Truth table generated	Truth table generated
3.0.11	Boolean identity example	(A+0)	Truth table generated	Truth table generated
3.0.12	Boolean identity example	(A.1)	Truth table generated	Truth table generated
3.0.13	Boolean identity example	(A.0)	Truth table generated	Truth table generated
3.0.14	Boolean identity example	(A+!A)	Truth table generated	Truth table generated
3.0.15	Boolean identity example	!(!A)	Truth table generated	Truth table generated
3.0.16	Null expression		Error message shown	Error message shown

Evidence of tests

The following table contains all of the evidence of the tests conducted on truth table generation. In the table there will be two images for each test, marked 3.0.0a and 3.0.0b. The images marked with 'a' are generated by the program and the ones marked 'b' are truth tables generated online (using the website EMathHelp), to validate the data within the tables generated by the program.

Test ID	Evidence

3.0.0a

A	B	A.B
0	0	0
0	1	0
1	0	0
1	1	1

3.0.0b

a	b	$a \wedge b$
1	1	1
1	0	0
0	1	0
0	0	0

3.0.1a

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

3.0.1b

a	b	$a \vee b$
1	1	1
1	0	1
0	1	1
0	0	0

3.0.2a

A	B	A^B
0	0	0
0	1	1
1	0	1
1	1	0

3.0.2b

a	b	$a \oplus b$
1	1	0
1	0	1
0	1	1
0	0	0

3.0.3a

A	!A
0	1
1	0

3.0.3b

a	$\neg a$
1	0
0	1

3.0.4a

A	B	C	A.B	(A.B)^C
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	0

3.0.4b

a	b	c	$a \wedge b$	$(a \wedge b) \oplus c$
1	1	1	1	0
1	1	0	1	1
1	0	1	0	1
1	0	0	0	0
0	1	1	0	1
0	1	0	0	0
0	0	1	0	1
0	0	0	0	0

3.0.5a

A	B	A.B	(A.B)+A
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	1

3.0.5b

a	b	$a \wedge b$	$(a \wedge b) \vee a$
1	1	1	1
1	0	0	1
0	1	0	0
0	0	0	0

3.0.6a

A	B	C	A.B	(A.B)^C	(A.B)+((A.B)^C)
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	1	1
1	1	1	1	0	1

3.0.6b

a	b	c	$a \wedge b$	$(a \wedge b) \vee (a \wedge b)$	$((a \wedge b) \vee (a \wedge b)) \oplus c$
1	1	1	1	1	0
1	1	0	1	1	1
1	0	1	0	0	1
1	0	0	0	0	0
0	1	1	0	0	1
0	1	0	0	0	0
0	0	1	0	0	1
0	0	0	0	0	0

3.0.7a

A	B	A+B	!(A+B)	!(!(A+B))
0	0	0	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	0	1

3.0.7b

a	b	$a \vee b$	$\neg (a \vee b)$	$\neg (\neg (a \vee b))$
1	1	1	0	1
1	0	1	0	1
0	1	1	0	1
0	0	0	1	0

3.0.8a

A	B	!A	!B	A+A	(!A).(!B)	((!A).(!B))^(A+A)
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	1	0	1
1	1	0	0	1	0	1

3.0.8b

a	b	$\neg a$	$\neg b$	$a \vee a$	$\neg a \wedge \neg b$	$(\neg a \wedge \neg b) \oplus (a \vee a)$
1	1	0	0	1	0	1
1	0	0	1	1	0	1
0	1	1	0	0	0	0
0	0	1	1	0	1	1

3.0.9a

A	B	!A	!B	(!A)+A	(!A).(!B)	((!A).(!B))^((!A)+A)
0	0	1	1	1	1	0
0	1	1	0	1	0	1
1	0	0	1	1	0	1
1	1	0	0	1	0	1

3.0.9b

a	b	$\neg a$	$\neg b$	$\neg a \vee a$	$\neg a \wedge \neg b$	$(\neg a \wedge \neg b) \oplus (\neg a \vee a)$
1	1	0	0	1	0	1
1	0	0	1	1	0	1
0	1	1	0	1	0	1
0	0	1	1	1	1	0

3.0.10a

1	A	A+1
1	0	1
1	0	1
1	1	1
1	1	1

3.0.10b

a	T	$a \vee T$
1	1	1
0	1	1

3.0.11a

0	A	A+0
0	0	0
0	0	0
0	1	1
0	1	1

3.0.11b

a	F	$a \vee F$
1	0	1
0	0	0

3.0.12a

1	A	A.1
1	0	0
1	0	0
1	1	1
1	1	1

3.0.12b

a	T	$a \wedge T$
1	1	1
0	1	0

3.0.13a

0	A	A.0
0	0	0
0	0	0
0	1	0
0	1	0

3.0.13b

a	F	$a \wedge F$
1	0	0
0	0	0

3.0.14a

A	!A	A+(!A)
0	1	1
1	0	1

3.0.14b

a	$\neg a$	$a \vee \neg a$
1	0	1
0	1	1

3.0.15a

A	!A	!(!A)
0	1	0
1	0	1

3.0.15b

a	$\neg a$	$\neg (\neg a)$
1	0	1
0	1	0

3.0.16

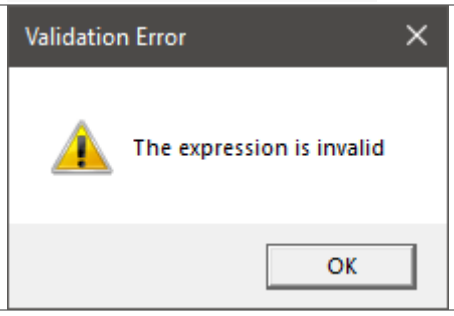


Diagram and expression minimisation

The following tests cover minimising diagrams and expressions with the Quine-McCluskey algorithm and Petrick's method. All of the expressions being tested within this section have been taken from the AQA teaching guide for Boolean Algebra.

It must be noted that testing Petrick's method is very difficult as a significant majority of Boolean expressions are simplified simply using the Quine-McCluskey Algorithm, as such only one test will be conducted on Petrick's method.

Test ID	Description	Test data	Expected Outcome	Outcome
4.0.0	Application of Boolean Identities.	A+A	A	A
4.0.1	Application of Boolean Identities.	A.B+A.B	A.B	(A.B)
4.0.2	Application of Boolean identities.	(A.B)+(A.B)+C	(A.B)+C	(C+(A.B))
4.0.3	Application of Boolean identities.	!(!(A+A)	A	A
4.0.4	Application of Boolean identities.	(A.!B)+(C.D)+(C.D)	(A.!B)+(C.D)	((C.D)+(A.!B))
4.0.5	Absorption law	A+(A.B)	A	Invalid Operation Exception: Stack Empty

Test comment: The program is incorrectly adding AND gates when formatting the minimised expression. This in turn is then passed to the Boolean converter which then tries to convert an invalid expression for display.

Solution: The program is immediately removing the dashes when converting the essential prime implicants into their expression form. This simplifies indexing, simplifying the process for output. The new method is below:

```
        private string ConvertImplicantToExpression(string epi)
        {
            string tmp;
            // Removing the dashes as they are not a part of the output.
            var removeDashes = new Regex("-");
            epi = removeDashes.Replace(epi, "");
            List<char> terms = epi.ToList();
            // Each term must be separated by an AND gate as the epi is a product.
            tmp = string.Join(".", terms);
            char input;
            for(var i = 0; i < tmp.Length; i += 2)
            {
                input = (char)(i + 65);
                // If the bit is a one then the output must be in the expression.
                if (tmp[i] == '1')
                {
                    tmp = tmp.Remove(i, 1);
                    tmp = tmp.Insert(i, input.ToString());
                }
                // If the bit is a zero then the complement is added to the expression => !A.
                else if (tmp[i] == '0')
                {
                    tmp = tmp.Remove(i, 1);
                    tmp = tmp.Insert(i, $"!{input}");
                }
            }
            // Add brackets to preserve pretty formatting.
            return $"({tmp})";
        }
```

`4.0.5	Absorption law	A+(A.B)	A	A
4.0.6	Distributive law	A+((A.B)+C)	A+C	(C+A)
4.0.7	De Morgan's law and the absorption law.	(!(!B+!A).(!A))+B	B	B
4.0.8	Application of Boolean identities.	(A^1)	!A	!A
4.0.9	Application of Boolean identities.	(A^0)	A	A
4.0.10	De Morgan's law	!(!A+!B)	A.B	(A.B)
4.0.11	Application of Boolean identities.	(A.1)	A	A
4.0.12	A mixture of many laws involved.	(!A.!B)^(A+A)	A+!B	(A+!B)
4.0.13	Boolean identity	A+1	1	Invalid Operation Exception: Stack empty

Test comment: The program always assumes that a prime implicant will be present in the minimised results. This means that there is no way of outputting expressions that minimise to constants.

Solution: Check for cases that result in constants within the minimise expression method. This gives the following function.

```
        public void MinimiseExpression(string expression)
        {
            // The input combinations that result in the expression evaluating to one.
            List<string> minterms = GetMinterms(expression);
            List<string> primeImplicants = GetPrimeImplicants(minterms);
            var PIchart = new Dictionary<string, string>();
            ConvertImplicantsIntoRegex(PIchart, primeImplicants);
            // Creating the coverages for each of the prime implicants using regex.
            SetRegexPatterns(PIchart, minterms);
            PIchart = ReplaceDashesFromRegex(PIchart);
            List<string> PIs = GetEssentialPrimeImplicants(PIchart, minterms);
            string coveredString = GetCoveredString(PIs, PIchart);
            // Filtering out any expression that evaluate to 0 or 1.
            if (PIs[0] == "--" || PIs[0] == "0")
            {
                if (PIs[0] == "--")
                {
                    minimisedExpression = "1";
                }
                else
                {
                    minimisedExpression = "0";
                }
            }
            else
            {
                // If a 0 remains in the covered string, then the essential prime implicants
                // do not cover all of the minterms and so petrick's method must be used.
                if (coveredString.Contains('0'))
                {
                    minimisedExpression = DoPetriksMethod(PIchart, PIs, primeImplicants,
minterms);
                }
                else
                {
                    minimisedExpression = ConvertEPIsToExpression(PIs);
                }
```

```
            }
        }
```

`4.0.13	Boolean identity	A+1	1	1
4.0.14	Boolean identity	A.0	0	Argument Out Of Range Exception

Test comment: When outputting expressions that minimise to constants, the checks are not expansive enough as they assume that a prime implicant will be found within the expression. This must be included when checking for constants.

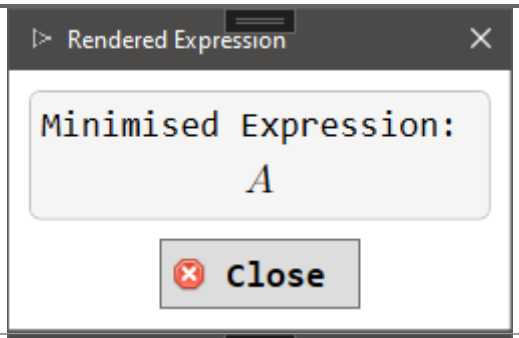
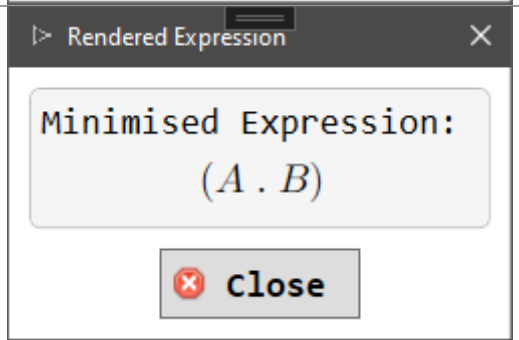
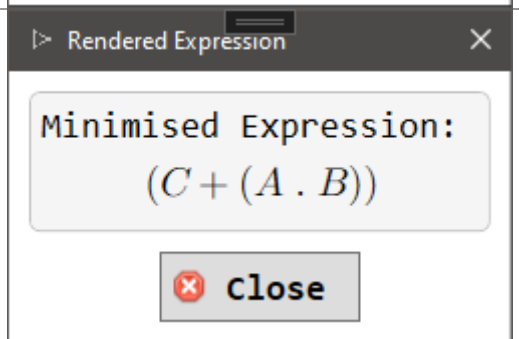
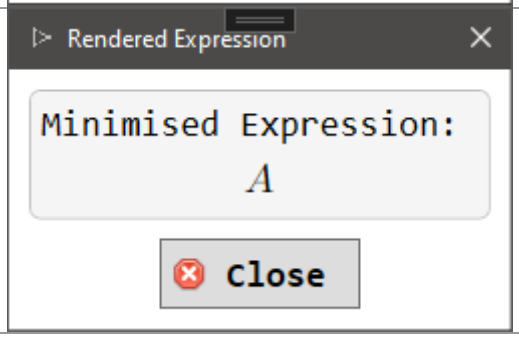
Solution: Added a check for when no prime implicants remain as the expression is being minimised. This gives the following MinimiseExpression method.

```
        public void MinimiseExpression(string expression)
        {
            // The input combinations that result in the expression evaluating to one.
            List<string> minterms = GetMinterms(expression);
            List<string> primeImplicants = GetPrimeImplicants(minterms);
            var PIchart = new Dictionary<string, string>();
            ConvertImplicantsIntoRegex(PIchart, primeImplicants);
            // Creating the coverages for each of the prime implicants using regex.
            SetRegexPatterns(PIchart, minterms);
            PIchart = ReplaceDashesFromRegex(PIchart);
            List<string> PIs = GetEssentialPrimeImplicants(PIchart, minterms);
            string coveredString = GetCoveredString(PIs, PIchart);
            // Filtering out any expression that evaluate to 0 or 1.
            if (PIs.Count == 0)
            {
                minimisedExpression = "0";
            }
            else if (PIs[0] == "--" || PIs[0] == "0")
            {
                if (PIs[0] == "--")
                {
                    minimisedExpression = "1";
                }
                else
                {
                    minimisedExpression = "0";
                }
            }
            else
            {
                // If a 0 remains in the covered string, then the essential prime implicants
                // do not cover all of the minterms and so petrick's method must be used.
                if (coveredString.Contains('0'))
                {
                    minimisedExpression = DoPetriksMethod(PIchart, PIs, primeImplicants,
minterms);
                }
                else
                {
                    minimisedExpression = ConvertEPIsToExpression(PIs);
                }
            }
        }
```

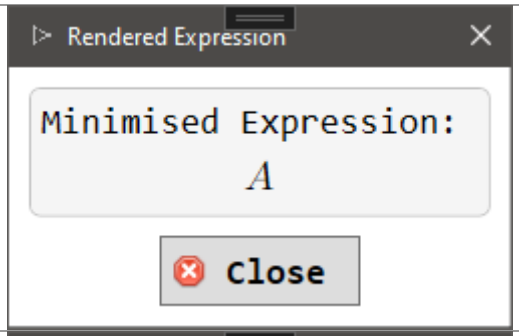
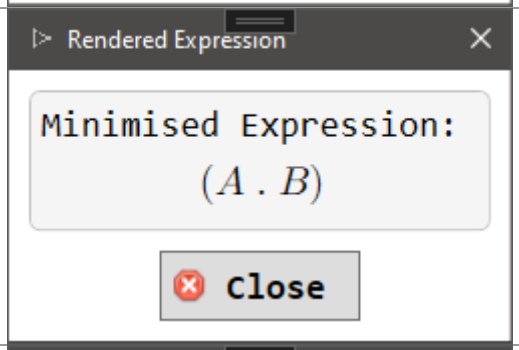
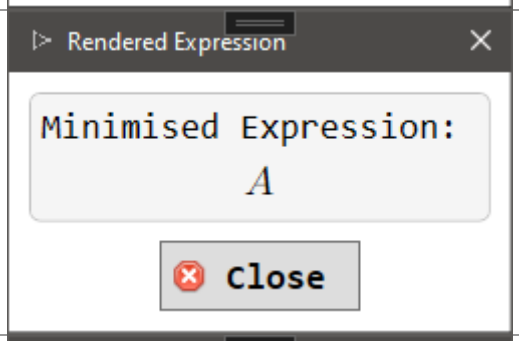
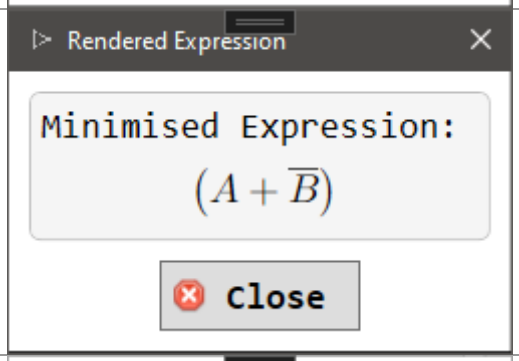
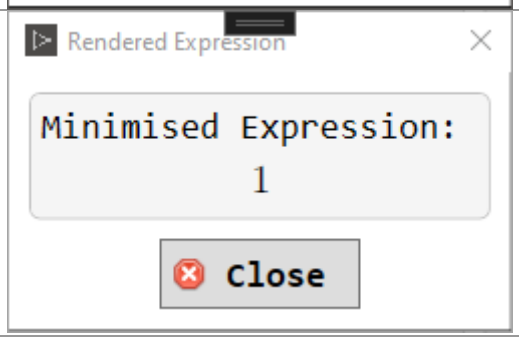
`4.0.14	Boolean identity	A.0	0	0
4.0.15	Boolean identity	1.(A+A)	A	A
4.0.16	Boolean identity	!(A+B+!A)	0	0
4.0.17	Boolean identity	A.1	A	A

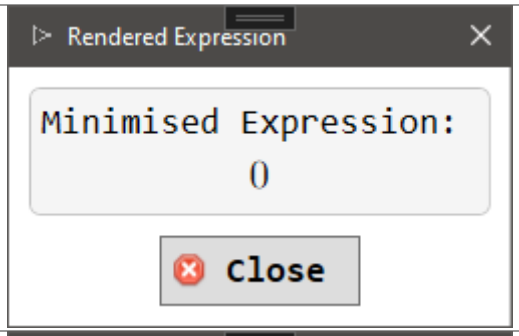
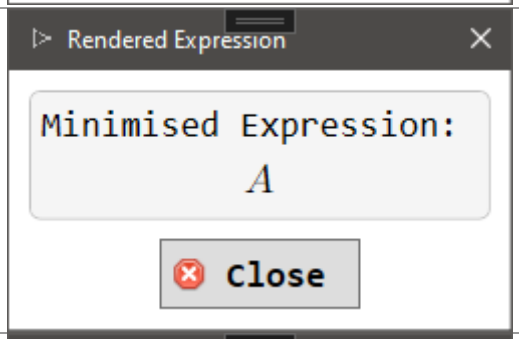
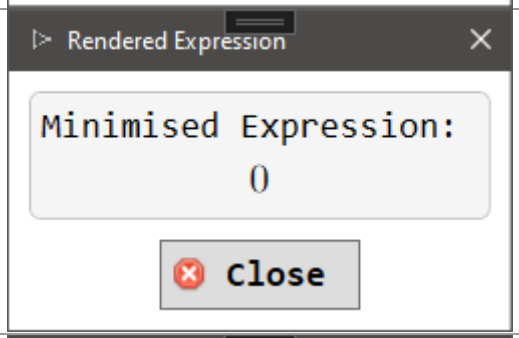
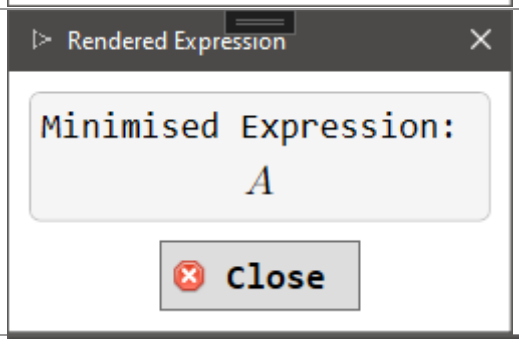
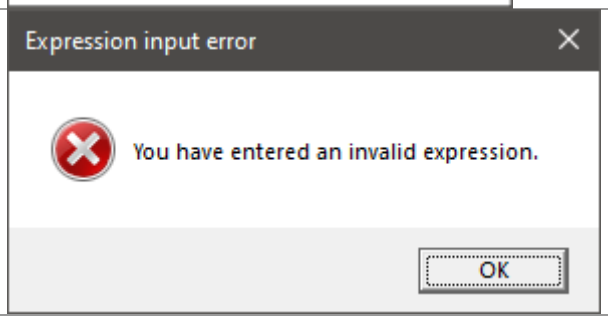
4.0.18	Null expression	" "	Error message shown	Error message shown
Test comment: As mentioned previously, it is very difficult to test Petrick's method due to the rarity of its use. The following test, however, is one test where the method must be used. This comes from Wikipedia.				
4.1.0	Application of Petrick's	(!A.!B.!C)+(!A.!B.C)+ (!A.B.!C)+(A.!B.C)+ (A.B.!C)+(A.B.C)	(!A.!B)+(B.!C)+(A.C)	(!A.!B)+(B.!C)+(A.C)

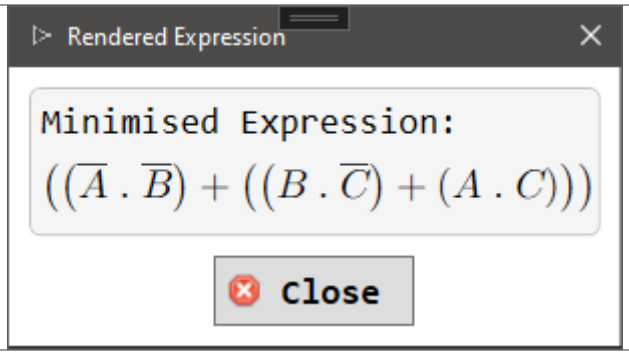
Evidence of tests

Test ID	Evidence
4.0.0	
4.0.1	
4.0.2	
4.0.3	

4.0.4	Rendered Expression Minimised Expression: $((C \,.\, D) + (A \,.\, \overline{B}))$ Close	
4.0.5	Rendered Expression Minimised Expression: A Close	
4.0.6	Rendered Expression Minimised Expression: $(C + A)$ Close	
4.0.7	Rendered Expression Minimised Expression: B Close	
4.0.8	Rendered Expression Minimised Expression: $\overline{A}$ Close	

4.0.9		
4.0.10		
4.0.11		
4.0.12		
4.0.13		

4.0.14	
4.0.15	
4.0.16	
4.0.17	
4.0.18	

4.1.0	

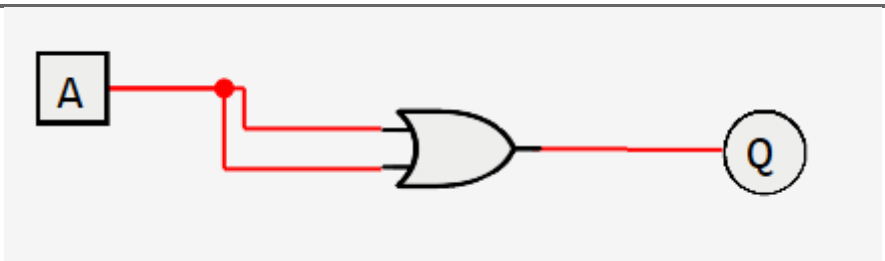
Minimising diagrams

The following tests are a continuation of the previous set of tests. They cover drawing minimised diagrams from currently drawn ones. There is no need for a large array of tests as both diagram drawing and minimisation have been tested before.

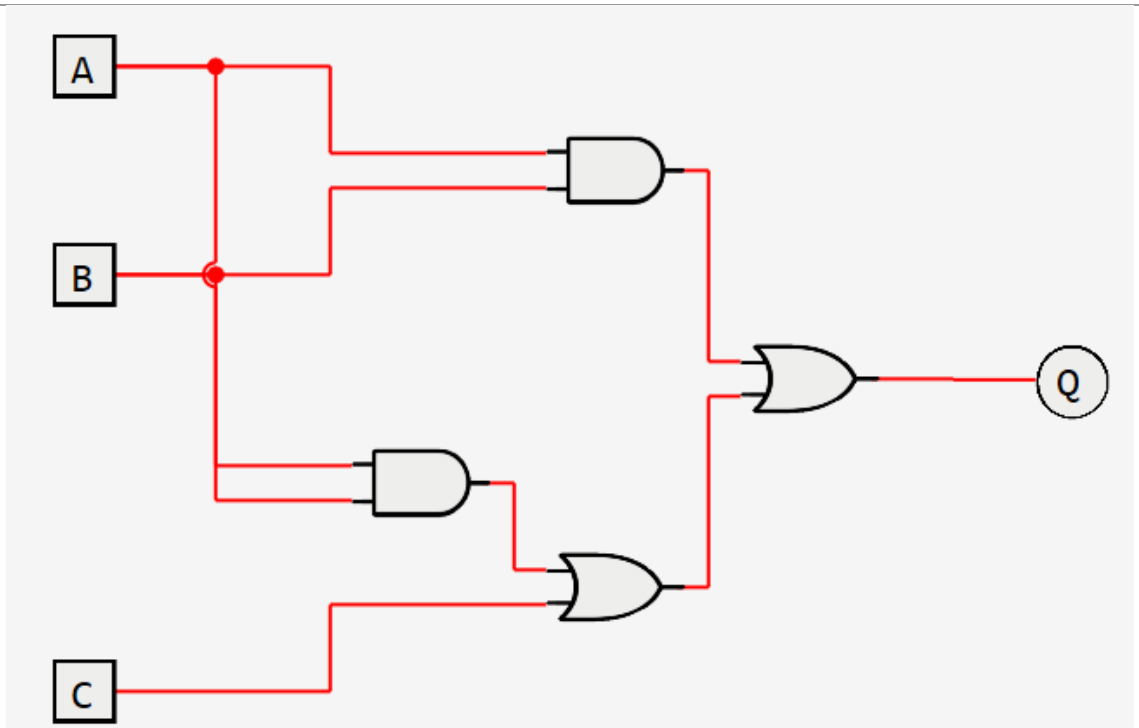
Test ID	Description	Test data	Expected outcome	Outcome
5.0.0	Boolean identity	A+A – Clicking 'Minimise diagram'	A	A
5.0.1	Boolean identity	(A.B)+(A.B)+C - Clicking 'Minimise diagram'	(A.B)+C	(A.B)+C
5.0.2	Nested NOT gates	!(!(A+A)) – Clicking 'Minimise diagram'	A	A
5.0.3	Repeated sub-expression	(A.!B)+(C.D)+(C.D) – Clicking 'Minimise diagram'	(C.D)+(A.!B)	(C.D)+(A.!B)
5.0.4	No diagram has been drawn	" " – Clicking 'Minimise diagram'	Error message	Error message

Evidence of tests

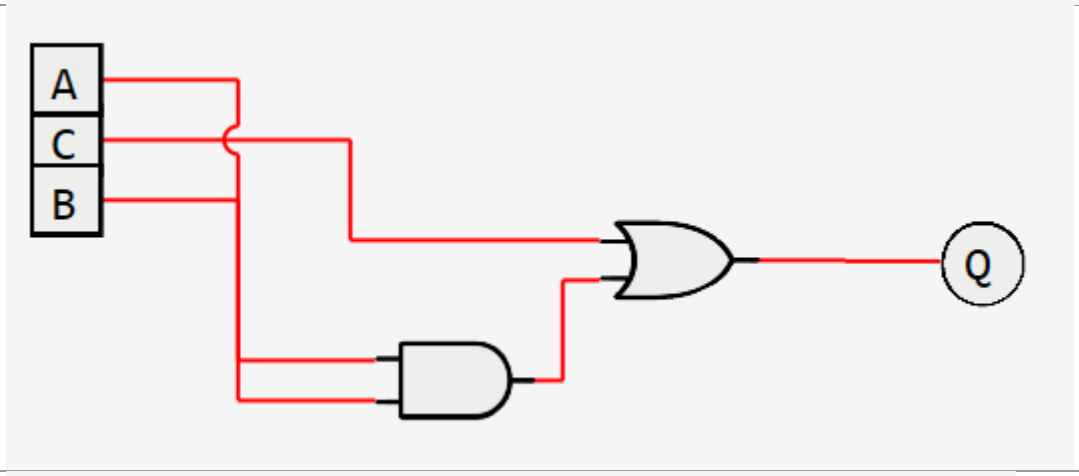
In the table below there are 2 images for each test, marked 5.0.0a and 5.0.0b (for example). The images marked with 'a' are the original diagrams and the ones marked with 'b' are the minimised diagrams.

Test ID	Evidence
5.0.0a	
5.0.0b	

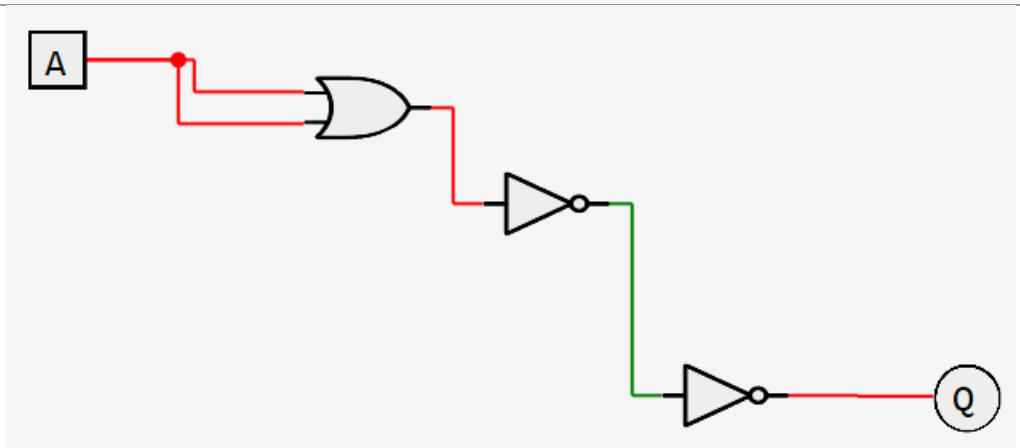
5.0.1a



5.0.1b

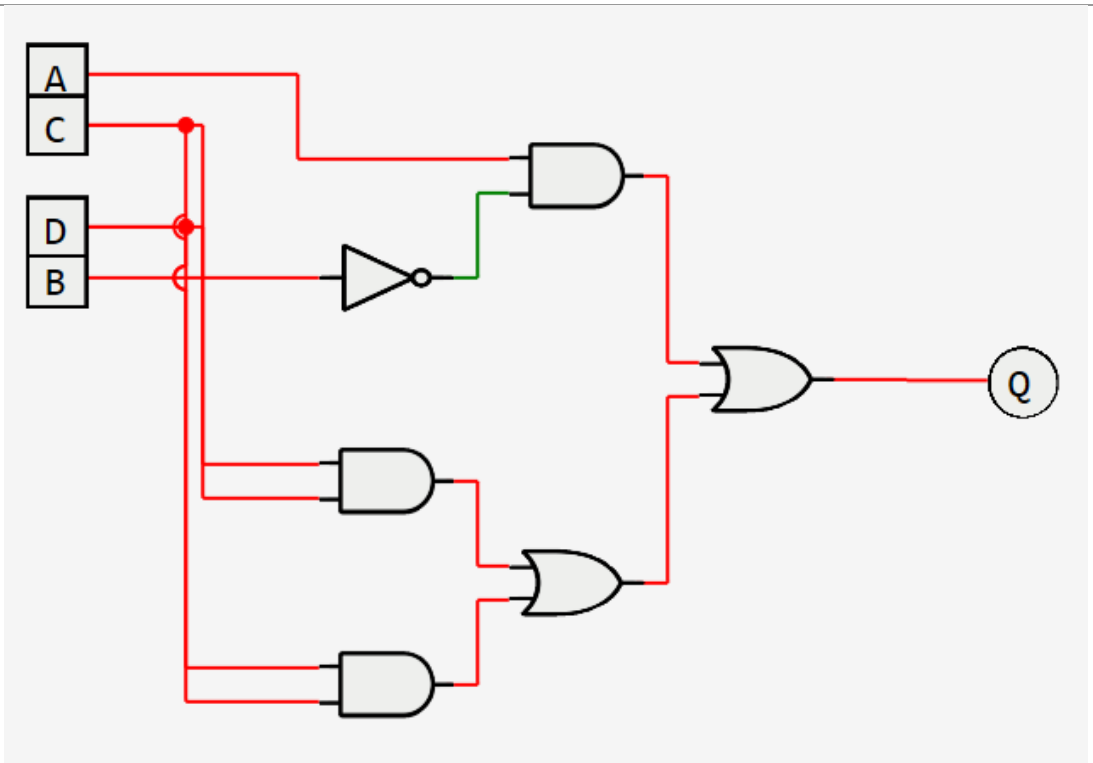


5.0.2a

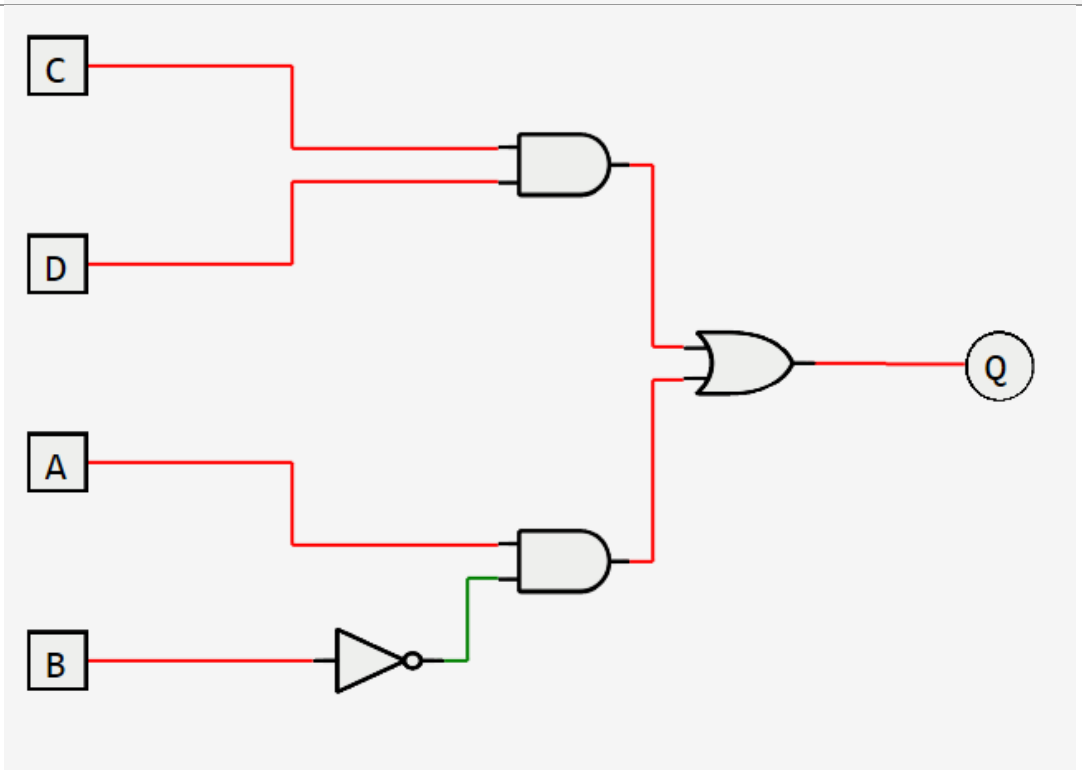


5.0.2b

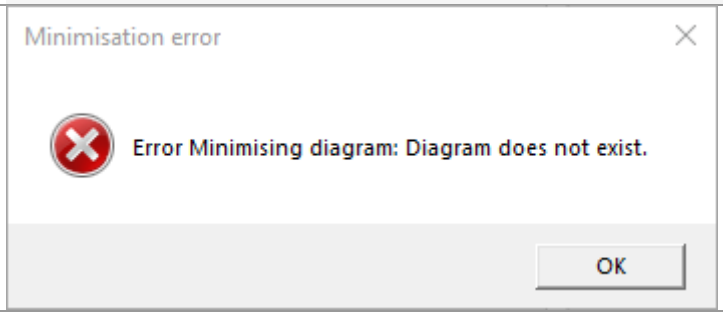
5.0.3a



5.0.3b



5.0.4



User interface

The following tests cover the user interface of the program and the functionality that it has. When considering the items for testing some buttons can be discounted due to them already being used to run previous tests. This gives the following list of items to test:

- Pop-up menus for gate information.
- 'Clear Diagram' and 'Clear Truth Table' buttons.
- 'Find Expression' button.
- File handling buttons:
    - Save
    - Load
    - Export

These tests are recorded in the table below:

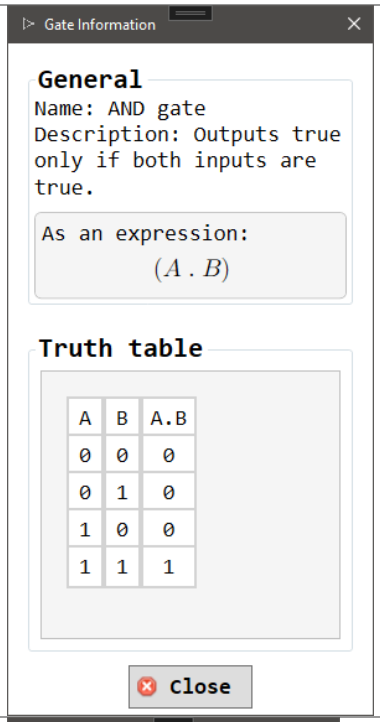
Main window button testing

Test ID	Description	Test data	Expected outcome	Outcome
6.0.0	Opening AND gate popup	Button press.	Pop-up opens	Pop-up opens
6.0.1	Opening OR gate popup	Button press.	Pop-up opens	Pop-up opens
6.0.2	Opening NOT gate popup	Button press.	Pop-up opens	Pop-up opens
6.0.3	Opening XOR gate popup	Button press.	Pop-up opens	Pop-up opens
6.0.4	Opening NAND gate popup	Button press.	Pop-up opens	Pop-up opens
6.0.5	Opening NOR gate popup	Button press.	Pop-up opens	Pop-up opens
6.0.6	Clearing the current diagram	(A.B) [An example expression for testing purposes] – Button press.	Diagram is cleared	Diagram is cleared
6.0.7	Clearing the current truth table	(A.B) [An example expression for testing purposes] – Button press.	Truth table is cleared	Truth table is cleared

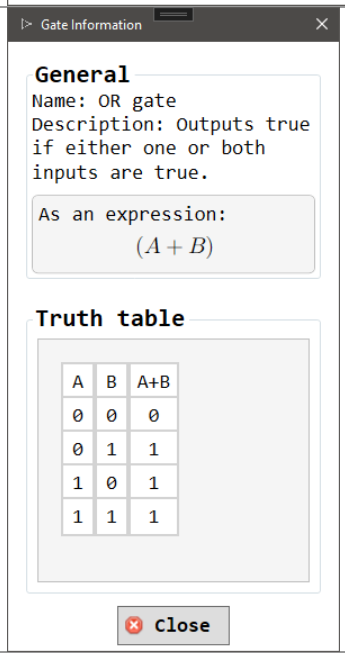
Evidence of tests

Test ID	Evidence

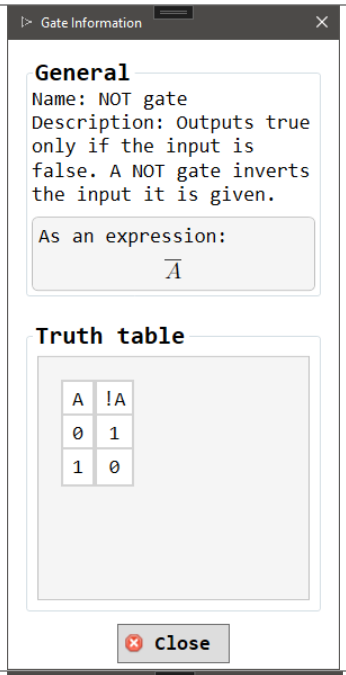
6.0.0


Gate Information
General
Name: AND gate
Description: Outputs true only if both inputs are true.
As an expression:
$(A \,.\, B)$
Truth table
A B A.B
0 0 0
0 1 0
1 0 0
1 1 1
Close

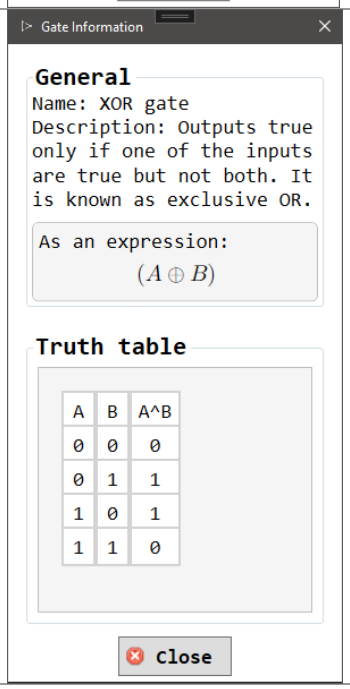
6.0.1


Gate Information
General
Name: OR gate
Description: Outputs true if either one or both inputs are true.
As an expression:
$(A + B)$
Truth table
A B A+B
0 0 0
0 1 1
1 0 1
1 1 1
Close

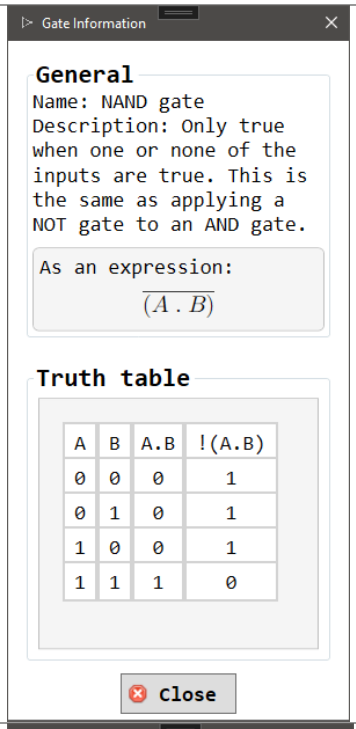
6.0.2


Gate Information
General
Name: NOT gate
Description: Outputs true only if the input is false. A NOT gate inverts the input it is given.
As an expression:
$\overline{A}$
Truth table
A !A
0 1
1 0
Close

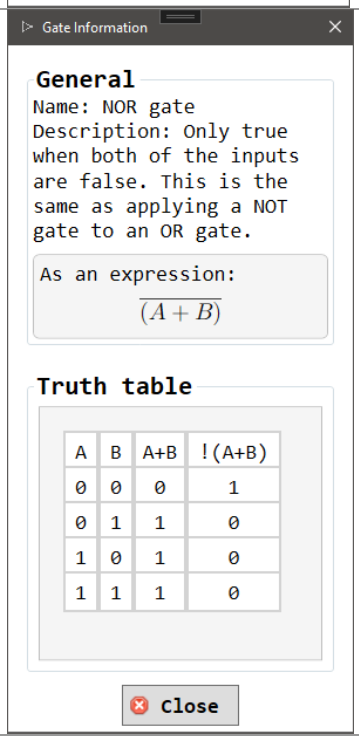
6.0.3


Gate Information
General
Name: XOR gate
Description: Outputs true only if one of the inputs are true but not both. It is known as exclusive OR.
As an expression:
$(A \oplus B)$
Truth table
A B A^B
0 0 0
0 1 1
1 0 1
1 1 0
Close

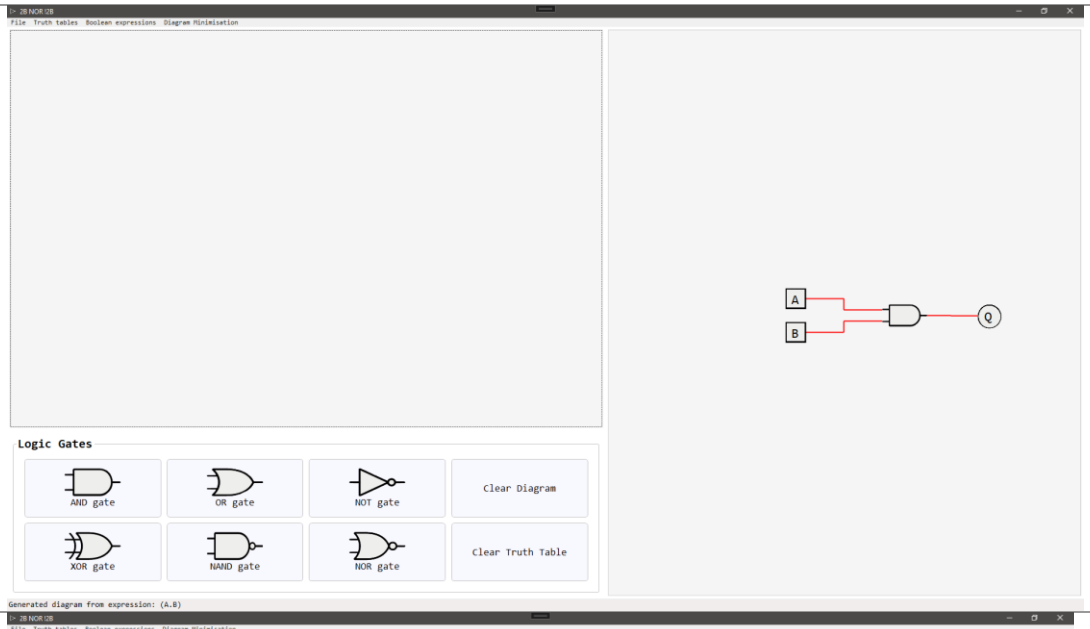
6.0.4


Gate Information
General
Name: NAND gate
Description: Only true when one or none of the inputs are true. This is the same as applying a NOT gate to an AND gate.
As an expression:
$\overline{(A \,.\, B)}$
Truth table
A B A.B !(A.B)
0 0 0 1
0 1 0 1
1 0 0 1
1 1 1 0
Close

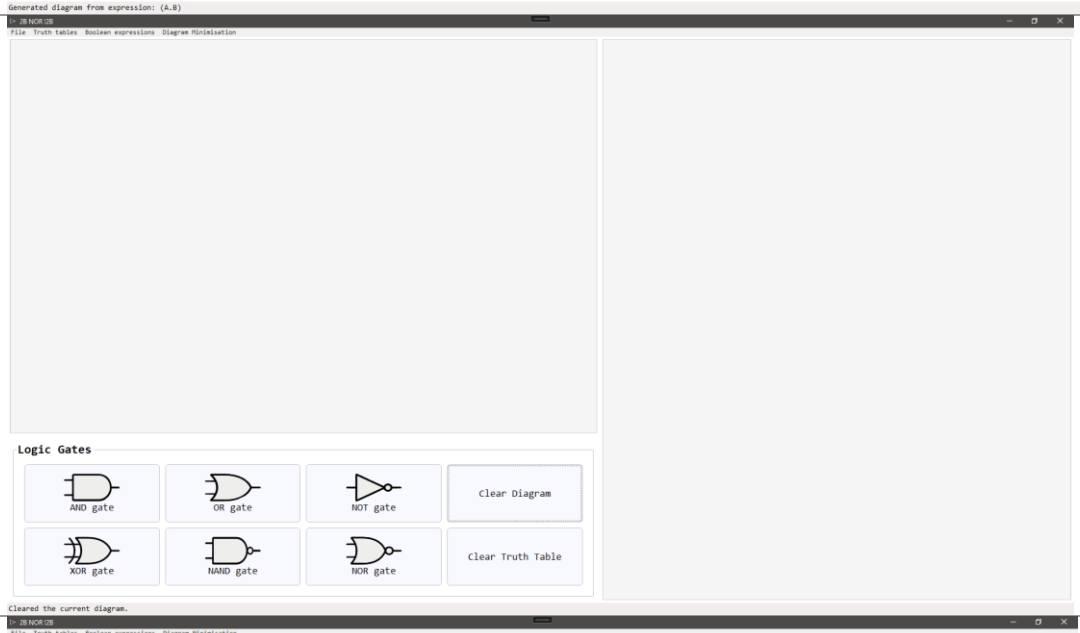
6.0.5


Gate Information
General
Name: NOR gate
Description: Only true when both of the inputs are false. This is the same as applying a NOT gate to an OR gate.
As an expression:
$\overline{(A + B)}$
Truth table
A B A+B !(A+B)
0 0 0 1
0 1 1 0
1 0 1 0
1 1 1 0
Close

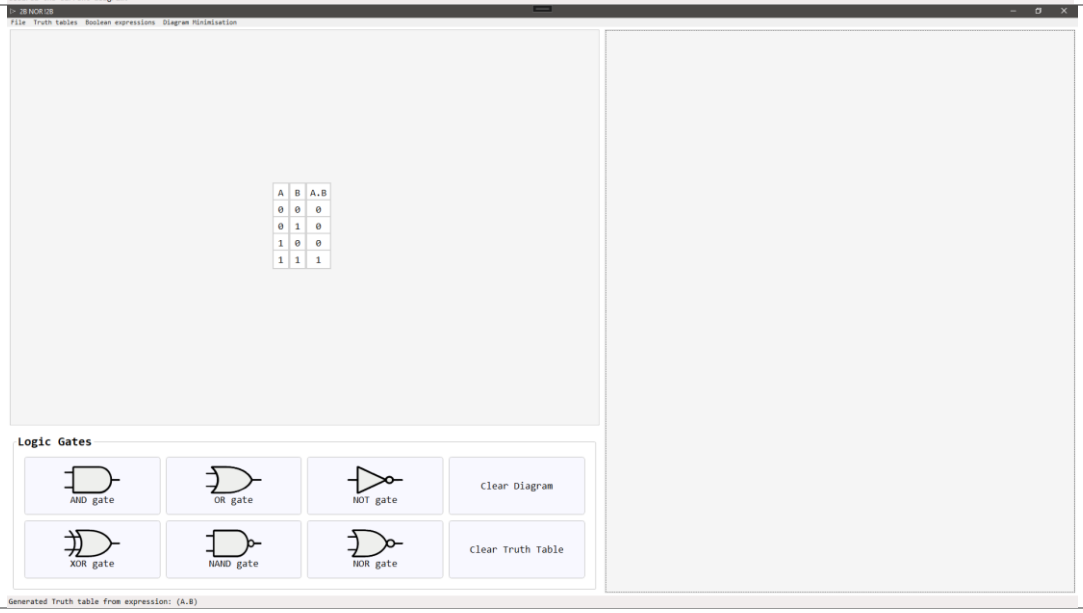
6.0.6a (before clear)

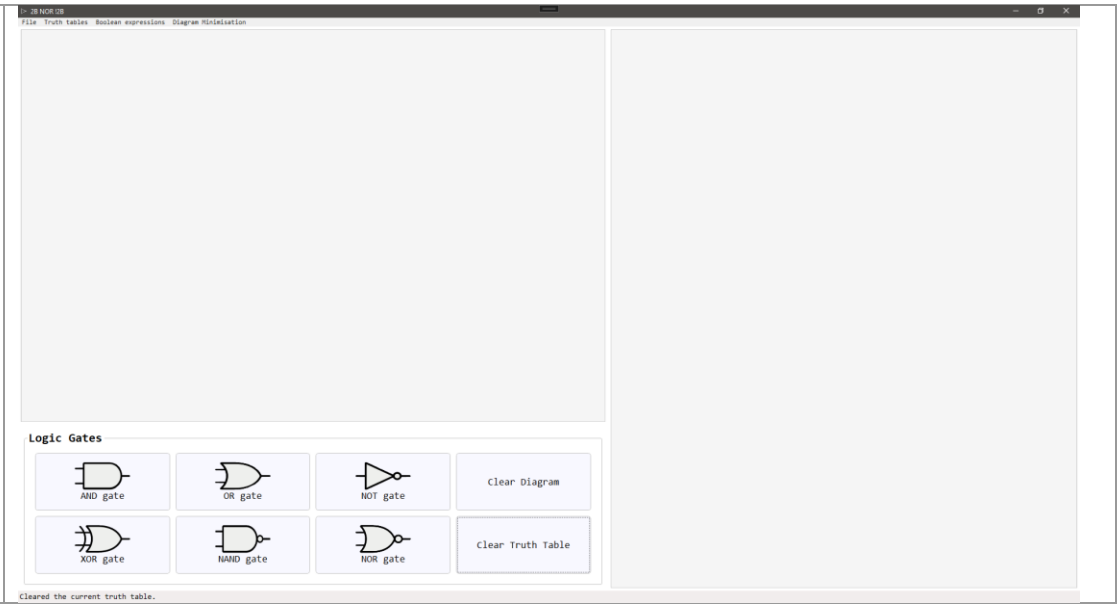


6.0.6b (after clear)



6.0.7a (before clear)



6.0.7b (after clear)	

Finding expressions

To get a rendered expression of the currently drawn diagram, the user can click the 'Find Expression' button. This will show them the rendered LATEX Boolean expression that has been drawn. This is shown below:

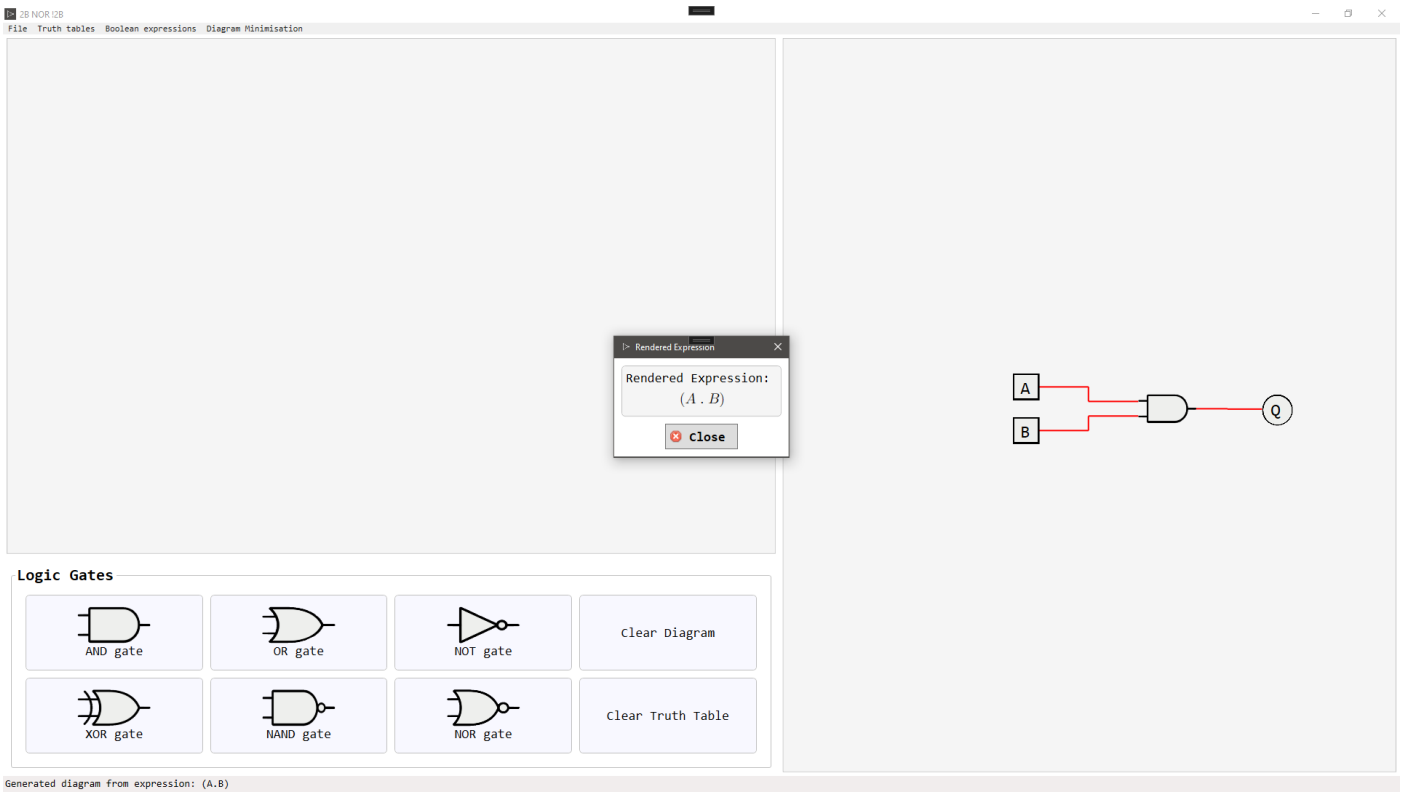


Figure 18: The 'Find Expression' button in use with an example expression.

Saving, Loading, and exporting diagrams

One of functions in the program is the ability to save, load and export drawn logic circuit diagrams within the program.

Test ID	Expression	Saved (Y/N)	Loaded (Y/N)	Exported (Y/N)
7.0.0	(A.B)+C	Y	Y	Y
7.0.1	(A.B)	Y	Y	Y
7.0.2	A	Y	Y	Y
7.0.3	(!A.!B)^(!A+A)	Y	Y	Y

7.0.4	(A.B)^(C+D)	Y	Y	Y
7.0.5	((A.B)+(C.D))^((E.F)+(G.H))	Y	Y	Y
7.0.6	(A.B)+(!C)	Y	Y	Y
7.0.7	Null expression	N	Y	N

Test comment: The program is not checking whether or not the string being loaded is null.

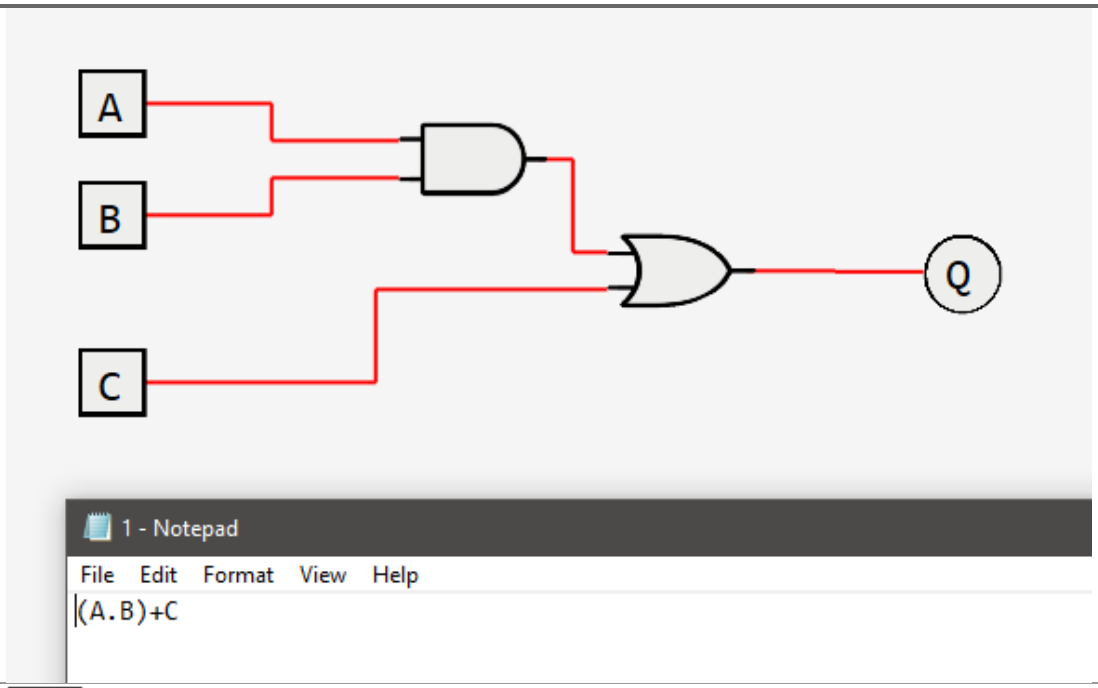
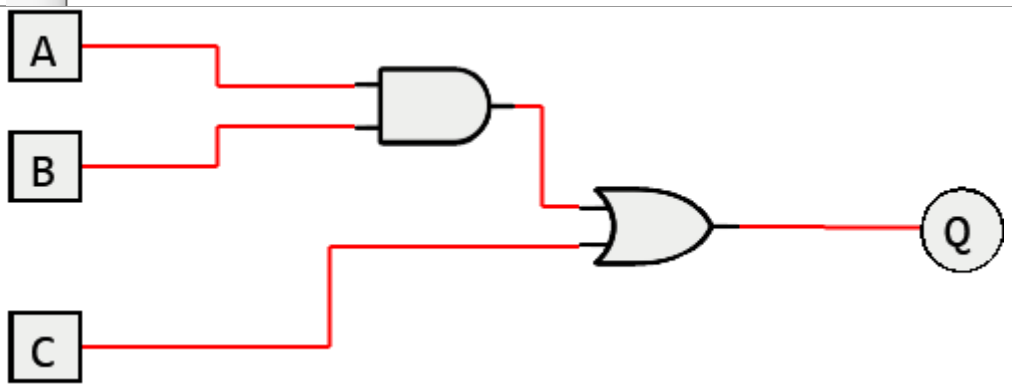
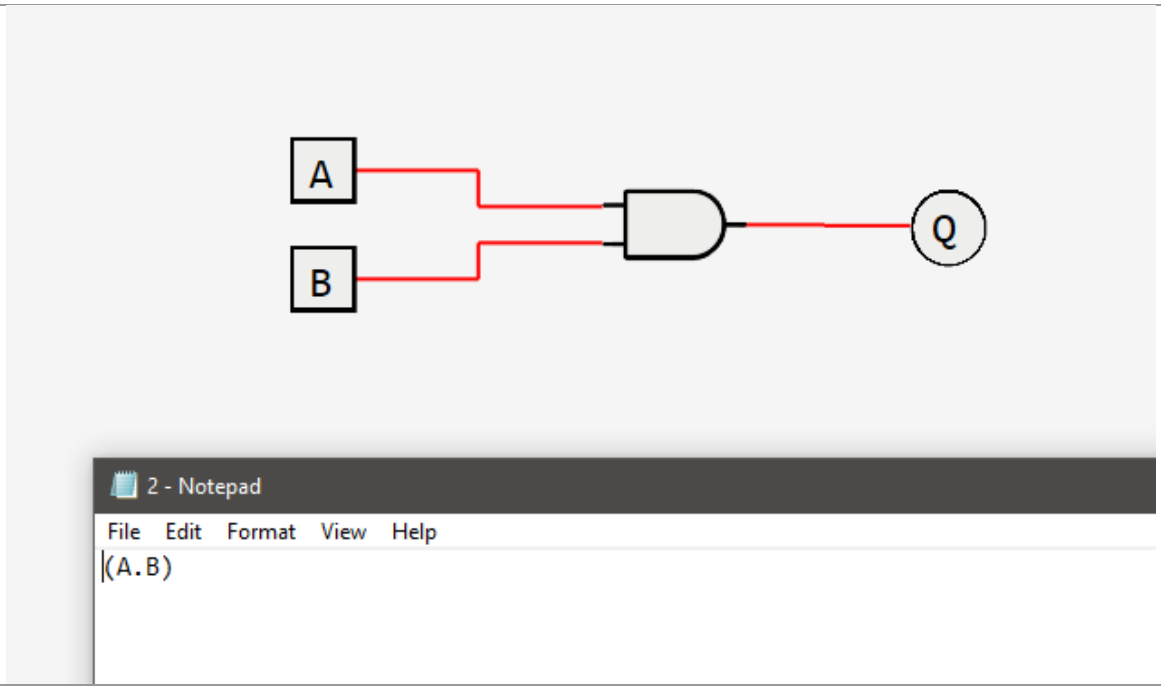
Solution: Add a check for whether or not the string being loaded is empty or not.

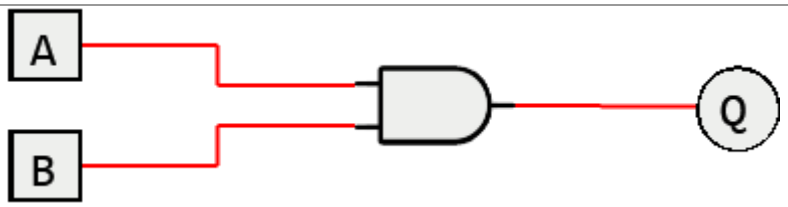
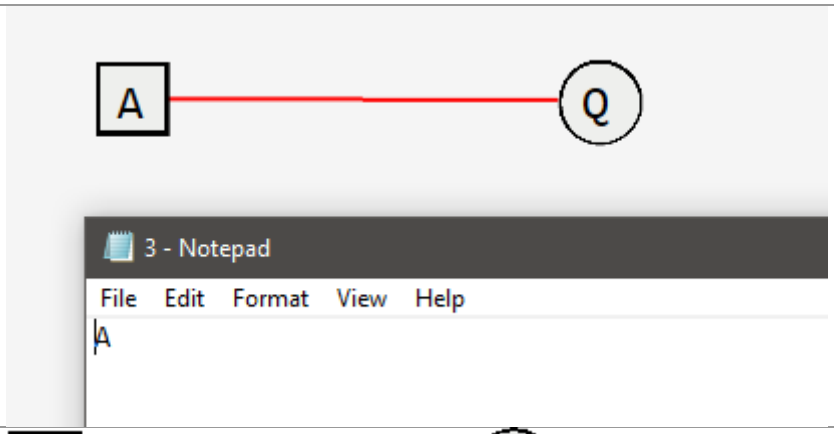
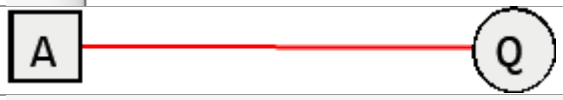
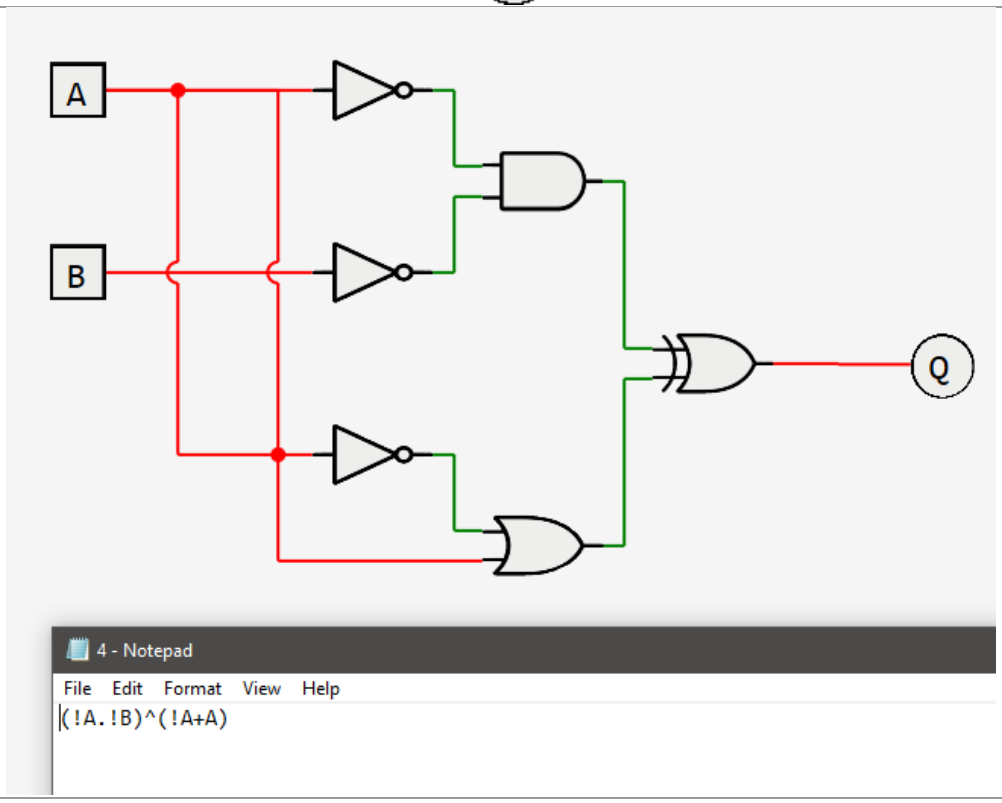
```
        private void MenuItem_LoadDiagram(object sender, RoutedEventArgs e)
        {
            var openFileDialog = new OpenFileDialog()
            {
                Filter = "Text file (*.txt)|*.txt|XML
(*.xml)|*.xml|Expression file (*.2B)|*.2B",
                DefaultExt = "Expression file (*.2B)|*.2B"
            };
            // Showing the open file dialog to get the open path.
            openFileDialog.ShowDialog();
            try
            {
                // Reading all text from the file and this the expression
to be drawn
                // to the main window. Only if it is valid otherwise an
error has happened.
                saveString = File.ReadAllText(openFileDialog.FileName);
                if (saveString.Length < 1)
                {
                    MessageBox.Show("This expression does not exist.",
"Open File Error", MessageBoxButton.OK, MessageBoxImage.Error);
                }
                else
                {
                    if (d.IsExpressionValid(saveString))
                    {
                        d.SetExpression(saveString);
                        d.DrawDiagram();
                        statusBar.Text = "Loaded diagram from " +
openFileDialog.FileName;
                    }
                    else
                    {
                        MessageBox.Show("This expression is invalid. Please
try again. ");
                    }
                }
            }
            catch (Exception ex)
            {
                // The file could not be opened by the program.
                MessageBox.Show($"Error opening file:\n{ex.Message}", "Open
File Error", MessageBoxButton.OK, MessageBoxImage.Error);
                e.Handled = true;
            }
        }
```

`7.0.7	Null expression	N	N	N

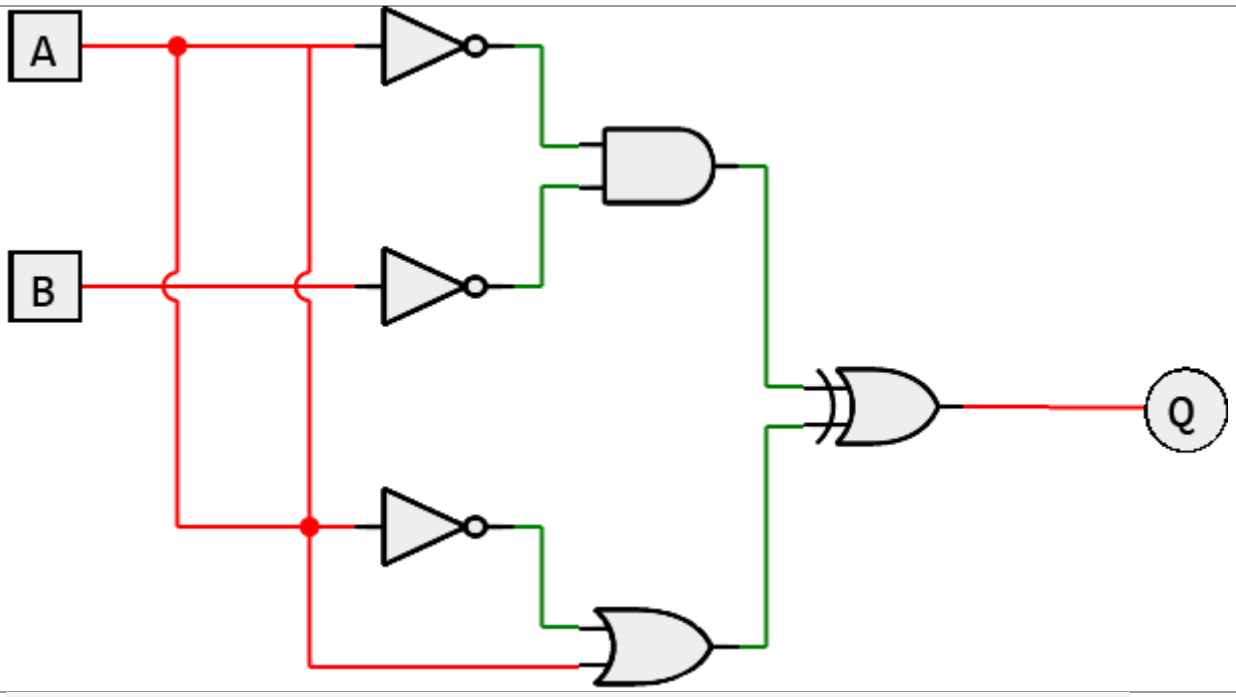
Evidence of tests

In the table below, there will be two images for each test. These will be marked with ‘a’ and ‘b’. Any image marked with ‘a’ is the evidence for the saving and loading test. Any image marked with ‘b’ is the evidence for the exporting test.

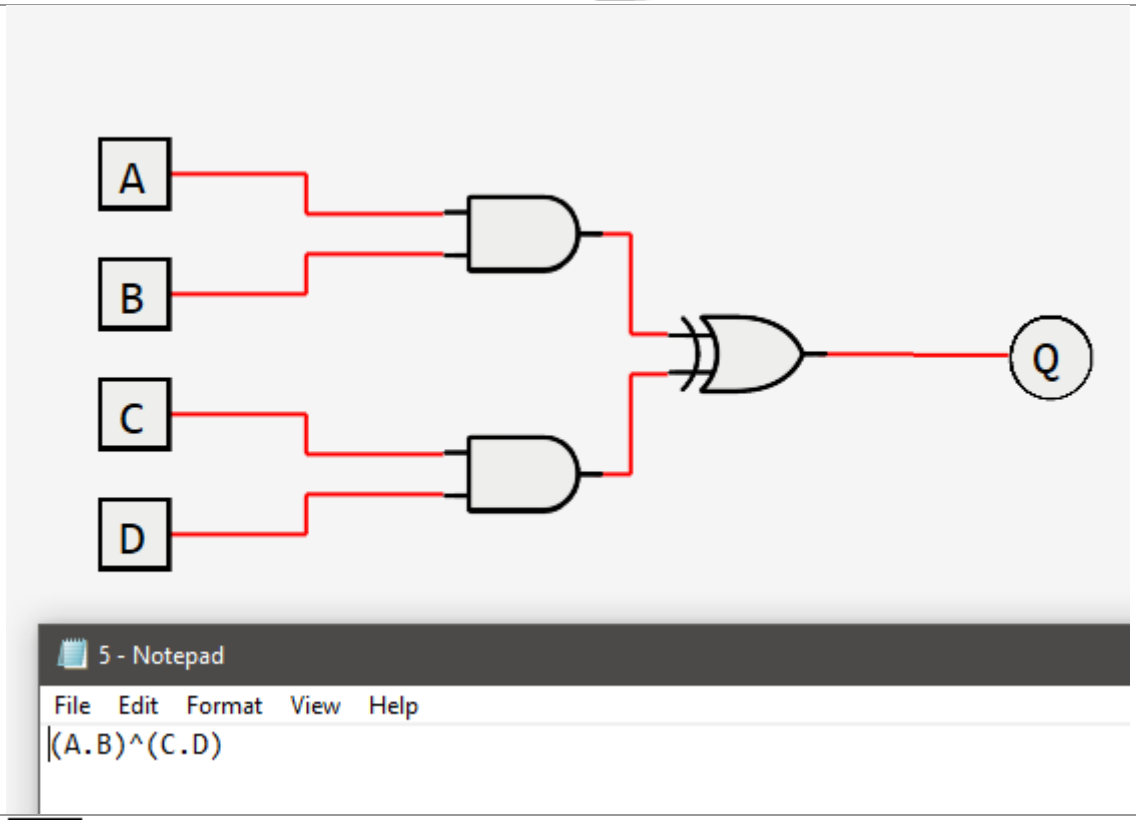
Test ID	Evidence
7.0.0a	
7.0.0b	
7.0.1a	

7.0.1b	
7.0.2a	
7.0.2b	
7.0.3a	

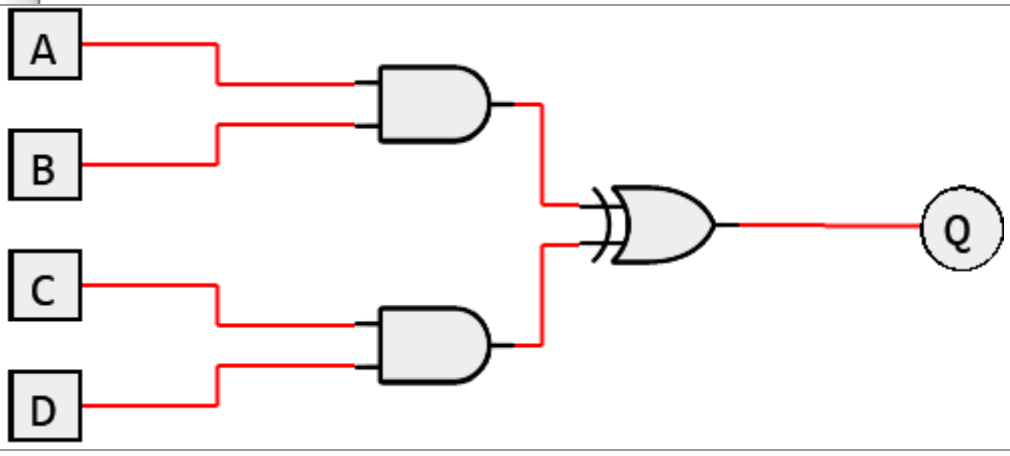
7.0.3b



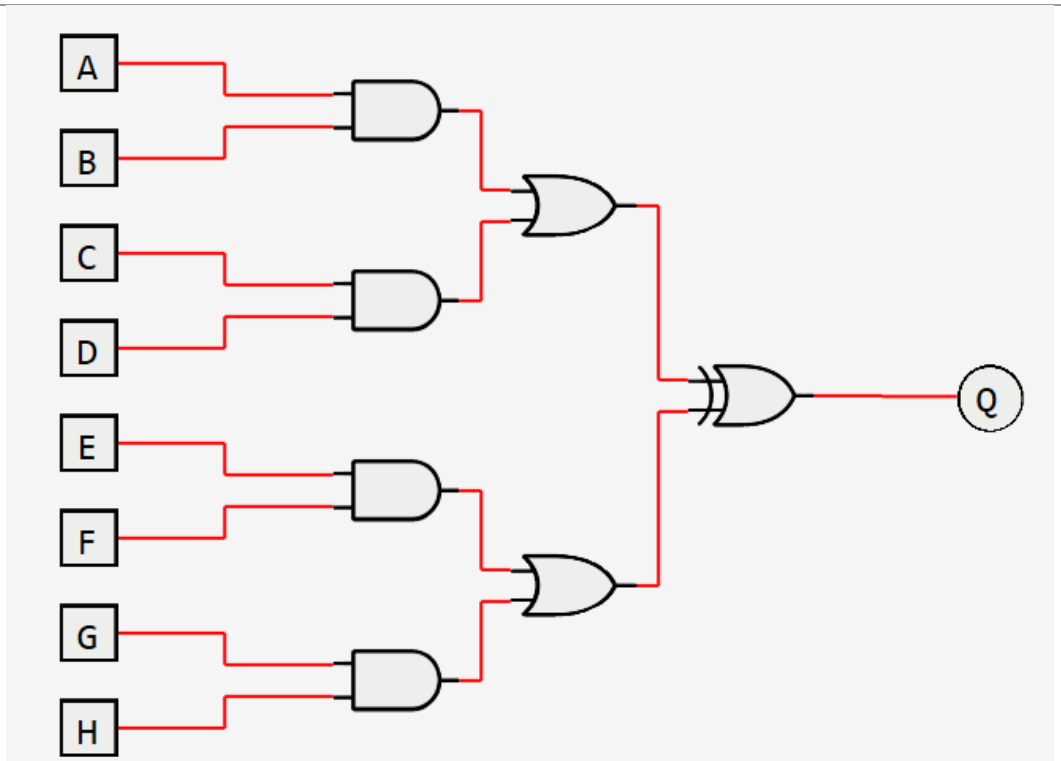
7.0.4a



7.0.4b

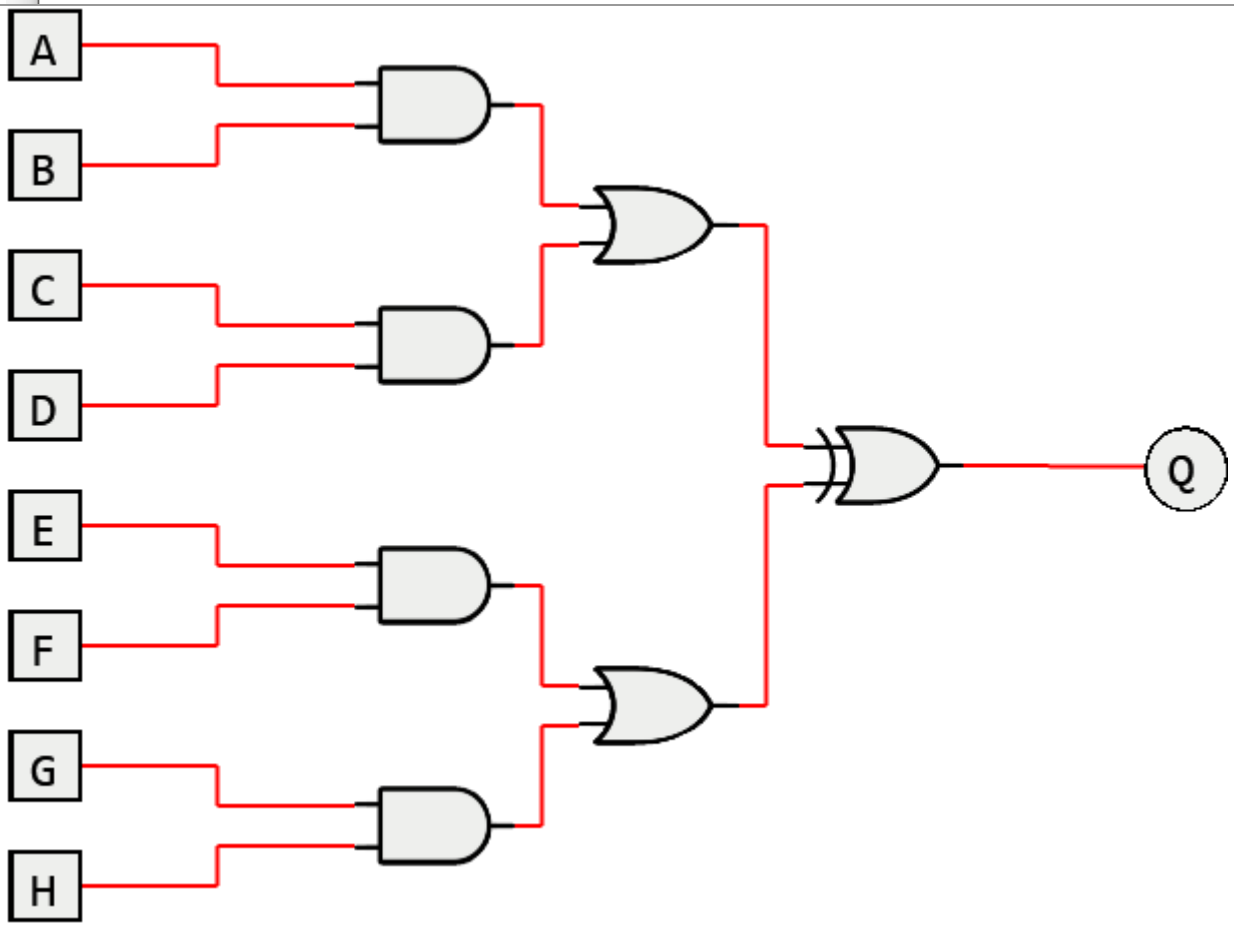


7.0.5a

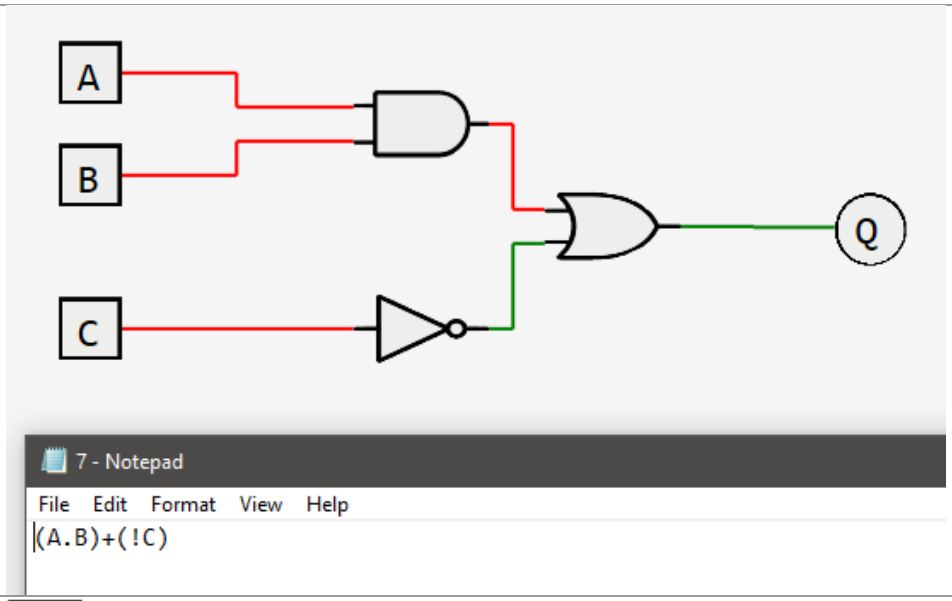

A
B
C
D
E
F
G
H
Q


6 - Notepad
File Edit Format View Help
((A.B)+(C.D))^((E.F)+(G.H))

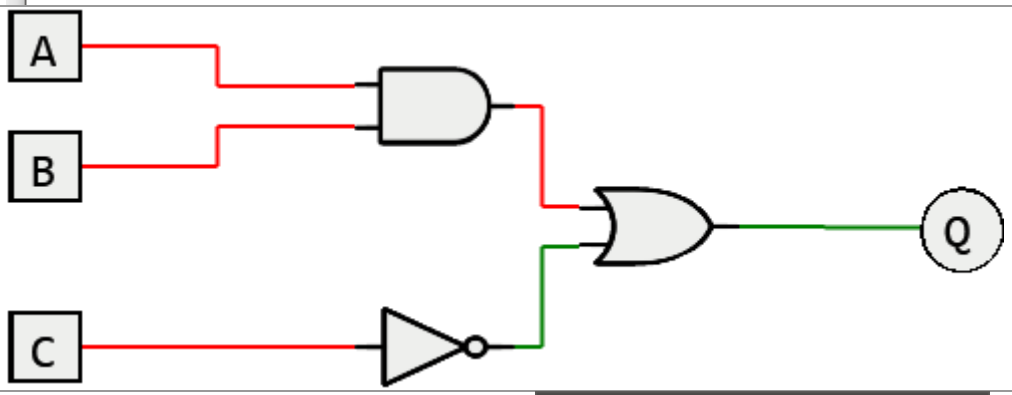
7.0.5b


A
B
C
D
E
F
G
H
Q

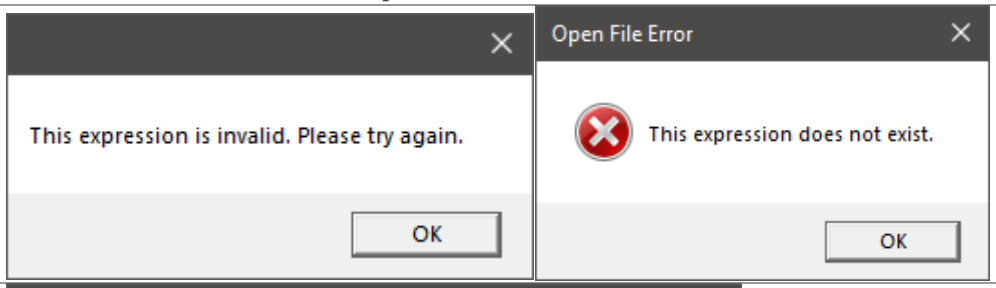
7.0.6a



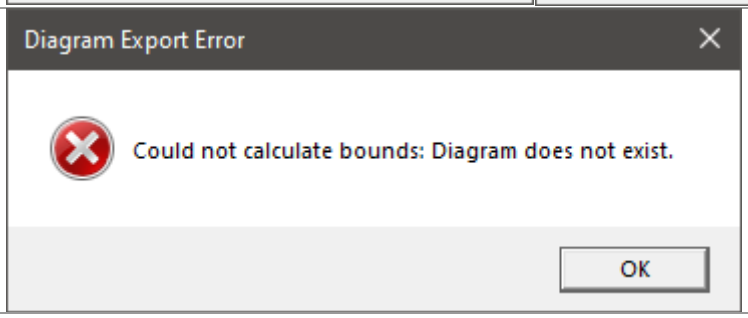
7.0.6b



7.0.7a



7.0.7b



Evaluation

Objectives

The following section compares the program with the objectives defined at the beginning of the project. It will discuss both the core and extension objectives.

Core

1. **Every feature should be no more than three clicks away.**

The point of this objective is to ensure that the user interface is clear and easy to use for any user, new or advanced. This is because if many clicks are required to use an application, then some tasks can be laborious for all users as they have to go through a huge process to complete task. When looking at the program's interface the features are easily accessible, and they all can be reached from the main window of the program. This is done through the array of buttons in the bottom-left corner of the main window, which contain all of the pop-up menus and the 'clear diagram' and 'clear truth table' buttons. The use of a menu-strip at the top of the application window also packages all of the program's features into a compact and neat space. When using the program only generating truth tables and logic gate diagrams require more than a single left click on the menu strip. This is because the user must confirm their entered expression when using the input dialog. The location of the pop-up menus on the main window also makes the help functions of the program easily accessible and removes the need for a large nesting of menus in the menu strip at the top of the main window. The lack of nested menus within the menu strip at the top of the application also means that it is very easy for a user to search for features within the program if they are unaware of their location. This is because navigating through the menus is very quick because there are few in number.

When a diagram has been drawn, the user can immediately start interacting with the diagram by simply clicking on the inputs of the diagram, with the diagram automatically reflected the new state by changing the colour of the wires. This system removes the need for an 'interact mode' or 'hand tool' which is a special cursor used to interact with drawn diagrams. This is the system that is currently employed by Logisim when interacting with drawn diagrams. As the most involved process within the program is creating a diagram which involves two clicks and inputting an expression, this is objective can be considered to be complete.

2. **A new user should be able to create a diagram and its respective truth table from a Boolean expression in less than 2 minutes.**

This builds upon the points made in the previous objective where diagram and truth table creation must be a simple as possible. This because they are the two features within the program that would be most used as they are the primary purpose for using the application in the first place. As mentioned later on, in the user feedback section, it took students an average 1.5 minutes to draw and generate the truth table for the expression (A.B)^A. This exceeds the initial criterion for the success of this objective, showing that the programs most important functions are simple and easy to use, as shown by the speed in which the new users completed the task. The truth table and diagram are given below:

A	B	C	A.B	(A.B)^C
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	0

Figure 20: The truth table new users were asked to recreate.

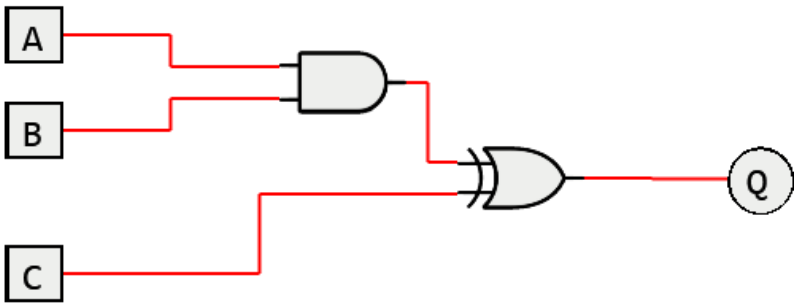


Figure 19: The logic gate diagram new users were asked to recreate.

One of the implications of this, is if the program is being used within a classroom setting. As diagram creation and truth table generation are simple and quick to learn how to do, this means that there is a very shallow learning curve when using the program. This contrasts Logisim, where it can take some students multiple lessons to learn how to use it (3+ hours). By extension, with the application in a lesson, as diagram drawing and truth table generation are quick to learn and use (shown by objective 1), significantly less lesson time can be spent on learning the software which means more time can be spent by the students learning and exploring logic gates and Boolean algebra.

3. **The program will not lag on a reasonably powerful computer.**

When writing the objectives for this project, I thought that this would be a much more important objective than what it eventually turned out to be. This is because there are not many points, within the program, where huge amounts of iteration is carried out. The stage where the most time is taken up is generating the prime implicants. This is because of nested loops and recursion being used within the program. Although, there are generally very few prime implicants for a given Boolean expression meaning that execution times are still very short. However, it must be noted that there are some sparing cases where the number of prime implicants can be extremely large (upwards of 1 million), but these cases are few and far between. The program consistently runs at 60fps+ which leads to believe that this objective for the program has been met.

4. **The program will not error regardless of input.**

This objective evaluates the validation and exception handling of the application. As shown in testing, the validation, although not perfect, covers a significant majority of expressions being entered into it. One of the reasons for this is the use of a dialog to enter the Boolean expressions. This makes for an easy entry point for a validation method to be used, which is what occurred in the program code. Additionally, the strength of the program's validation comes from the variety of ways in which the user-entered expression is being validated. Some of the techniques used include regular expression to check for anomalous brackets, stacks in order to validate the order of brackets within the expression and a postfix to infix conversion to check to see if the expression produces valid infix. This diversity of checking means that very specific expressions must be used in order for an invalid expression to pass through. However, it is still possible for these to occur. This is because Boolean expressions are an irregular language, making virtually any combination of brackets, logic gates, inputs, and constants possible. This means that it is extremely difficult to write a validator that covers all cases. This leads me to think that the wording for this objective is incorrect as it is almost impossible to fully validate every single Boolean expression with the same system. I think a more fitting wording would be "The program will not crash regardless of inputted expressions". This is because, as shown by my testing, the program can correctly validate a majority of expressions that are entered into it.

Something else to consider is that although the program cannot correctly validate every expression that is entered into it, the use of exception handling throughout the program's user interface means that although the program can accept false positives, the application will not crash due to the exceptions being handled and shown to the user through an error message, which describes the error that has occurred and what they need to fix for an exception to not be thrown. Examples within the program include the saving and loading functions for diagrams, where exceptions are raised when there is an erroneous file be saved/loaded. Therefore, (although this objective could have been worded better), I feel that the program has met this objective.

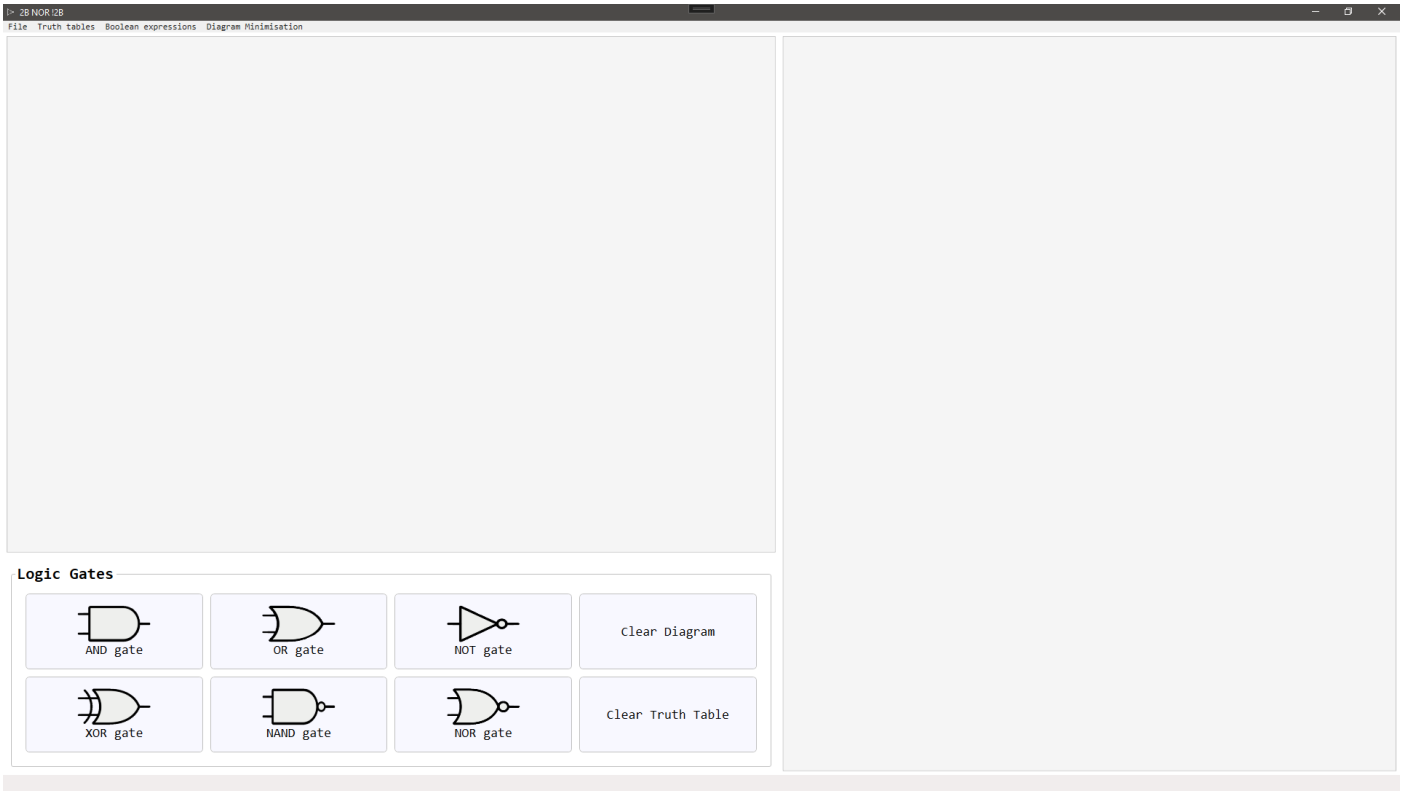
5. **The system's output will be versatile for students/teachers.**

The use of this program is by students and teachers within a classroom setting, this makes it very important for both students and teachers to be able to export diagrams from the program, as it allows students to show their work within documents and it also allows for teachers to produce worksheets or teaching materials for their classes. Currently the program can export PNG images of diagrams drawn within the program. Diagrams can also be saved and loaded into/out of the program, allowing students to continue their work at a later point. The format of these files are simply ".txt" as they only store a Boolean expression. These files are validated to ensure that the program does not error when they are being saved/loaded. Diagrams that are loaded into the program can also be further interacted with such as interacting with the diagram and minimising it. Additionally, the changes of saving and

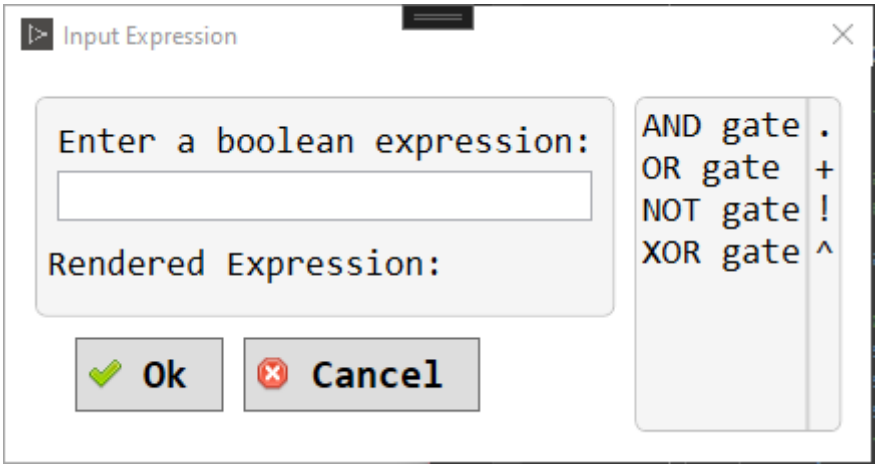
loading into and out of the program are indicated on the status bar at the bottom of the window. This small feature makes the program more complete as an application, but also informs the user that they have successfully carried out their desired action. As the program can export multiple formats, this objective is deemed to be complete.

6. **The program should provide help to users if they encounter difficulty using the program.**

Due to the applications' use case, being used by students learning about logic gates within a lesson, it is imperative that the program provides sufficient resources to help students should they encounter difficult with the program. The program provides help to users in a variety of ways. The main one of these are the pop-up menus that are accessible from the main window of the application. The display can be seen in the bottom-left corner of the main window, making them easily accessible (as mentioned in objective 1). The array can be seen below:



These pop-ups provide general information for each of the logic gates within the A-level specification which allow users to quickly reference information without leaving the application and searching through the internet. This improves the workflow when learning logic gates as the student doesn't have to leave the application. The success of these pop-ups is further evaluated through the student feedback. Another way the application provides help to users is through the Boolean expression input dialog. This can be seen below:



This is through the small table on the right side of the dialog. This reminds the user of the symbols within the expression and the logic gates that they represent which removes any potential confusion as to what each symbol represents. As mentioned in the user feedback, students could complete important tasks within the program, suggesting that the help provided by the program is sufficient, satisfying this objective.

7. **The program should cover all elements within the GCSE and A-Level specifications.**

As shown by the client feedback, discussed later on, this objective is deemed to be achieved successfully. This is because their examination of the program (given in the client feedback) they concluded that the program was fit for use within a classroom situation which is the programs' primary use case. The student feedback of the program also agrees with this with praise given by the students in a questionnaire given to them about the program.

8. **The program will be written to the current coding conventions from Microsoft.**

As mentioned in the technical solution for the application, the code written for the application follows the Microsoft C# coding conventions, for general programming, identifiers, and comments. The program follows these fully, ensuring that the code is readable, consistent, and understandable. One of the implications of this is that it makes it easy to continue developing the program as it well documented meaning that the purpose and function of parts of the code are easy to find out and understand. This also means that code can be easily understood by others. This makes it easy to implement the code written for the application into other projects as the classes and methods are documented as to what they do. Some of the conventions also force good programming practice, such as limiting the number of characters per line of code. This is because I am forced to simplify the logic of the program and not write huge if statements and convoluted for loops that are complex, unwieldy, and verbose which make the program more difficult to read, write and use in other parts of the application.

9. **The program should minimise a complex logic gate diagram.**

As mentioned in the definition of this objective, diagram minimisation is only something that very partially explored within the A-level specification. From testing, the program can successfully minimise a significant majority of entered Boolean expressions. The program provides functionality for drawing these minimised diagrams (as shown in testing). This is useful for students revising Boolean algebra. This is because a student can do the question validate their answer with the program and find the truth table/diagram of the result. One point of note is that the program will not be able to simplify all entered expressions. This is because of the nature of the Quine-McCluskey algorithm. This is because some expressions have millions of prime implicants which then ultimately produces an intractable problem as the program will have to iterate through millions of cells trying to find a valid product of sums. It should be noted that the likelihood of this occurring is almost negligible and so the program still covers the goal of being able to minimise user-entered Boolean expressions. The criterion for the success of this objective is that the program should be able to simplify objectives in under 5 seconds. This is easily achieved by the program ultimately satisfying this objective.

Extension objectives

The following section covers some extension objectives/features that were defined in the analysis of the project.

10. **The program could have automatic diagram alignment to produce more visually appealing diagrams.**

This extension, I believe is useful for correcting the formulae used to calculate the position of the nodes on the canvas. The formula given in the definition provides a calculable method for improving diagrams. The usefulness of this extension is slightly diminished as the program only allows the creation of diagrams through expressions and not drag and drop drawing. However, it would still be an interesting challenge to try and find a way to redraw the diagram editing the formulae used to calculate the position of the nodes on the canvas. I also think that this feature would be very intensive for the computer to execute. This means that the algorithm would need to be heavily optimised. This is to avoid, drawing and calculating the diagram many, many times which is slow and inefficient. A method for identifying where issues could likely be would be very interesting and challenging to implement. It would

be worthwhile to implement this feature however I feel there are other more fitting extensions that should be prioritised over this one, such as showing the algebraic steps when minimising expressions.

11. **The program could provide questions to the user about simplification and logic diagrams.**

One of the best ways to learn a topic and improve your understanding, is the use of practice questions. Therefore, a system to ask a variety of questions would be extremely useful for students. This, in my opinion is enough for a whole other project in itself, and is therefore, beyond the scope of this project. The application in its current form does have the framework for asking questions, such as drawing diagrams and truth tables. For this system to be worthwhile, I feel a drag and drop implementation of the logic gates would be required. This is because with the framework given by the current application, the variety of questions would not be wide enough to provide a useful testing environment for the students. Being able to drag and drop components to complete diagrams, expressions and truth tables would provide a much more complete system, which is currently not possible with the current framework of the project. As I said before, adding this system would be another project in itself and one that would be interesting to implement outside of the project.

12. **The program could show the algebraic steps when minimising.**

Out of all of the extension objectives this is the one that can be most feasibly implemented into the application and would fit well into it. This has added benefits on being able to help students learn Boolean algebra as it shows them how to apply the laws and identities and ultimately, familiarises them with Boolean algebra which is needed for their exam. One of the reasons that this objective is feasible is that the program already carries out Boolean algebra. This is within Petrick's method where the distributive law is applied to the product of sums to convert it into the sum of products. This, however, would not be the most helpful for the users' learning, however. This is due to the nature of the expressions being dealt with. An example of the algebra is shown below, and has been taken from Wikipedia:

```
= (K+L)(K+M)(L+N)(M+P)(N+Q)(P+Q)
= (K+LM)(N+LQ)(P+MQ)
= (KN+KLQ+LMN+LMQ)(P+MQ)
= KNP + KLPQ + LMNP + LMPQ + KMNQ + KLMQ + LMNQ + LMQ
```

This algebra is beyond the scope of the A-level specification is also too different to the kind of A-level questions that could be asked within exams. Another drawback of showing the simplification in Petrick's method, is the rarity in which the method is used. Throughout the course of the whole project, I have only found one Boolean expression that can be used to test my code for it. The frequency in which Petrick's method is required, means that showing the steps of it is not the best for the students learning. However, the framework can be taken to provide the simplification completely algebraically.

Client feedback

To ensure that the program is fit for its purpose, the program has been given to the client, who has been asked to check through the program. To record their thoughts, they have been asked to complete a checklist with each of the AQA A-level Computer Science specification points for logic gate and Boolean algebra. Some specification points have been omitted as they do not relate to the function of the program or cannot be considered due to the constraints on the program.

Specification Item	Completed (Y/N)	Comments (where necessary)

Construct truth tables for the following logic gates: • NOT • AND • OR • XOR • NAND • NOR.	Y	"The truth table is clear. But I would prefer if it had a bigger font size"
Be familiar with drawing and interpreting logic gate circuit diagrams involving one or more of the above gates.	Y	"I liked the diagram because it was clear. Clear and easy to read"
Complete a truth table for a given logic gate circuit.	Y	
Write a Boolean expression for a given logic gate circuit.	Y	"That is very good, very impressive. Because that is very difficult to do."
Draw an equivalent logic gate circuit for a given Boolean expression.	Y	
Be familiar with the use of Boolean identities and De Morgan's laws to manipulate and simplify Boolean expressions.	Y	"It is accurate, which is the most important thing."

Some overall comments of the client are given below, as well as the specification specific comments in the table above:

> *"This program contains many helpful features which not only enable the user to evaluate logic diagrams and their attendant expressions but also to learn the basic rules and procedures of working with logic gates and Boolean algebra. I find the program to be useful not only for those new to the topic but also for more advanced users. "*

Overall, the feedback from the client was wholly positive towards the program, indicating that it is fit for use within the classroom to help students learn and improve their understanding of logic gates and Boolean algebra. The main positive comment from their feedback was that the program was useful for a variety of students with a variety of abilities.

The one negative comment given the in table was to do with the font size of the truth tables where the client found the font size to be too small. This is simple change to the values added to the current x and y position of the cell on the canvas. This change now produces the following table.

A	B	C	A.B	(A.B)^C
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	0

Figure 21: An updated truth table with a larger font size.

User feedback

As well asking the client to test the program, it was also given to a group of A-level Computer Science students to find their feedback from using the program. They were instructed to use the program and explore its features/functionality and then asked to complete small questionnaire to give their thoughts. The goal of my questions where to find their general opinion on the program and its effectiveness as a learning tool for students. The list of questions in the questionnaire is given in Appendix C.

Their feedback corresponded with that of the client, being wholly positive. Some points of note include the user interface in which they gave an average score 8.67/10 for the first question (“On the scale, how good do you think the program looks?”), indicating that the program looks good. Some of their comments on the looks of the program include “Clean aesthetic, not too cluttered. Good colour choice” and “the UI design [is] aesthetically nice”.

This addresses the issue of Logisim, discussed in the Analysis section, where the user interface of Logisim makes the program difficult to use. This is also supported by an average score of 4.33/5 for the seventh question (“Do you find the program simple to use?”).

The students found that the most useful feature within the program was the truth table generation. This could be because the process of filling in truth tables can quickly become laborious for large table and the program makes this a much faster process. This is shown by their comments in question 4, as shown below:

“Truth tables are a pain to do by hand, so having them generated is useful.”

“Flawless execution. Able to generate from expression and from [the] current diagram.”

Question 5 also addresses objective 6 (“The program should provide help to users if they encounter difficulty using the program. “). Where the students scored the pop-up gate information 5/5 for how informative the menus were. As well as objective 6, objective 4 (“the program will not error regardless of input”) was also addressed by question 9 (“Did you have any problems with the validation? Was the program robust when you were using it?”). The students gave an average score of 8.66/10. This score although, not perfect, is very much satisfactory. This means that although the program is very difficult to break, it is not impossible, as indicated by the less than ideal score. One

point of complaint for the validation was the case sensitivity for the inputs when entering Boolean expressions. This is shown by the quote “The case sensitivity for the inputs is slightly annoying”. This is not a huge issue and one that can be fixed by looking through all of the functions so that they accept case insensitive inputs.

As mentioned above, their comments where wholly positive, indicating the success of the program when its being used by students which corresponds with the comments given by the client.

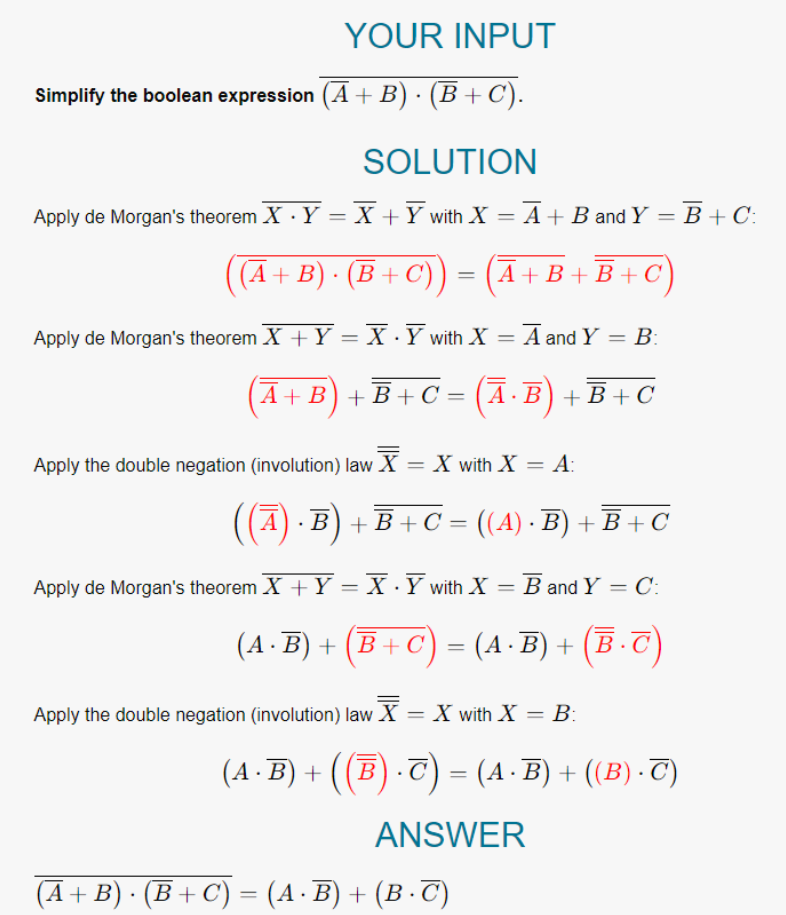
As well as giving the students a questionnaire, some other A-level Computer Science students, who did not complete the survey, were asked to draw a diagram from a Boolean expression and generate its respective truth table. This addresses objective 2, defined in the analysis. These students were chosen as they would be as new to the program as possible. This would give me the most accurate results of how easy to use the program is. On average, it took completely new students 1.5 minutes to draw the diagram and generate the truth table from the expression (A.B)^A. This is a positive result and a sign that the programs’ user interface is understandable and easy to use.

Improvements

This section contains any improvements/changes or additions to the program that could be added if the application was revisited. This is list is meant to serve as the next things that I would implement into the program given more time:

- Be able to export truth tables
    - Although it is more important that the program can export diagrams. A very nice quality of life feature will be to be able to export truth tables. This will make the process of learning logic gates even easier as the user does not have to spend time either copying the table or using snipping to crop an image of the table.
    - This feature would simply be implemented as a menu item in the file section within the menu strip at the top of the main window of the application. The exported table will be in the PNG format with a transparent background to make integration into work as simple as possible.
- Show the Boolean algebra steps during Petrick’s method.
    - As mentioned above, showing the steps of simplifying Boolean expressions using the Boolean identities and laws is extremely useful for students and teachers. This is because it allows teachers to easily show how expressions are simplified, how the laws are applied and the process of answering questions. It helps students by helping correct or validate their answers to practice questions, but also teaching them all of the rules behind Boolean algebra and simplification.
    - This feature could be implemented as a small button on the minimised expression window when it is being shown to the user. This button would open a pop-up window showing the complete steps and the process behind it, that the computer took. An example of this can be found online at

EMathHelp. An example of this is shown below:



YOUR INPUT

Simplify the boolean expression $\overline{(\overline{A}+B)\cdot(\overline{B}+C)}$.

SOLUTION

Apply de Morgan's theorem $\overline{X\cdot Y}=\overline{X}+\overline{Y}$ with $X=\overline{A}+B$ and $Y=\overline{B}+C$:

$$\left(\overline{(\overline{A}+B)\cdot(\overline{B}+C)}\right)=\left(\overline{\overline{A}+B}+\overline{\overline{B}+C}\right)$$

Apply de Morgan's theorem $\overline{X+Y}=\overline{X}\cdot\overline{Y}$ with $X=\overline{A}$ and $Y=B$:

$$\left(\overline{\overline{A}+B}\right)+\overline{\overline{B}+C}=\left(\overline{\overline{A}}\cdot\overline{B}\right)+\overline{\overline{B}+C}$$

Apply the double negation (involution) law $\overline{\overline{X}}=X$ with $X=A$:

$$\left(\left(\overline{\overline{A}}\right)\cdot\overline{B}\right)+\overline{\overline{B}+C}=((A)\cdot\overline{B})+\overline{\overline{B}+C}$$

Apply de Morgan's theorem $\overline{X+Y}=\overline{X}\cdot\overline{Y}$ with $X=\overline{B}$ and $Y=C$:

$$(A\cdot\overline{B})+\left(\overline{\overline{B}+C}\right)=(A\cdot\overline{B})+\left(\overline{\overline{B}}\cdot\overline{C}\right)$$

Apply the double negation (involution) law $\overline{\overline{X}}=X$ with $X=B$:

$$(A\cdot\overline{B})+\left(\left(\overline{\overline{B}}\right)\cdot\overline{C}\right)=(A\cdot\overline{B})+((B)\cdot\overline{C})$$

ANSWER

$$\overline{(\overline{A}+B)\cdot(\overline{B}+C)}=(A\cdot\overline{B})+(B\cdot\overline{C})$$

- Simultaneous expressions.
  - One of the drawbacks of the program, as mentioned within the constraints and limitations of the program, is that Boolean expressions can only represent one output at a time. Ie Q = (A.B). This means that drawn diagrams can only every contain one output pin at any one time.
  - An improvement that would be very interesting to implement would be the idea of simultaneous Boolean expressions. This would allow the user to enter multiple Boolean expressions within the expression input dialog and the program would be able to interpret the commonalities and linkages between the two expressions and ultimately create one cohesive diagram that could be interacted with.
    - This would require a change to the underlying data structure required for the program. This is because a binary tree could no longer be used to represent the logic gate diagram. An unweighted-directed graph would be required to be able to represent the simultaneous Boolean expressions.
    - In this case, the direction of the of the edges is what would be used to trace through the graph to make the diagram interactive and be able to draw it from a Boolean expression.
    - Another method of implementing this feature would be to create two binary trees representing each Boolean user-entered Boolean expression. A join method could then be written to identify the common subexpressions and then the graph could be generated that way.
  - It must be said that this would be a very interesting challenge to try and implement. The ability to draw diagrams containing multiple outputs would also make the program much more usable and would also be an alternative to drag-and-drop diagram creation.
- Drag and drop diagram creation.
  - One of the main reasons for using Logisim is the ability to create logic gate diagrams by dragging and dropping components from a side-panel into the canvas. This provides the user with the most

flexibility in diagram creation. It also allows the user to experiment with more complex diagrams, such as diagrams with multiple outputs and also allows built-in circuits such as half-adders and full-adders to be drawn and interacted with, which creates a program that is more complete to the A-level specification.
    - This could be implemented by converting the pop-up help menu in the bottom-left corner of the current application into the side panel where the gates could be dragged from. This improvement would make the program more versatile. However, more complex wire drawing logic is required. This is because the position of each of the nodes is not known and also the components may be dragged around the canvas once the wire has been drawn meaning the wire must snap to the new position of the gate.
    - This would be a significant amount of work to add to the current application, but it would provide a very useful feature to improve the learning of the students.
- Help-pages
    - Currently the program only provides resources on the logic gates that are within the A-level specification. This is something mentioned in the student questionnaire (Appendix C). Having documentation on how to use the program would also provide another way of getting help when students are confused. This would improve the autonomy of the students to be able to learn and fully use the application.
    - This would be implemented as another item on the main menu strip at the top of the main window of the application. This would then have a drop-down menu to open more feature specific guides.
    - These guides would detail on how to achieve certain tasks within the program such as drawing diagrams, generating truth tables and file handling.
        - These guides would be complete with each images and videos to provide as comprehensive resources as possible.
- Truth tables without steps.
    - Currently within the program there is no way to generate a truth table without the steps involved. This would be a small feature that would produce more concise tables. However, it should be noted that most questions that refer to truth table show the constituents steps involved. This means that this addition will simply be for the quality of life of the user and provide a huge benefit to students revising for exams.

Aside from the points made above, there are also many small things that could be improved such as small visual changes, for example grid squares on the diagram canvas, improved visuals on the buttons and menus. Making the status bar, more visible, descriptive, and better looking and other changes like the ones listed.

Closing thoughts

Finally, I believe that the application fully satisfies the core objectives, defined at the beginning of the project. Additionally, I am extremely happy that the program is fit for use within a real classroom environment, as shown by the comments from the client. The diagram minimisation is also something that can be used by the students to aid their learning and understanding of Boolean algebra. The extensions given in the previous sections also give a good starting point for exploring new programming techniques and furthering my own learning. In conclusion, creating, researching, and programming this project has been an extremely rewarding experience and I am extremely happy with the result.

Appendices

Appendix A – Interview transcript

When you are teaching logic gates to students, do you find questions or demonstration to be the more effective method in aiding the students learning?

Depends what aspect I am teaching. I mean from the beginning you would start with switches and the very basics of how they work. I then build the idea of logic gates and the actual symbols used to represent logic gates. I then move

onto truth tables and the actual functions of the gates. After that, gates that are joined together in multiple different configurations.

Do you want to be able to create truth tables with my program? How would this improve your teaching?

Yes, truth tables I think are important. Many exam questions are based on truth tables. Some ask you to fill in truth tables, and I think being able to use the truth tables shows that you actually understand what is going on with the gates themselves. If you have a more complicated setup of gates having the intermediate outputs is very useful and that can only be properly represented using truth tables.

If you would like truth tables, how would you like these to be structured? Columns of gates such as XOR B or just input or output?

No, I would. I would like it to have the labels. Which is dependent upon the gates in the diagram. So, I would like to say for example, Column A, Column B and then Column A AND B. So, all of the intermediates and the input columns are shown with the title of the columns as well. So that you can relate back to the diagram of logic gates.

Which, if any, pre-made circuit would you like to be able to use within my programme?

Well, it depends what age group it is for. You know, obviously the three basic AND, OR and NOT gates. Then at the higher levels, you would want adders, half-adders, flip-flops, D-type flip-flops. In the lower years, I am not that fussed about pre-made circuits. The only point where you would have to study pre-made circuits is KS5 (A-level).

Do you want to be able to save and load circuits into and with my programme?

I would like to be able to save the circuits so if you are halfway through a lesson, and you have not finished the diagram you can carry on. I would also like to be able to output some sort of image so that I could then put the diagrams into written work such as a Word document. So, I would like to save, load, and also export diagrams.

Do you want to explore images of drawn circuits? Would you prefer a specific file format?

Well, I think the most common file formats would be the best ones. Possibly a PNG because that would support transparency for the background so that you can easily place diagrams into Word documents. I think I would rather have a PNG output because it is versatile.

Do you want to be able to input Boolean and/or logical expressions and then generate circuits from them? Do you find one type of expression to be more useful than the other? What notation would you use for these expressions?

Well, for Boolean algebra, I would standard notation. It is the ANSI/IEEE standard 91-1984 because it is specified in the A-level specification the type of notation that you would be expected to know. And then putting in an algebraic expression Boolean expression. Then if it can generate the circuit diagram. Would it generate the simplest diagram? Because that would be ideal.

Do you want to minimise be able to minimise circuits?

Well, I would like it to be able to generate the circuit diagram appropriate to the age group. So, for example, you do not learn EOR gates at KS4. So, I would not want it to output one with it or an XOR, at key stage 4. But I would like it to do that that key stage five. So really, I would want circuit diagrams appropriate to the age group.

Do you want a grid when drawing circuits? If so, would you like this to be toggle able when exporting images of circuits? Would you like the ability to add truth tables to exported images?

That one is easy. Yes, yes, and yes.

Do you want to be able to generate logic circuit questions? What format would these questions take, and would you find it essential?

That is much harder I mean. How when you say questions it you know you. I am not sure what you mean because there are so many ways of asking a question. Like you could show them a picture and say what kind of gate is this? Or you could show them inputs, outputs. And a gate, the same question, what will the output be? Well, that is what I mean, is that there's different kinds of questions. So, it would be nice if it generated questions, but not essential.

What format would these questions take?

Again, it is specific to the age group, isn't it? You know, I am not going to ask the same question of a sixth form student as I'm going to ask a year seven student. So, it would have to be specific to the age group and the form. Well, I mean that is how long this piece of string. I do not think there's any specific answer to that.

Would you like a splash screen where the user can select their ability? Or a built-in menu?

Something like that. That is a very good idea.

Would you like a splash screen?

Whichever works best. In Game Maker, for example, you can go file and you can select simple or advanced mode. So, you could go file then simple, medium, or advanced or have a splash screen with 3 big buttons. One that says 7/8/9, one that says 10/11, and one that says 12/13.

What are some of the drawbacks / issues that you find when using Logisim to teach logic gates?

Scaling sometimes can be an issue. So, if I want to zoom in to get it big enough, sometimes I am not happy with how much it's showing me. The wires, joining wires up and going from the input points on the diagrams to the output point on another one. You know, just drawing in the wires. That can get messy and difficult to draw. It is also really user generated as well. If the user does not line things up, things can get messy. You have to think ahead when you are drawing a circuit diagram on Logisim, so everything lines up properly otherwise you end up with a spaghetti junction.

So maybe if the program could guide them, on where they put things, you know, when you use like a drawing programme like Photoshop it gives you these faint blue lines or faint pink lines that tell you when things are centred or tell you when things are aligned in a certain way. So maybe some sort of way of them not being allowed to put things higgledy piggledy and then end up with a spaghetti kind of diagram.

A feature that could clean a circuit?

Well, that can clean it up in terms of wiring and also clean up in terms of alignment, so the circuits aligned nicely like top and bottom, front to back.

All the any key differences within the key stage 3 logic gates specification and the GCSE logic gate specification? If so, what are these?

We only teach them the basic of switches and then AND, OR and NOT gates. But not really much else.

What are some common errors that students have when drawing logic circuits with software or by hand? Would a validation button that checks for these errors be essential?

Well, that that goes back to what I said really. I mean, when you say errors, is it errors in the logic they have used? Or is it errors in the diagram they have drawn? You know, if it is the picture they've drawn, it's just not thinking ahead and not lining things up so that you can connect them easily. In terms of logic, that is a more difficult question. That is more about how they learn it and what's going on in their head.

Are there any common logical errors that people make?

Maybe NOT gates, they do not always understand the simplest gate and that it inverts the output. You know, so maybe like the concept of inversion but otherwise I can't really say there's any specific type of error. The validation could check that the diagram has been drawn properly but aside from that. You should validate the drawing of the circuit.

Have you found incomplete logic circuits to be an effective way for students to learn?

Yes, sometimes. Sometimes it can be. You can give an incomplete circuit and say this is what I expect the output to be. What kind of gates will give me the output? So yes, that can work.

Would you like the ability to generate logical expression questions be solely for a level students?

Explain what you mean by that. For example, could a GCSE student draw a diagram and get a logical expression from it for their own understanding, their own their own learning? Would you like that to be a restricted feature or just kind of expressions in general to be a restricted thing? I do not know. I do not think it should be restricted at all. I think that if you made a circuit and that circuit gave you the expression and the truth table then that is useful across the board. I mean with the lower year groups, you just do not have to talk about it, you know?

Would you restrict any features from certain age groups?

Yes, like I said I'd restrict things like the gates that they don't need to use. Hmm, because I do not want to confuse them because then you said put putting more symbols and shapes and stuff into the equation, so I would definitely restrict them according to what was available to them. So, if there was “a drawer” and in that there was all of the gates the students need and only them that would be helpful. One thing about Logisim is that you get a menu down the left hand side, and you get a lot of pins, inputs, outputs, and it starts to get confusing and cluttered. So, something that is a bit plainer and clearer for the younger years, and something a bit more fully featured for the older ones.

So, if there was like a draw, and you know, in that draw there were the gates that they need and only the gates that they need. One thing about logicism is that you get this menu down the left hand side, you get a lot of pins, input pin, output pin, and it starts getting confusing and cluttering. So, there is something a bit plainer and clearer for the younger years, and something a bit more fully featured for the older ones.

Have you found any features that you would like to be added to Logisim? What are these? How effective are they? Why would you add them?

There is not much I would add. I would like it to be better structured and get rid of the clutter lower down. Introduce more clutter higher up, and just make it more appropriate and easier to use and draw Shapes snap better, lines draw better, and it just becomes a faster, more elegant user experience.

Do you prefer lots of sub menus and a larger workspace or a smaller workspace with a with more side buttons?

That is what I'm saying to you about scaling. If there is a circuit with four or five columns of gates, it's so small. That I find it difficult to see and snap to the wires. So, it is how the program scales it. I think it is better to have a bigger work surface and a more compact menu. It is mainly just uncluttering the interface of Logisim.

Would this software be used by students within lessons, or would it be solely used for demonstration by yourself?

I would like the software to be general purpose. I would like the students to have the same interface as me so they can see me demonstrating something and then they can replicate it themselves. But it would be used by both students and me. You could have something like a teacher/ expert mode?

What would be some example tasks that students would be asked to complete?

Being given a series of gates joined together and having to complete a truth table or something of that nature.

Are there any features slash circuits that are beyond the A level specification that you would like to see?

No.

Would you be interested in something like a truth table style of questions?

Yes, I definitely would.

Would the programme be more geared towards diagram drawing or questions?
The drawing aspect is of higher priority, but it would be nice to have the questions.

Are there any special features that or special things that you would like within the program?

No, it is clear. Anything special to use, I mean. I just want it to be a more streamlined and look more modern unlike Logisim. One thing about Logisim is that exported images are pixelated and so that would be a small improvement, but it is not a deal breaker.

I would like the menus to be a bit clearer as they can be slightly too vague/opaque. A lot of features are also for electronic engineers etc. and so there is a lot of stuff that we just never use. So, I would like it more geared towards Computer Science.

Appendix B – Student questionnaire

Below contains the list of questions and a table containing all of the students' answers to the respective questions.

1. How do you revise logic gates? What are some effective techniques that you have found?

ID	Response
1	Have a Boolean algebra expression. Convert it to logic gates.
2	Mostly past paper questions, going over certain gate combinations like an adder etc.
3	I have not learnt logic gates yet.
4	I revise logic gates by drawing them and then making a table to see the outputs from the inputs
5	I do not.
6	I do not.
7	i do not.
8	i do not understand.
9	Occasionally look at diagrams of logic gates and answer questions/fill out tables of the outputs
10	I read about them and do a couple questions
11	I do not revise logic gates

12	I have not learnt them yet.
13	I try drawing different logic gates.
14	I have not revised this.
15	I do not understand
16	Looking at the symbols and remembering the names for them
17	Seneca, getting other people to ask me questions on the topic etc.
18	Experimenting with them, with a diagram of gates with inputs at the beginning and outputs at the end.
19	going on websites that will do quiz's and try to complete as much as I can going on websites or doing some revision on Multiwingspan.
20	When revising the AND logic gate, I tend to imagine the inputs as positives and negatives. What I mean by this is that a condition that has been met is positive and one that has not is negative, and the output is a positive that can only be outputted if there are two positive conditions.
21	I do not revise logic gates.
22	I do not
23	I do not revise logic gates much specifically but if I did more, I would use textbooks as well as a website called Lucid Chart which has the symbols so you can create your own logic gate program which could be useful to use and put in practise
24	I draw them by hand when doing past papers
25	Nand game is enjoyable, but also challenges your knowledge of logic gates.
26	Practice drawing individual gates and circuits from memory. Practice questions whereby you have to use gates to complete a circuit to solve a problem
27	Practice questions on paper, I never use software to help answer the question. I do like to have a visual of the logic gates as it is tough to picture if they are not provided in a question.
28	Usually, my revision of logic gates comes from past papers. It is not really a topic I tend to revise on its own.
29	Repeatedly solving problems involving tracing and drawing them
30	I personally find that doing practice questions is the most effective form of revision on logic gates.

2. What would you want the product to look like? A colourful design or something more simplistic? What choices would you make?

ID	Response
1	Simplistic to make the menu less cluttered. Not too colourful to look more professional but there could be a light mode and dark mode. The only exception to the colourlessness could be the connection between gates. A window containing all the gates which can be placed to any side. Buttons to generate Boolean algebra expressions and truth tables. Button to replace the current logic gates with custom Boolean algebra expressions.
2	A colourful part set could be helpful in identifying parts and spotting patterns in circuits. But a more simplistic look would be truer to real examination conditions, so if possible both!
3	I am not sure.
4	I think a more simplistic design would make sense.
5	I like a simplistic design that is easy to understand for the user.
6	Simplistic and easy to understand with helpful guides.
7	I would want the program to look colourful similar to python syntax - e.g., NOT gates are red, AND gates are blue, etc
8	I would like the product to look interesting for a user.
9	Simple with minimal colours
10	Something that is easy to operate and does not distract from what it is meant to do
11	An easy to read simplistic and understandable design
12	Simplistic to show the product better.
13	A simplistic design.
14	I would make it colourful enough to allow it to be eye catching, but simple to use/read.
15	A colourful design

16	More simplistic
17	Simple design, could use colours to represent important information (like how specific functions of coding are highlighted in different colours in an IDE).
18	Colours would help, for instance grey for the wires and a thicker black line for the gates, and a red colour for wires which are powered on.
19	a colourful design so it looks a bit more fun and interesting
20	I would choose a mix between a colourful and simplistic design. Something that makes the product look interesting, but not eye-catching enough to distract the user from the purpose of the product.
21	More simplistic design as it would make it easier to read and make it harder to get distracted from the colours.
22	I like very colourful stuff because it makes it easier to understand and makes it look nicer
23	I would like something that has shortcuts and easy to use and a cool design or colourful one instead of a black or white screen for example a background of a car etc. You could also have a shortcut that allows the user to access the logic symbols and gates that could have a symbol and a name to remind you of it. And possibly an area that explains each logic symbol/gates and how to use the summarized so it isn't a giant paragraph as that would not look user friendly and people might be discouraged from reading or using the function.
24	Something simple so it does not get in the way
25	Simplistic, with a minimal UI style.
26	Perhaps have a choice between them, I would prefer a simplistic style however.
27	A simplistic design, not overloaded with tools and easy to understand what the tools do.
28	I feel like a colourful design would be more enticing. However, a simplistic design could stop me from getting distracted easily.
29	A more simplistic design would make for a more effective study tool as it would be more similar to questions you would have to deal with
30	Something relatively simplistic would be nice.

3. Do you draw logic circuit diagrams by hand? What is your opinion of this?

ID	Response
1	I sometimes do this because logic gates are very simple to draw, and it is quick to make an entire logic circuit.
2	It is generally faster to do a rough sketch by hand than making a circuit in an application like Logisim so yes
3	I have not learnt logic circuit.
4	I draw them by hand and find it quite easy, but sometimes they look rough without a ruler
5	No.
6	A little bit, it is alright.
7	No, because I find it annoying to draw the gates themselves
8	No, I do not.
9	When I do draw them, it is difficult to do by hand
10	Sometimes, it allows me to visualize what is going on
11	No seems like a good idea to do so though
12	No, but it can be useful if you do not know how to draw them on your computer quickly.
13	Yes, but drawing by hand is quite untidy.
14	I have not drawn one recently enough to say.
15	I do not.
16	Yes, but it is easier on a computer
17	Yes, it is fine for me.
18	No, my handwriting is bad, and I cannot draw very well and a program to "run" the logic gates with certain inputs is much more work by hand.
19	maybe by hand so you enjoy making the program
20	It is easier to draw logic circuit diagrams via a computer. I find it more efficient and more likely to be tidy and less wonky.

21	No, I do not draw logic gates.
22	I do not find it useful
23	No as I prefer to do this online as it feels better, looks better and it would be a pain to draw it over and over again.
24	I do, it is slow and hard to validate
25	Yes. It is easier than working on a computer in most cases.
26	Drawing it by hand is irritating as errors are hard to remove and gates can look inconsistent
27	Yes, I usually draw by hand. I do not have a problem with this apart from it sometimes getting messy if a mistake is made.
28	I do. It is not that bad as logic gates aren't that complex. However, gates like XOR and XAND and more difficult to do.
29	Yes, and it can get very complicated and messy very quickly
30	Drawing logic gate diagrams by hand is easier but doing it on the computer works as well and I am more likely to use logic gate software.

4. Have you used Logisim? Is so, what are some issues that you have with it? What are some ways that you find it difficult to use? Why does this make Logisim difficult to use?

ID	Response
1	I have not used it sorry.
2	Yes, it is far too clunky for small circuits, does appear to be useful for more complex circuits but that's about it, it's generally not a good or intuitive application.
3	I have not used Logisim.
4	No
5	No.
6	Yes, there's no issues with them. It is pretty easy to understand.
7	no
8	I have not heard of Logisim.
9	No
10	I don`t think that I have used Logisim much, though it can be a bit difficult to remember what some of the logic does
11	I have not used Logisim
12	No, I have not used Logisim.
13	No
14	I have not used it.
15	I do not know
16	I do not know
17	I have not used Logisim
18	No.
19	I do not know
20	I have not used Logisim
21	Yes, it is quite confusing and not very easy to use for beginners. There is not a very useful tutorial for making things in Logisim.
22	I think Logisim is a good program, and it does what it is made for well
23	No, I have not used Logisim before
24	Yes, Logisim is awful to use, setting up input buttons And Some functions are super hidden even though they are important for the GCSE and A-level specifications.
25	The fact wires cannot cross makes larger circuits almost impossible to make.
26	Yes briefly, the drag and drop tray can be confusing as there are a lot of options
27	Yes, it is very crowded, and tools are hard to understand. Not a fan of the input and output nodes. The drag and drop system is not the easiest to use. Colour coding is not the best.

28	Very briefly and only is school lessons. It is alright but the interface isn't very helpful, and I often find myself having to spend a long time finding a specific item. Also, when there are a lot of wires, I can lose track of what is connected.
29	Its UI is slightly dated but it is mostly a solid program for study purposes
30	Logisim can be slow to start as you have to figure out how to use it initially, making it somewhat less convenient than it could be.

5. Conversely, what are some parts of Logisim that you find useful for your learning? Are there some features that you particularly enjoy using? What are these? Why do you find them useful for your learning?

ID	Response
1	I have not used it sorry.
2	Not many, from my brief time the only thing that I find useful is the number of gates you can link together to make big circuits but even then, it is annoying manually turning on and off individual inputs.
3	I have not used Logisim.
4	I do not know
5	I have not used it.
6	I like drawing the logic gates and the dots are helpful
7	I have not used it.
8	I do not know.
9	N/A
10	
11	I do not know
12	I have not used Logisim.
13	I have not used it before.
14	I have not used it.
15	No
16	I do not know
17	N/A.
18	
19	I do not know
20	I have not used Logisim
21	It can be used to create simple logic simulation which can be quite useful for visualizing things.
22	I do not use it very often so I'm not sure
23	I do not know
24	It is better than by hand and is fairly simple after you get the hang of it
25	It honestly lacks usability to the point where it is not enjoyable.
26	The way it creates a truth table for your output can be useful
27	I like how lines between gates connect no matter where the gates are placed.
28	Logisim overall is a good software which provides the ability to make complex circuits
29	It shows traceable paths of logic with given outputs and is relatively simple to use
30	Once you have gotten used to Logisim, it can be used to create detailed and effective diagrams due to the wide range of tools.

6. How would you improve Logisim? What are some features that you would add to the program?

ID	Response
1	I have not used it sorry.
2	Add NUM pad capabilities to allow you to either bind the inputs so they turn on and off, or for parts like NUM 1 is an AND gate, needs to be much faster to create simpler circuits, essentially add more shortcuts and make the UI more intuitive
3	I have not used Logisim.
4	I do not know.

5	I have not used it.
6	I do not know
7	i have not used it
8	more specific inputs
9	N/A
10	
11	I do not know
12	I have not used Logisim
13	I have not used it before.
14	make sure that the display is easy to use.
15	I do not know
16	I do not know
17	N/A.
18	
19	I do not know
20	If it is not in Logisim, a feature that can show what all of the logic would do in the circuit.
21	Make it more beginner friendly as it can be hard to learn. A in-depth tutorial.
22	It has everything it needs I think
23	I do not know
24	Remove some of the menus to make it easier to get to some gates
25	Allow wires to cross, make it drag and drop similar to Nand game.
26	A tutorial of sorts
27	More beginner friendly, maybe a help page
28	I feel like improvements to Logisim are too large to just "implement". A lot of the changes I would make would require GUI overhauls
29	I would update the UI and add the ability to be able to convert the drawn logic gates into a written logical notation form
30	I would change the UI to feel less cluttered and more organized, meaning it is easier to navigate and therefore use.

7. When using help pages, what information would you like to know? What would you find most useful?

ID	Responses
1	A description of every logic gate. A description of Boolean algebra. A guide to how to use the application. (especially how to place logic gates)
2	A very dumbed down explanation of how parts work, what inputs they accept and what they spew out, for tool tips again dumbed down and Clear, they need to make sense so you cannot use terminology too hard to understand, this is especially true for beginners
3	I am not sure.
4	What symbol corresponds to what and what it does.
5	Information about how to use things, and examples of them being used.
6	The types of logic gates
7	it would be helpful if there were descriptions of all of the gates included, along with examples of how they are used. it would also be helpful if there were examples of different systems
8	what each logic gate means, how to create certain programs, etc
9	Explanation of the gates and what they do with examples. Show how to add, remove, and connect gates in the program
10	What each of the logic does
11	I would like how to pages and some explanation of working bits of code
12	The way to string together certain gates to create certain patterns such as whether you go and then or then and.
13	I am not sure.
14	How to access everything, anything I would need to know if I had no understanding of the topic

15	I do not know
16	Things to help you remember the gates
17	I would like examples to go with descriptions, and good, detailed explanations so people who are completely new to the topics understand how to use the program and the logic gates.
18	How the gates work, how the program works.
19	things that will improve your learning
20	Examples that help to better understand what conditions have to be met for the gates.
21	The basic structure of programs and what each of the parts do e.g., what 'and' does.
22	Information on how to use the program and also what everything does
23	Could have a function that allows the user to take a quick quiz e.g., ten questions to refresh your memory on it and make it fun instead of just looking at information and maybe have a bigger quiz for revision e.g., 30-40 question quiz.
24	I would like it to be well lay out so I can find what I am looking for
25	Documentation on each of the gates.
26	A navigation tool that tells me where I can find different buttons etc
27	Explanation of what the gates output depending on their input.
28	I like that there is a lot of information so if I do have a major issue there is always someone who has had the same thing and has found a solution
29	A description of different types of gates and their usual outputs would help as a reminder along with the associated notation for these gates
30	A short summary on how to use various tools.

8. Do you find answering questions to be a good way to revise logic gates?

ID	Responses
1	Yes
2	Yes
3	I am not sure.
4	Yes
5	Yes, because it keeps me focused.
6	Yes
7	I do not know
8	Yes
9	Yes
10	Yes
11	Yes
12	Yes, for it would make you think through the gates which is the main part of the logic gates.
13	Yes
14	No opinion
15	No
16	Yes
17	Yes, but a variety of question types so multiple choice but also questions with inputting your own answer.
18	It can help with knowing what the questions will be like in exams but probably wouldn't be as effective as using them.
19	I think it is useful but if u don't know what to write then you have to do some research to get the answer if we had like a page before the questions that would be helpful
20	Yes
21	I do not know.
22	I do not know.
23	Yes, I would say answering questions is more fun, interactive, and memorable especially when i get stuff wrong. In the question before I suggested making a quiz function, you could have different ending when you get a question wrong e.g., Try again next time, this would make your quiz more memorable if you were to do it.

24	Yes, the ultimate way
25	Yes
26	Yes
27	Yes
28	Yes. As stated before, past papers are the only way I revise logic gates as I do not isolate the topic and revise it
29	Yes, it is the best way to revise them in my opinion
30	Why yes, I do in fact.

9. A-level students only. Would you complete logic expression simplification questions? Would you find this helpful?

ID	Responses
1	Yes, but the GUI would have to be very easy to use. Otherwise pen and paper is a better alternative (at least for me personally)
2	Kind off, simplifying expressions can be useful to save in processing time but if it is not intuitive as to why this works or why I'm doing it no.
3	I am not sure.
4	N/A
5	.
6	I do not know
7	N/A
8	.
9	Not A-level
10	N/A
11	I do not know
12	I am not in A-level.
13	I am not an A-level student.
14	.
15	No
16	Not an A-level student.
17	N/A.
18	
19	I do not know.
20	I am a year 10
21	?
22	Year 10
23	I am not an A-level student.
24	Yes, if they were readily available and intuitive
25	I am not particularly familiar with logic expressions.
26	Not particularly helpful for exams, but it would consolidate my understanding of logic gates
27	Yes, this would be helpful, especially if it is clearly pictured.
28	Yes, I believe it would be helpful
29	Yes, and I would find a way to do them simply with visual representation of the steps taken helpful
30	Probably, yes.

10. What are some features that you would like within this program?

ID	Responses
1	Primary Truth tables Logic gates to Boolean algebra expression Boolean algebra expression to logic gates

	Extension Example logic circuits Export logic circuits to PNG or JEPG
2	Logisim but better essentially.
3	I am not sure.
4	I think we should be able to drag drop the different logic gates and have the lines between them be dynamic so they never go diagonal, and we should be able to see the output given from a certain input.
5	.
6	I do not know
7	
8	A help statement
9	After adding and connecting gates, a button that displays a table of inputs and what outputs they would give. A light/indicator that turns on when True.
10	The ability to see all of the logic that you need and what you can do with them
11	I do not know
12	I am not A-level.
13	I have not use it before.
14	No opinion.
15	No
16	I do not know.
17	N/A.
18	A list of tasks which asks you to do certain things, e.g., build a simple calculator, and checks if it works (and how you could improve it). This would help to build understanding of how to use them.
19	I do not know
20	I have not used Logisim
21	I do not know.
22	Colour
23	Mini quiz function, logic symbol information, shortcuts, cool background/colourful.
24	Difficulty levels so that I can build confidence with these questions
25	Tasks of varying difficulty.
26	Hotkeys for common gates, pre-built circuits such as an adder for example, questions asking you to create a circuit for a problem,
27	Produce a truth table from the circuit made within the program.
28	The ability to save and export any of my logic circuits.
29	The visual representation and ability to simply use it to solve harder questions.
30	The ability to export created diagrams.

Appendix C – Student questionnaire for program testing.

1. On the scale, how good do you think the program looks. *

1	2	3	4	5	6	7	8	9	10

2. What is your reason for this score? Please identify specific reasons for your opinion etc. *

Enter your answer

3. Which feature do you find most useful for your learning? *

- ◯ Diagram drawing and interactivity
- ◯ Truth table generation
- ◯ Diagram minimisation
- ◯ Help menu

4. What is your reason for this choice? *

Enter your answer

5. Do the help pop-up menus cover all of the information necessary for each of the logic gates? *

1	2	3	4	5

6. What are some things, If any, that you would add the help menus to make them more useful?

Enter your answer

7. Do you find the program simple to use? *

1	2	3	4	5

8. What is the reason for this? *

Enter your answer

9. Did you have any problems with the validation? Was the program robust when you were using it? *

1	2	3	4	5	6	7	8	9	10

10. What are some limitations of the program that affect your learning that you have found whilst using the program? *

Enter your answer

11. What are some new features or changes that you would make to improve the form/function of the program? *

Enter your answer